Catching Numeric Inconsistencies in Graphs

WENFEI FAN, University of Edinburgh & BDBC, Beihang University & SICS, Shenzhen University
XUELI LIU, College of Intelligence and Computing, Tianjin University
PING LU, BDBC, Beihang University
CHAO TIAN, Alibaba Group

Numeric inconsistencies are common in real-life knowledge bases and social networks. To catch such errors, we extend graph functional dependencies with linear arithmetic expressions and built-in comparison predicates, referred to as numeric graph dependencies (NGDs). We study fundamental problems for NGDs. We show that their satisfiability, implication, and validation problems are Σ_2^p-complete, Π_2^p-complete, and coNP-complete, respectively. However, if we allow non-linear arithmetic expressions, even of degree at most 2, the satisfiability and implication problems become undecidable. In other words, NGDs strike a balance between expressivity and complexity. To make practical use of NGDs, we develop an incremental algorithm IncDect to detect errors in a graph G using NGDs in response to updates ΔG to G. We show that the incremental validation problem is coNP-complete. Nonetheless, algorithm IncDect is localizable, i.e., its cost is determined by small neighbors of nodes in ΔG instead of the entire G. Moreover, we parallelize IncDect such that it guarantees to reduce running time with the increase of processors. In addition, to strike a balance between the efficiency and accuracy, we also develop polynomial-time parallel algorithms for detection and incremental detection of top-ranked inconsistencies. Using real-life and synthetic graphs, we experimentally verify the scalability and efficiency of the algorithms.

CCS Concepts: • **Information systems → Inconsistent data**; **Data cleaning**;

Additional Key Words and Phrases: Numeric errors, graph dependencies, incremental validation

ACM Reference format:
Wenfei Fan, Xueli Liu, Ping Lu, and Chao Tian. 2020. Catching Numeric Inconsistencies in Graphs. *ACM Trans. Database Syst.* 45, 2, Article 9 (June 2020), 47 pages.
https://doi.org/10.1145/3385031

Fan is supported in part by ERC 652976, Royal Society Wolfson Research Merit Award WRM/R1/180014, EPSRC EP/M025268/1, Shenzhen Institute of Computing Sciences, and Beijing Advanced Innovation Center for Big Data and Brain Computing. Lu is supported in part by NSFC 61602023. Liu is supported in part by NSFC 61902274.
Authors' addresses: W. Fan, University of Edinburgh & BDBC, Beihang University & SICS, Shenzhen University, 10 Crichton Street, Edinburgh, UK, EH8 9AB; email: wenfei@inf.ed.ac.uk; X. Liu (corresponding author), College of Intelligence and Computing, Tianjin University, 135 Yaguan Road, Jinnan District, Tianjin, China, 300350; email: xueli@tju.edu.cn; P. Lu, BDBC, Beihang University, 37 Xue Yuan Road, Haidian District, Beijing, China, 100191; email: luping@buaa.edu.cn; C. Tian, Alibaba Group, 969 West Wen Yi Road, Yu Hang District, Hangzhou, China, 311121; email: tianchao.tc@alibaba-inc.com.

0362-5915/2020/06-ART9 $15.00
https://doi.org/10.1145/3385031

1 INTRODUCTION

A variety of dependencies have recently been studied for graphs [9, 21, 31, 33, 43, 73]. These dependencies are often defined in terms of graph patterns and aim to capture inconsistencies among entities in a graph. They are useful in, e.g., knowledge acquisition, knowledge base enrichment, and spam detection in social networks. However, semantic inconsistencies in real-life graphs often involve numeric values. To catch such errors, arithmetic calculation and comparison predicates are often a must. These expressions are, unfortunately, not supported by existing graph dependencies.

Example 1.1. Consider inconsistencies taken from real-life knowledge bases and social graphs.

(1) Yago. It is recorded that an institute BBC Trust was created in 2007 but destroyed in 1946, as shown in graph G_1 of Figure 1. To detect the error, we need to check whether wasDestroyedOnDate - wasCreatedOnDate $\geq c$ for a non-negative constant c. However, neither arithmetic operator $-$ nor comparison predicate $\geq$ is supported by existing proposals for graph dependencies.

(2) Yago. A village Bhonpur in India is claimed to have 600 females and 722 males, but its total population is 1,572 (see graph G_2 of Figure 1). To catch this inconsistency, we need an arithmetic equation femalePopulation +malePopulation = populationTotal.

(3) DBpedia. There are two cities, Corona and Downey, in California. Based on the 2014 population census, it is known that Corona has a larger population than that of Downey. However, Downey is ranked ahead of Corona in population (11th vs. 33rd; see graph G_3 of Figure 1). The inconsistency should be checked by using a condition that x.population $<$ y.population implies x.populationRank $>$ y.populationRank, where x and y denote two different places.

(4) Twitter. Suppose that two accounts refer to the same company. If the two substantially differ in the numbers of their followers and followings, then the one with less followers and followings is likely to be a fake account [57]. To specify this rule, we need a condition $a\times(x.\text{follower} - y.\text{follower}) + b\times(x.\text{following} - y.\text{following}) > c$, for accounts x and y, and constants a, b, and c. The condition is specified by both arithmetic expressions and comparison predicate. It helps us find, e.g., fake account NatWest_Help depicted in graph G_4 of Figure 1.

The example raises several questions. How should we extend graph dependencies to catch numeric errors? Does the extension make it harder to reason about the dependencies? If so, how can we strike a balance between the expressive power and complexity? Can we make practical use of such an expansion to uniformly catch inconsistencies in real-life graphs, numeric or not?

Contributions & organization. This article tackles these questions.

(1) NGDs. We propose a class of numeric graph dependencies, referred to as NGDs (Section 3). NGDs are a combination of (a) a pattern Q to identify entities by graph homomorphism and (b) an attribute dependency $X \rightarrow Y$ on the entities identified. They extend graph functional dependencies (GFDs [31, 33]) by supporting linear arithmetic expressions and built-in comparison predicates $=, \neq, <, \leq, >, \geq$. We show that NGDs are able to catch numeric inconsistencies commonly found in real-life graphs. Moreover, they subsume GFDs [31, 33] and relational conditional functional dependencies (CFDs [26]) as special cases. Thus, they capture inconsistencies that can be detected by GFDs and CFDs, besides numeric errors that are beyond the capacity of GFDs and CFDs.

(2) Fundamental results. We study two classical problems for NGDs (Section 4), stated as follows:

- The *satisfiability* problem is to decide whether a given set Σ of NGDs has a model, i.e., a graph satisfying all NGDs of Σ.

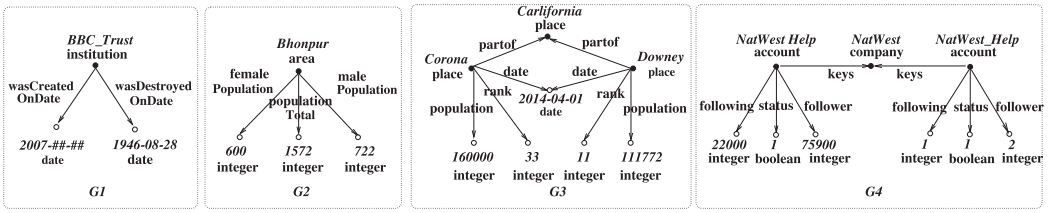


Fig. 1. Numeric inconsistencies in real-life graphs.

- The *implication* problem is to decide whether a set Σ of NGDs entails another NGD φ, i.e., for all graphs G that satisfy Σ, G also satisfies φ.

These problems are not only of theoretical interest, but also find practical applications. The satisfiability analysis enables us to check whether a set of NGDs is consistent themselves before the NGDs are used as, e.g., data quality rules. The implication analysis helps us remove redundant rules φ that are logical consequences of a set of Σ of rules and optimize the data-cleaning process.

(a) We show that the increased expressive power of NGDs comes with a price. Their satisfiability and implication problems become Σ_2^p-complete and Π_2^p-complete, as opposed to coNP-complete and NP-complete for GFDs, respectively [31, 33]. The complexity bounds are robust to a great extent: They remain Σ_2^p-hard and Π_2^p-hard, respectively, even when only equality $=$ is used, in the absence of $\neq, <, \leq, >, \geq$, or when no arithmetic operations are used at all. These tell us that unless P = NP, it is harder to reason about NGDs than about GFDs.

(b) We show that if we expand NGDs by allowing non-linear arithmetic expressions, then both the satisfiability and implication problems become undecidable, even when the degree of the arithmetic expressions is at most 2 and even in the absence of comparison predicates $\neq, <, \leq, >, \geq$.

The undecidability results justify the choice of linear arithmetic expressions. That is, NGDs strike a balance between expressivity and complexity when arithmetic and comparison are a must.

(3) Practical applications. We develop techniques for detecting inconsistencies in real-life graphs, numeric or not, by employing NGDs as data quality rules (Sections 5 to 7).

(a) We show that the *validation problem*, i.e., deciding whether a given graph satisfies a set of NGDs, is coNP-complete. The complexity is the same as for GFDs [31, 33]. That is, NGDs do not complicate the process of error detection. Better still, the parallel algorithms developed in Reference [33] for detecting errors with GFDs can be readily extended to NGDs, retaining the same complexity.

(b) In light of this, we focus on incremental inconsistency detection in graphs, a problem that has not been studied before, to the best of our knowledge (Section 5). Given a graph G and a set Σ of NGDs, suppose that we have already identified a set $\mathrm{Vio}(\Sigma, G)$ of violations of Σ in G, i.e., entities in G that violate at least one NGD in Σ. We want to find *changes* $\Delta\mathrm{Vio}$ to $\mathrm{Vio}(\Sigma, G)$, such that

$$\mathrm{Vio}(\Sigma, G \oplus \Delta G) = \mathrm{Vio}(\Sigma, G) \oplus \Delta\mathrm{Vio},$$

where ΔG is a set of updates to the graph G, and $X \oplus \Delta X$ denotes X updated by ΔX.

The need for incremental detection is evident. Real-life graphs G are often big, e.g., the social graph of Facebook has billions of nodes and trillions of edges [41]. Error detection is expensive (coNP-complete). Moreover, real-life graphs are constantly changed. It is often too costly to recompute $\mathrm{Vio}(\Sigma, G \oplus \Delta G)$ starting from scratch in response to frequent updates ΔG. These reasons highlight the need for incremental algorithms. We use (a mild extension of) the batch algorithms

of Reference [33] to compute $\mathrm{Vio}(\Sigma, G)$ once and then incrementally compute changes $\Delta\mathrm{Vio}$ in response to ΔG. The rationale behind this is that in the real world, changes are typically small, e.g., less than 5% on the entire Web in a week [58]. When ΔG is small, $\Delta\mathrm{Vio}$ is often small as well and is much less costly to compute than $\mathrm{Vio}(\Sigma, G \oplus \Delta G)$ by making use of previous computation for $\mathrm{Vio}(\Sigma, G)$.

(c) While desirable, the incremental detection problem is nontrivial. We show that the problem is also coNP-complete, even when both graphs G and updates ΔG have *constant sizes* (Section 5).

(d) We also establish the parameterized complexity of the validation and incremental validation problems. We show that both problems are co-W[2]-hard [5], and they become fixed-parameter tractable (FPT [35]) for NGDs defined with connected graph patterns (Section 5).

(e) In response to the practical need, we develop two algorithms for incremental error detection with NGDs (Section 6), which make incremental error detection feasible in large-scale graphs.

One is a sequential *localizable* algorithm IncDect. It incrementalizes subgraph search by *update-driven evaluation*. Its cost is determined by the d_Σ-neighbors of nodes in ΔG, where d_Σ is the maximum diameter of the patterns in Σ [28]. In practice, Σ is much smaller than G and so is d_Σ. It reduces the computations on (possibly big) graphs G to smaller d_Σ-neighbors of those nodes in ΔG.

The other one is a parallel algorithm PIncDect. We show that it is *parallel scalable* relative to IncDect: Its cost is $O(t(|G|, |\Sigma|, |\Delta G|)/p)$, where p is the number of processors used, and $t(|G|, |\Sigma|, |\Delta G|)$ is the cost of IncDect. That is, PIncDect guarantees to reduce running time when more processors are used. We propose a *hybrid strategy* to split skewed work units and dynamically balance workload, based on cost estimation, to balance computation and communication.

(f) To strike a balance between the efficiency and the number of errors detected, we also develop parallel polynomial-time (PTIME) algorithms for detection and incremental detection of *top-ranked* inconsistencies with NGDs of a special form when the ranking function is adopted to measure the importance of the errors (Section 7). We show that both algorithms are also parallel scalable.

(4) Experimental study. Using real-life and synthetic graphs, we empirically evaluate the scalability and efficiency of our algorithms (Section 8). We find the following: (a) Incremental error detection with NGDs is effective: Sequential algorithm IncDect is on average 6.7× faster than its batch counterpart when $|\Delta G|$ accounts for 10% of $|G|$, and still does better even when $|\Delta G|$ is up to 33% of $|G|$. (b) The incremental algorithms scale well with $|G|$. (c) Algorithm PIncDect is parallel scalable and efficient: It is on average 3.7× faster when the number p of processors increases from 4 to 20. It takes 225 s on graphs of 28M nodes and 33.4M edges when $p = 20$. (d) Hybrid workload balancing improves the performance by 1.73× on average. (e) The parallel PTIME algorithm for detection (respectively, incremental detection) of top-ranked errors is efficient, which takes at most 103 s (respectively, 35 s) on graphs of size (30M, 60M) using eight processors (when $|\Delta G| = 15\%|G|$).

The novelty of the work consists of (1) NGDs, a class of graph dependencies to capture semantic inconsistencies in real-life graphs, numeric or not, balancing the expressive power and complexity; (2) fundamental results for reasoning about NGDs, demonstrating the complications introduced by arithmetic expressions and comparison predicates; and (3) the first incremental (numeric) error detection algorithms for graphs, parallel and sequential, with performance guarantees.

Related work. This article extends its conference version [30] by including (1) detailed proofs for all the results (Theorems 4.2, 4.4, 5.3, and 6.3, Corollaries 4.3 and 5.1); these proofs are highly nontrivial; they are interesting in their own right and illustrate the subtleties involved in the interactions among arithmetic expressions, comparison predicates, and GFDs; (2) parameterized

complexity of the validation and incremental validation problems for NGDs to reveal where the complexity arises; we show that both problems are co-W[2]-hard but become fixed-parameter tractable for NGDs with connected patterns (Theorems 5.2 and 5.4); (3) parallel PTIME (incremental) detection algorithms for top-ranked inconsistencies with a form of NGDs (Section 7); and (4) new experimental results to show the performance of our algorithms for detecting top-ranked inconsistencies (Section 8).

We categorize the other related work as follows:

Dependencies for graphs. Dependencies have been studied for RDF [9, 21, 24, 43, 52, 73] and for generic graphs [31, 33]. This line of work started from Reference [52]. It extends RDF vocabulary to define keys, foreign keys, and functional dependencies (FDs). Using triple patterns with variables, References [9, 21] interpret FDs with triple embedding and homomorphism. A class of FDs was formulated in Reference [73] with path patterns; these FDs were extended in Reference [43] to support CFDs. References [17, 38] study a class of first-order Horn clause with binary predicates as soft constraints to facilitate knowledge base reasoning.

Closer to this work are GFDs on general graphs [33], defined in terms of (a) a pattern Q interpreted via subgraph isomorphism and (b) an extension of an FD carrying constant and variable literals. GFDs are extended to graph entity dependencies (GEDs) in Reference [31] by supporting literals with node identities to express (recursively defined) keys of Reference [24], interpreted via graph homomorphism.

This work defines NGDs by extending GFDs and interprets pattern matching by graph homomorphism following Reference [31]. It differs from References [31, 33]: (1) NGDs support both arithmetic operations and comparison predicates, extending GFDs and GEDs. (2) As shown by the fundamental results (Sections 4 and 5), the presence of either arithmetic operations or built-in predicates makes satisfiability and implication problems Σ_2^p-complete and Π_2^p-complete, respectively, as opposed to coNP-complete and NP-complete for GFDs and GEDs. (3) We establish the parameterized complexity of the detection and incremental detection problems for NGDs and identify FPT practical special cases. (3) We develop the first (parallel) incremental error detection algorithms for graphs with performance guarantees, which complement the batch detection algorithms for GFDs [33]. We also provide the first PTIME parallel algorithms for (incremental) detection of top-ranked errors.

Dependencies on numeric data. Several dependency classes have been studied for detecting numeric errors in relations [23, 34, 37, 40, 49, 63]. Metric functional dependencies [49] and sequential dependencies [40] extend FDs by supporting (numeric) metrics and intervals on ordered data, respectively. Differential dependencies [63] constrain distances of numeric attribute values among different tuples. However, none of these supports arithmetic operations. There has also been work on repairing numeric data using constraints defined in terms of aggregate functions [34] and disjunctive logic programming [37]. Their satisfiability and implication problems are open, and the complexity is suspected high. Numeric functional dependencies (NFDs) [23] extend CFDs and support linear arithmetic expressions and built-in predicates like NGDs.

This work differs from the prior work as follows: (1) NGDs are defined on schemaless graphs with a graph pattern and an attribute dependency. They cannot be expressed as dependencies of References [23, 40, 49, 63]. As shown in Reference [31], GFDs, a special case of NGDs, are not expressible even as equality-generating dependencies with constants, which subsume CFDs. As an evidence, the validation problem is coNP-complete for NGDs and GFDs, but is in polynomial time (PTIME) for CFDs [26] and NFDs [23]. (2) The techniques for handling graph dependencies are quite different from those for relational counterparts. For instance, we make use of the data locality of graph homomorphism to check NGDs (Section 6), a departure from relational

dependencies. (3) To strike a balance between the complexity and expressivity, we do not consider aggregations; in fact, most numeric errors we encounter in real-life graphs can be caught without using aggregations.

Comparison predicates have been included in dependencies for detecting inconsistencies [23], data exchange [8], and views for query rewriting [6, 7]. However, (1) the comparisons in References [6–8] are posed over dense orders, whereas we study linear arithmetic constraints over integers, whose satisfiability problem is NP-complete [59], as opposed to PTIME for densely ordered domains. (2) Chasing with NGDs, e.g., testing satisfiability, always terminates [31], but the chase in Reference [8] may not. (3) References [6–8] do not consider any arithmetic operators supported by NGDs.

Algorithms for error detection. Error detection has been studied for relations [29, 60, 68] and RDF [48, 66, 69]. Reference [29] studies (incremental) CFD validation in horizontally or vertically partitioned relations. A continuous framework is developed in Reference [68] to clean relations that may change, using FDs that may also evolve. Reference [60] combines logical and quantitative data cleaning by using metric FDs. Consistency checking in RDF is conducted by logical reasoning [66] or by unsupervised detection of numerical outliers [69]. Reference [48] detects errors in RDF with SPARQL queries. On general graphs, Reference [64] studies repairing of vertex labels with a form of neighborhood constraints. Batch algorithms are developed for catching violations of GFDs [33] or recursively defined keys [24] in graphs, in parallel.

Our algorithms differ from previous ones in the following: (1) We provide the first incremental error detection algorithms that are localizable [28] and relatively parallel scalable. As far as we know, none of the previous error detection algorithms is shown parallel scalable except the batch ones of References [24, 33]. However, the parallel algorithms in References [24, 33] cannot be directly incrementalized. Localizable algorithms have only been developed for graph queries [28], e.g., keyword search. (2) We propose update-driven search and a hybrid dynamic strategy to achieve relative parallel scalability. The strategy balances the workload at runtime at two levels: (a) it makes use of cost estimation to split and distribute stragglers and (b) it monitors the status of processors and reassigns work units from a busy processor to those with a light load. While (b) is along the same lines as work stealing and shedding [14, 42], we find that it does not work well alone unless in combination with (a). (3) We study incremental detection of top-ranked semantic errors, which has not been considered in the previous work, in particular prior work on top-k graph search, e.g., References [18, 71, 72].

Incremental detection of the violations of NGDs over fragmented and distributed graphs is more intriguing than conventional graph pattern matching: We have to compute violations that are either newly introduced or removed by updates only. As a consequence, previous algorithms for parallel pattern matching, e.g., References [44, 51, 67], cannot be applied directly in this context.

2 PRELIMINARIES

We first review basic notations. Assume alphabets Γ, Θ, and U denoting labels, attributes, and constant values, respectively.

Graphs. We consider directed *graphs* $G = (V, E, L, F_A)$ with labeled nodes and edges, and attributes on its nodes. Here (1) V is a finite set of nodes; (2) $E \subseteq V \times V$ is a set of edges, in which (v, v') denotes an edge from node v to v'; (3) each node v in V (respectively, edge e in E) carries label $L(v)$ (respectively, $L(e)$) in Γ; and (4) for each node v, $F_A(v)$ is a tuple $(A_1 = a_1, \ldots, A_n = a_n)$ such that $A_i \neq A_j$ if $i \neq j$, where a_i is a constant in U and A_i is an *attribute* of v drawn from Θ, written as $v.A_i = a_i$, carrying the content of v such as keywords and blogs as found in social networks.

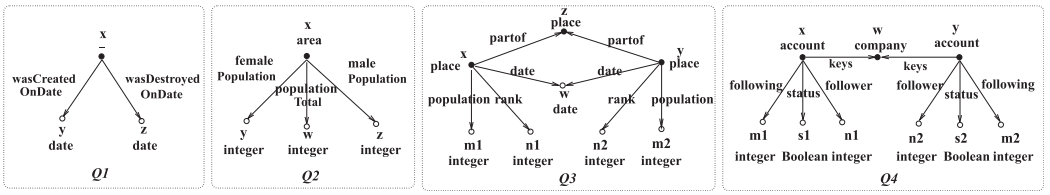


Fig. 2. Graph patterns.

We use two notions of subgraphs given as follows:

- A graph $G' = (V', E', L', F'_A)$ is called a *subgraph of* $G = (V, E, L, F_A)$, denoted by $G' \subseteq G$, if $V' \subseteq V$, $E' \subseteq E$, and for each node $v \in V'$, $L'(v) = L(v)$ and $F'_A(v) = F_A(v)$; similarly for each edge $e \in E'$, $L'(e) = L(e)$.
- A subgraph G' is *induced by* a set V' of nodes if $V' \subseteq V$ and E' consists of all the edges in E whose endpoints are both in V'.

Graph patterns. A *graph pattern* is a directed graph $Q[\bar{x}] = (V_Q, E_Q, L_Q, \mu)$, where (1) V_Q (respectively, E_Q) is a set of pattern nodes (respectively, edges); (2) L_Q is a function that assigns a label $L_Q(u)$ (respectively, $L_Q(e)$) in Γ to each pattern node $u \in V_Q$ (respectively, edge $e \in E_Q$); (3) $\bar{x}$ is a list of distinct variables; and (4) μ is a bijective mapping from $\bar{x}$ to V_Q, i.e., it assigns a distinct variable to each node v in V_Q.

For a variable $x \in \bar{x}$, we use $\mu(x)$ and x interchangeably when it is clear in the context. We allow wildcard "_" as a special label in pattern $Q[\bar{x}]$, where _ is not a label in Γ.

Remark. (1) We consider directed graphs and patterns in this article. Nonetheless, the techniques can be readily adapted to undirected graphs and patterns. In fact, an undirected pattern can be easily represented as a directed pattern by encoding each undirected edge with two directed edges. (2) We allow generic graph patterns, which can be either cyclic or acyclic.

Graph pattern matching. We adopt the homomorphism semantics of pattern matching following References [9, 21, 31]. A *match* of pattern $Q[\bar{x}]$ in graph G is a mapping h from Q to G such that (a) for each node $u \in V_Q$, $L_Q(u) = L(h(u))$; and (b) for each $e = (u, u')$ in Q, $e' = (h(u), h(u'))$ is an edge in G and $L_Q(e) = L(e')$. In particular, $L_Q(u) = L(h(u))$ always holds if $L_Q(u)$ is "_", i.e., wildcard matches any label from Γ to indicate generic entities; similarly for wildcard and edge labels.

We denote the match as a vector $h(\bar{x})$, consisting of $h(x)$ for all $x \in \bar{x}$, in the same order as $\bar{x}$. Intuitively, $\bar{x}$ is a list of entities to be identified by Q, and $h(\bar{x})$ is such an instantiation in G.

Example 2.1. Four graph patterns are shown in Figure 2. Here Q_1 depicts an entity x connected to date y and z with edges labeled wasCreatedOnDate and wasDestroyedOnDate, respectively. Node x is labeled "_", denoting arbitrary entities regardless of their labels. In G_1 of Figure 1, x is mapped to BBC_Trust. Similarly, Q_2–Q_4 can be interpreted by referencing their counterparts in Figure 1.

3 NUMERIC GRAPH DEPENDENCIES

We extend GFDs [31, 33] to support arithmetic and built-in predicates. We start with basic notations.

Literals. Consider a graph pattern $Q[\bar{x}]$. A *term* of $Q[\bar{x}]$ is either an integer c in U or an integer "variable" $x.A$, where $x \in \bar{x}$ and A is an attribute in Θ (note that attributes are not specified in Q).

A *linear arithmetic expression* e *of* $Q[\bar{x}]$ is defined as follows:

$$e ::= t \mid |e| \mid e + e \mid e - e \mid c \times e \mid e \div c,$$

where t is a term, c is an integer, and $|e|$ is the absolute value of e. The *degree* of expression e is at most 1, representing the maximum sum of the exponents of variables in its monomials (e.g., $x.A$).

For instance, all the arithmetic expressions given in Example 1.1 are linear. As will be seen in Section 4, we adopt linear e to strike a balance between the expressive power and complexity.

A *literal* l *of* $Q[\bar{x}]$ is of the form $e_1 \otimes e_2$, where e_1 and e_2 are linear arithmetic expressions of $Q[\bar{x}]$, and $\otimes$ is one of the built-in comparison operators $=, \neq, <, \leq, >$, and $\geq$.

NGDs. A *numeric graph dependency*, denoted by NGD, is of the form $Q[\bar{x}](X \rightarrow Y)$, where

- $Q[\bar{x}]$ is a graph pattern called the *pattern* of φ; and
- X and Y are (possibly empty) sets of literals of $Q[\bar{x}]$.

Intuitively, an NGD φ is a combination of two constraints: (a) a *topological constraint* imposed by graph pattern Q, to identify entities in a graph, and (b) an *attribute dependency* $X \rightarrow Y$, defined with linear arithmetic expressions connected with built-in predicates, to be enforced on the entities identified by Q. To simplify the discussion, we also write X and Y as conjunctions of literals.

Numeric graph dependencies extend GFDs of References [31, 33] by supporting

(a) linear arithmetic expressions with $+, -, \times, \div$, and $|\cdot|$, and
(b) comparisons with built-in predicates $=, \neq, <, \leq, >, \geq$.

In other words, GFDs of References [31, 33] are a special case of NGDs when literals are restricted to terms connected with equality "=" only, i.e., literals of the form $x.A = c$ or $x.A = y.B$.

Here variable $x.A$ can carry any values of an atomic type τ, where τ is either numeric (integer) or non-numeric (e.g., string, timestamp, enumerated types). Arithmetic operations are defined on numeric values as usual. Built-in predicates $=, \neq, <, \leq, >$, and $\geq$ are defined on A-attribute values as long as these predicates are defined in the domain of its type τ, e.g., integer and string.

Example 3.1. To catch those errors spotted in Example 1.1, we define the following NGDs, in terms of the patterns depicted in Figure 2, with arithmetic expressions and built-in predicates.

(1) Yago. NGD $\varphi_1 = Q_1[x, y, z](\emptyset \rightarrow z.\text{val} - y.\text{val} \geq c)$. Here X is empty set $\emptyset$ and Y includes a single literal. From Q_1 of Figure 2, we can see that x, y, and z denote an entity, the date when it was created, and the date when it was destroyed, respectively; val is an attribute for the integer values of y and z in days (not shown in Q_1); and c is a non-negative constant integer. It states that an entity cannot be destroyed within c days of its creation. It catches the error in graph G_1 of Figure 1.

(2) Yago. NGD $\varphi_2 = Q_2[w, x, y, z](\emptyset \rightarrow y.\text{val} + z.\text{val} = w.\text{val})$. The NGD says that, in any area x, its total population w.val should equal the sum of its female population y.val and its male population z.val. It catches the inconsistency in graph G_2 of Figure 1.

(3) DBPedia. NGD $\varphi_3 = Q_3[\bar{x}](m_1.\text{val} < m_2.\text{val} \rightarrow n_1.\text{val} > n_2.\text{val})$, where $\bar{x}$ includes two places x and y in the same area z. It states that if the population m_1.val of x is less than the population m_2.val of y in the same census w, then the populationRank n_1.val of x is behind the populationRank n_2.val of y. It captures the inconsistency in graph G_3 of Figure 1.

(4) Twitter. NGD $\varphi_4 = Q_4[\bar{x}]((s_1.\text{val} = 1) \wedge (a \times (m_1.\text{val} - m_2.\text{val}) + b \times (n_1.\text{val} - n_2.\text{val}) > c) \rightarrow s_2.\text{val} = 0)$. Here $\bar{x}$ includes two accounts x and y about the same company w, where x (respectively, y) has n_1.val (respectively, n_2.val) followers and is following m_1.val (respectively, m_2.val) accounts, and has status s_1.val (respectively, s_2.val) indicating whether x (respectively, y) is real. Integers a and b specify the weights of following and followers, respectively; and c is the threshold for their difference (see Example 1.1). It states that if the gap between the followers and following of a real

account x and account y exceeds c, then the chances are that y is fake. It catches NatWest_Help in G_4 of Figure 1 as a fake account.

Remark. NGDs and GFDs depart from their relational counterparts as follows:

(1) Consider a functional dependency (FD) $R(X \rightarrow Y)$ defined on a relation schema R with attributes X and Y. Here R specifies the "scope" of the FD, i.e., $X \rightarrow Y$ is to be applied to an instance D of R such that for any tuples t_1 and t_2 in D, if $t_1[X] = t_2[X]$, then $t_1[Y] = t_2[Y]$. In contrast, graphs are semistructured and are often schemaless. To define a GFD (i.e., an FD on graphs G), we can no longer find a set of tuples as its scope. Instead, we define a GFD $Q[\bar{x}](X \rightarrow Y)$ with (a) a pattern Q to identify associated entities in G as the "scope," i.e., it identifies entities to which the "FD" is applied; and (b) a dependency $X \rightarrow Y$ on the attributes of the entities identified by Q.

(2) As shown in Reference [31], GFDs can express (a) conditional functional dependencies (CFDs [26]), (b) equality generating dependencies (EGDs [4]), and (c) a form of tuple generating dependencies (TGDs [4]) for the existence of attributes, when relational tuples are represented as vertices in a graph. For example, let $R(\mathsf{CC}, \mathsf{ZIP}, \mathsf{STR})$ be a relation schema; then $(\mathsf{CC} = 44, \mathsf{ZIP}) \rightarrow \mathsf{STR}$ is a CFD over R, which states that in the UK, zip code (ZIP) uniquely determines one street (STR). This CFD can be expressed as a GFD: $Q[x, y](x.\mathsf{CC} = y.\mathsf{CC} \wedge x.\mathsf{CC} = 44 \wedge x.\mathsf{ZIP} = y.\mathsf{ZIP} \rightarrow x.\mathsf{STR} = y.\mathsf{STR})$, where Q consists of *two nodes x and y* labeled with R, each denoting a tuple of R.

NGDs subsume GFDs as a special case, and hence can also express CFDs, EGDs, and limited TGDs. In particular, NGDs support constant bindings of CFDs [26], which have proven useful in detecting errors in relations [25]. Hence, NGDs can catch non-numeric inconsistencies that GFDs and CFDs can detect, in addition to numeric errors. Moreover, NGDs support conditions defined in terms of arithmetic operations and built-in predicates, which are beyond CFDs, EGDs, and TGDs. Note that NGD φ_4 of Example 3.1 cannot be expressed by numeric functional dependencies (NFDs) [23], since NFDs do not support preconditions with arithmetic expressions.

(3) Association rules have recently been studied for graphs, which often have existential semantics to deduce the existence of certain edges, e.g., graph-pattern association rules (GPARs) proposed in Reference [32]. In contrast, GFDs and NGDs are universal logic sentences. While they can enforce the existence of attributes (i.e., limited TGDs), they cannot deduce the existence of edges. Hence, GFDs and NGDs are not able to express graph association rules such as GPARs.

Semantics. Consider a match $h(\bar{x})$ of Q in a graph G.

We say that match $h(\bar{x})$ *satisfies* a literal $l = e_1 \otimes e_2$ of $Q[\bar{x}]$ if (a) for each variable $x.A$ in l, node $v = h(x)$ carries attribute A, and (b) $h(e_1) \otimes h(e_2)$, where $h(e_i)$ denotes the arithmetic expression obtained from e_i by substituting $h(x)$ for each x in e_i for $i \in [1, 2]$; here $h(e_1) \otimes h(e_2)$ is interpreted following the standard semantics of arithmetic operations and build-in predicates.

For instance, for $e_1 > e_2$, where e_1 is $x.A + x.B$ and e_2 is 3, $h(x)$ satisfies $e_1 > e_2$ if (a) node $v = h(x)$ carries attributes A and B, and (b) the value of $v.A + v.B$ is greater than 3.

For a set Z of literals, we write $h(\bar{x}) \models Z$ if $h(\bar{x})$ satisfies *all* literals in Z, i.e., their conjunction. We write $h(\bar{x}) \models X \rightarrow Y$ if $h(\bar{x}) \models X$ implies $h(\bar{x}) \models Y$, i.e., if $h(\bar{x}) \models X$, then $h(\bar{x}) \models Y$.

A graph G *satisfies* NGD$\varphi = Q[\bar{x}](X \rightarrow Y)$, denoted by $G \models \varphi$, if *for all* matches $h(\bar{x})$ of Q in G, $h(\bar{x}) \models X \rightarrow Y$. Graph G *satisfies* a set Σ of NGDs, denoted by $G \models \Sigma$, if for all NGDs $\varphi \in \Sigma$, $G \models \varphi$.

Intuitively, to check whether $G \models \varphi$, we need to examine all matches $h(\bar{x})$ of Q in G. We check whether $h(\bar{x}) \models Y$ only if $h(\bar{x})$ is a match of Q and it satisfies the precondition X.

Example 3.2. Consider graph G_1 depicted in Figure 1 and NGD φ_1 given in Example 3.1. Then $G_1 \not\models \varphi_1$, since there exists a match $h(x, y, z)$: $x \mapsto$ BBC_Trust, $y \mapsto$ 2007-#-# and $z \mapsto$ 1946-08-28,

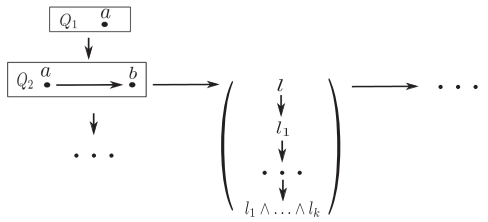


Fig. 3. GFD generation.

such that $h(y).\text{val} > h(z).\text{val}$, i.e., $h(x, y, z) \not\models Y$. That is, $h(x, y, z)$ denotes the list of entities that make a *violation* of φ_1 in G_1. Similarly, $G_2 \not\models \varphi_2$, $G_3 \not\models \varphi_3$ and $G_4 \not\models \varphi_4$.

NGDs generation. How can we find NGDs? Below, we address this question.

NGD *design*. Traditionally, data quality rules are typically designed by domain experts. To design an NGD $Q[\bar{x}](X \rightarrow Y)$, one needs to specify the following: (1) pattern Q, which identifies the "scope" of $X \rightarrow Y$; (2) linear arithmetic expressions on the attributes of the nodes in Q; and (3) literals defined with the arithmetic expressions and built-in predicates. Using domain knowledge, experts can figure out (2) and (3) on attributes of their interest. However, it is nontrivial to design a suitable pattern Q by inspecting all nodes carrying attributes of their interest in a large real-life graph.

To this end, one can adopt query reverse engineering [46] to generate Q. More specifically, one may (a) first select some nodes of interest and (b) then by treating these nodes as output, use the query reverse engineering techniques to compute "input queries," which are patterns Q that can be mapped to the selected nodes. Domain experts may then inspect the patterns generated.

NGD *discovery*. To help non-expert users find useful NGDs, we extend the GFD discovery algorithm of Reference [27] to discover NGDs from graphs. Intuitively, the algorithm of Reference [27] interleaves "vertical levelwise expansion" for mining frequent patterns Q and "horizontal levelwise expansion" for mining literals in $X \rightarrow Y$ in a single process. As depicted in Figure 3, the algorithm "cold starts" with single-node patterns Q_1. It then iterates the following steps: (1) vertically expand a pattern Q_i by adding one edge to generate a new pattern Q_{i+1}; and (2) horizontally expand literals $l_1, \ldots, l_k$ for each literal l of Q_{i+1} by adding one literal each time, such that $Q_{i+1}[\bar{x}](l_1 \wedge \ldots \wedge l_k \rightarrow l)$ makes a candidate GFD. The process proceeds until the GFDs generated exceed a predefined size. Optimization techniques are developed in Reference [27] to prune unnecessary search.

Compared with GFD discovery, NGD discovery is more challenging, since the literals of NGDs carry linear arithmetic expressions and cannot be retrieved directly from graphs. Nonetheless, such literals can be identified by adopting feature clustering [19] and linear regression [56]. More specifically, for a pattern Q, we (1) extract attributes $A_1, \ldots, A_n$ pertaining to Q, (2) use feature clustering to group these attributes into, e.g., $\{A_{1,1}, \ldots, A_{1,n_1}\}, \ldots, \{A_{k,1}, \ldots, A_{k,n_k}\}$, and (3) for each group $\{A_{j,1}, \ldots, A_{j,n_j}\}$ ($j \in [1, k]$), train a linear regression model using the data corresponding to these attributes in the matches of Q and treat this linear regression model as a literal.

4 FUNDAMENTAL PROBLEMS FOR NGDS

We next study two fundamental problems associated with NGDs, namely, the satisfiability and implication problems. The main result of this section is that the presence of either linear arithmetic expressions or built-in predicates makes these problems harder unless P = NP. This said, we show that NGDs strike a balance between the complexity and expressive power.

4.1 The Analysis of NGDs

We first state the two fundamental problems and establish their complexity bounds for NGDs.

(1) Satisfiability. We consider two notions of satisfiability.

A set Σ of NGDs is *satisfiable* if there exists a graph G such that (a) $G \models \Sigma$ and (b) there exists a NGD $Q[\bar{x}](X \to Y)$ in Σ such that pattern Q has a match in G. Intuitively, condition (b) is to ensure that the NGDs can be applied to nonempty graphs.

We say that Σ is *strongly satisfiable* if there exists a graph G such that (a) $G \models \Sigma$ and (b) for each NGD $Q[\bar{x}](X \to Y)$ in Σ, there exists a match of Q in G. Intuitively, condition (b) requires that all patterns in Σ find a match in G to ensure that the NGDs in Σ do not conflict with each other.

The *satisfiability problem* for NGDs is to decide, given a set Σ of NGDs, whether Σ is satisfiable. The *strong satisfiability problem* is to decide whether Σ is strongly satisfiable.

Example 4.1. Consider a set Σ_0 consisting of two NGDs: $\varphi_5 = Q[x](\emptyset \to x.A = 7 \wedge x.B = 7)$ and $\varphi_6 = Q[x](\emptyset \to x.A + x.B = 11)$, where pattern Q has a single node x labeled "_". One can verify that there exist nonempty graphs that satisfy φ_5 and φ_6 when taken separately. However, φ_5 and φ_6 are not satisfiable when put together. Indeed, the values of attributes A and B on each node must be 7 as required by φ_5, while their sum is required to be 11 by φ_6, which is impossible.

Suppose that pattern Q in φ_6 is replaced by Q' that has a single node labeled "a." Then Σ_0 becomes satisfiable. Indeed, consider graph G having a single node v labeled "b" with $v.A = v.B = 7$. Then $G \models \Sigma_0$. But Σ_0 is not strongly satisfiable, since for any graph G', if all patterns in Σ_0 find a match in G', then there must exist nodes labeled ''a," and the conflicts above arise again.

Similarly, one can verify that the NGDs below are not (strongly) satisfiable: $\varphi_7 = Q[x](x.A \leq 3 \to x.B > 6)$, $\varphi_8 = Q[x](x.A > 3 \to x.B > 6)$, and $\varphi_9 = Q[x](\emptyset \to x.B < 6 \wedge x.A \neq 0)$.

These show that the presence of either linear arithmetic expressions or built-in comparison predicates beyond equality makes the satisfiability analysis more intriguing than that of GFDs [31, 33].

(2) Implication. A set Σ of NGDs *implies* another NGD φ, denoted by $\Sigma \models \varphi$, if for all graphs G, if $G \models \Sigma$, then $G \models \varphi$. That is, the NGD φ is a logical consequence of the set Σ of NGDs.

The *implication problem* for NGDs is to determine whether $\Sigma \models \varphi$ for given NGDs Σ and φ.

As remarked in Section 1, the practical need for studying these problems is evident, besides theoretical interest, for determining whether data quality rules discovered from possibly dirty data are sensible, and for optimizing data quality rules, among other things.

Complexity. We next settle the complexity of these problems.

Recall that the satisfiability problem for relational functional dependencies (FDs) is trivial, i.e., for any set Σ of FDs over a relation schema R, there always exists a nonempty instance of R that satisfies Σ [25]. Moreover, the implication problem for FDs is in linear-time (cf. Reference [4]). It is known that the satisfiability and implication problems for GFDs are coNP-complete and NP-complete [33], respectively. These are comparable to their counterparts for relational CFDs, which are NP-complete and coNP-complete, respectively [26]. In contrast, NGDs make our lives harder.

Theorem 4.2. *For* NGDs*, (a) the satisfiability problem and strong satisfiability problems are both* Σ_2^p*-complete and (b) the implication problem is* Π_2^p*-complete.*

Here Σ_2^p is the class of decision problems that are solvable in NP by calling an NP oracle, i.e., $\Sigma_2^p = \mathsf{NP}^{\mathsf{NP}}$. It is considered more intriguing than NP unless P = NP. Similarly, $\Pi_2^p = \mathsf{coNP}^{\mathsf{NP}}$, which is also above NP in the polynomial hierarchy (see Reference [59] for details).

Proof. We analyze the fundamental problems for NGDs one-by-one.

The satisfiability problem for NGDs. We show that the satisfiability problem for NGDs is Σ_2^p-complete. The proof is a little involved. We first show that the satisfiability problem for NGDs has a small model property, based on which, we give an Σ_2^p algorithm to check whether a given set Σ of NGDs is satisfiable. We then prove that the problem is Σ_2^p-hard.

The small model property. We show that if a set Σ of NGDs is satisfiable, then Σ has a model G_Σ such that $G_\Sigma \models \Sigma$, $|G_\Sigma| \leq 3(|\Sigma|+1)^5$ and Q has a match in G_Σ for some $\varphi = Q[\bar{x}](X \to Y)$ in Σ.

By the definition of satisfiability, for any satisfiable set Σ, there exists a graph G such that $G \models \Sigma$ and Q has a match h_φ in G for some NGD $\varphi = Q[\bar{x}](X \to Y)$ in Σ. Based on the match h_φ, we construct graph G_Σ in two steps. (a) We first deduce a subgraph G_φ of G by using the topological structure derived from h_φ. (b) We then revise the attribute values in G_φ to get a small model G_Σ of Σ.

(a) We deduce G_φ as the subgraph of G "induced" by match h_φ, which includes those nodes and edges that are mapped from Q. That is, $G_\varphi = (V_\varphi, E_\varphi, L_\varphi, F_A^\varphi)$, where (1) $V_\varphi = \{h_\varphi(x) \mid x \in \bar{x}\}$, where $\bar{x}$ refers to the list of distinct variables in NGD $\varphi = Q[\bar{x}](X \to Y)$; (2) $E_\varphi = \{(h_\varphi(x_1), h_\varphi(x_2)) \mid (x_1, x_2) \in E_Q\}$, where E_Q is the set of edges in pattern $Q[\bar{x}]$; (3) L_φ is such defined that $L_\varphi(v) = L(v)$ for $v \in V_\varphi$, and $L_\varphi(e) = L(e)$ for $e \in E_\varphi$; and (4) we define F_A^φ by taking attributes that only appear in Σ; more specifically, for each NGD $\varphi' = Q'[\bar{x'}](X' \to Y')$ in Σ, match $h_{\varphi'}$ of Q' in G_φ, and integer variable $x'.A$ that appears in X', $F_A^\varphi(v').A = F_A(v').A$, where $v' = h_{\varphi'}(x')$; moreover, if $h_{\varphi'}(\bar{x'}) \models X'$, then $F_A^\varphi(h_{\varphi'}(y')).A = F_A(h_{\varphi'}(y')).A$ for each integer variable $y'.A$ that appears in Y'.

This is well-defined, since $G \models \Sigma$. From the construction, we have that $G_\varphi \models \Sigma$, $|V_\varphi| \leq |\Sigma|$, and each node in G_φ has at most $|\Sigma|$ attributes. Note that the labels and attribute values in G_φ may be of size exponential in $|\Sigma|$ since they are copied directly from G.

(b) We revise G_φ to obtain G_Σ, in which we revise the labels and values of attributes in G_φ to eliminate those "unbounded" ones. We first replace all labels in G_φ that are not in Σ with a single label l_Σ that does not occur in Σ such that $|l_\Sigma| \leq |\Sigma|$. One can verify that for any pattern Q in Σ, h is a match of Q in G_φ if and only if h is a match of Q in G_Σ, since the substituted label only matches the wildcard "_" in h. That is, all the matches of patterns from Σ in G_φ remain unchanged after the label replacement by the definition of graph pattern matching. Hence, the graph still satisfies Σ.

It remains to revise the attribute values. The main challenge is to ensure that $G_\Sigma \models \Sigma$ after the values are changed. We "normalize" the attributes by solving an integer linear programming problem $\mathsf{L}_\Sigma : D\bar{y} \leq \bar{b}$ constructed from graph G_φ, where D is an integral $m \times n$ coefficient matrix and $\bar{b}$ an integral m-component vector. Denote by $A_1, \ldots, A_n$ the attributes that appear in G_φ. We show that the size of graph G_Σ derived from G_φ by replacing the value of each A_i with a corresponding c_i for $i \in [1, n]$ is at most $3(|\Sigma|+1)^5$ and $G_\Sigma \models \Sigma$, where $(c_1, \ldots, c_n)$ is a feasible solution to L_Σ of length polynomial in $|\Sigma|$. The linear programming instance L_Σ is constructed in three steps: (i) identify the set S of instantiated literals "enforced" on G_φ by Σ; (ii) eliminate the absolute value operator; and (iii) transform the instantiated literals in S into linear inequations of the form $e_i \leq b_i$.

First, a set S of instantiated literals enforced by Σ on G_φ is identified, which includes all instantiated literals that are needed to inspect when checking whether $G_\varphi \models \Sigma$. More specifically, for each NGD $\varphi' = Q'[\bar{x'}](X' \to Y')$ in Σ, match $h_{\varphi'}$ of Q' in G_φ, and literal l in X', we add $h_{\varphi'}(l)$ to S, where $h_{\varphi'}(l)$ refers to the instantiated literal of l by substituting $h_{\varphi'}(x)$ for each variable x in l. The instantiated literal $h_{\varphi'}(l)$ is also included in S for each l in Y' when $h_{\varphi'}(\bar{x'}) \models X$.

Second, to comply with linear programming, we remove absolute value operator from instantiated literals. For each $|e|$ in S, if e is evaluated to be non-negative, then we replace $|e|$ by e; otherwise, $|e|$ is substituted by $-e$. Note that e's value can be readily evaluated after identifying S.

Finally, we transform the instantiated literals in S into the form $e_i \leq b_i$ of inequality constraints. To this end, we aim to eliminate built-in operators except $\leq$, while ensuring all the inequalities hold on G_φ. That is, the corresponding inequation of each instantiated literal $h(l)$ in S depends on whether $h(l)$ is a logical truth in G_φ, denoted by $G_\varphi \models h(l)$. For instance, consider $h(l) = (v.A \leq 3)$. Then it is converted to $-v.A + 1 \leq -3$ if $G_\varphi \not\models h(l)$, i.e., $v.A > 3$ in G_φ, and remains unchanged otherwise. This is consistent with the truth value of the original instantiated literals on it.

Depending on the satisfiability of each instantiated literal $h(l)$ on G_φ, i.e., whether $G_\varphi \models h(l)$, we first transform $h(l)$ into the form of $e'_1 \leq e'_2$ as follows:

$h(l)$	$G_\varphi \models h(l)$	$G_\varphi \not\models h(l)$
$e_1 = e_2$	$e_1 \leq e_2$ and $e_2 \leq e_1$	$e_1 + 1 \leq e_2$ if $e_1 < e_2$, otherwise $e_2 + 1 \leq e_1$
$e_1 \neq e_2$	$e_1 + 1 \leq e_2$ if $e_1 < e_2$, otherwise $e_2 + 1 \leq e_1$	$e_1 \leq e_2$ and $e_2 \leq e_1$
$e_1 < e_2$	$e_1 + 1 \leq e_2$	$e_2 \leq e_1$
$e_1 \leq e_2$	$e_1 \leq e_2$	$e_2 + 1 \leq e_1$
$e_1 > e_2$	$e_2 + 1 \leq e_1$	$e_1 \leq e_2$
$e_1 \geq e_2$	$e_2 \leq e_1$	$e_1 + 1 \leq e_2$

We then transform them into the required form of $e_i \leq b_i$ by using arithmetic transformations to move constant to the right side and remove operator $\div$. This completes the construction of S. The instantiated literals in S are satisfied by G_φ and can be regarded as a linear integer programming problem instance $\mathsf{L}_\Sigma : D\bar{y} \leq \bar{b}$ after the transformation above. Here variables $y_1, \ldots, y_n$ in $\bar{y}$ correspond to the attributes in G_φ, i.e., A_i, for each $i \in [1, n]$.

We construct graph G_Σ from G_φ by using some feasible solution $(c_1, \ldots, c_n)$ to L_Σ of bounded length polynomial in $|\Sigma|$. More specifically, the value of each A_i in G_φ is normalized to c_i in G_Σ, the answer to the corresponding variable y_i of A_i in L_Σ for $i \in [1, n]$.

We now prove the existence of such solutions. It is known that if $D\bar{y} \leq \bar{b}$ has a n-component integer solution, then it has one solution $(c_1, \ldots, c_n)$ with $c_i \leq (n+1)M$ for each $i \in [1, n]$, where M refers to the maximal absolute value for the determinants of the square submatrices of $[D, \bar{b}]$, and $[D, \bar{b}]$ denotes the augmentation matrix of $D\bar{y} \leq \bar{b}$ [20]. One can verify that $M \leq (2^{|\Sigma|}(|\Sigma|^2 + 1))^{|\Sigma|^2+1}$, since the number of variables in L_Σ is at most $|\Sigma|^2$, i.e., $n \leq |\Sigma|^2$, and each value in $[D, \bar{b}]$ is no larger than $2^{|\Sigma|}$. As the attribute values in G_φ constitute a feasible solution to L_Σ by the definition of instance L_Σ, such a solution $(c_1, \ldots, c_n)$ of bounded size always exists, in which each $\|c_i\| \leq \log_2(2^{|\Sigma|^3+|\Sigma|}(|\Sigma|^2+1)^{|\Sigma|^2+2}) \leq 3(|\Sigma|+1)^3$ for $i \in [1, n]$, and $\|c_i\|$ denotes the size of integer c_i.

We next show that G_Σ is a model of bounded size. Suppose by contradiction that $G_\Sigma \not\models \Sigma$. Then there exists a NGD $\varphi' = Q'[\bar{x}'](X' \to Y')$ in Σ and a match $h_{\varphi'}$ of Q' in G_Σ such that $h_{\varphi'}(\bar{x}') \models X'$ and $h_{\varphi'}(\bar{x}') \not\models Y'$. By the definition of G_Σ, $h_{\varphi'}$ is also a match of Q' in G_φ. Suppose that $h_{\varphi'}(\bar{x}') \models X'$ in G_φ. Then $h_{\varphi'}(\bar{x}') \models Y'$ and the instantiated $h_{\varphi'}(l)$ also exists in set S for any literal l in Y', since $G_\varphi \models \Sigma$. As $(c_1, \ldots, c_n)$ is a feasible solution to L_Σ and the satisfiability of original literals from S on G_φ is preserved in the answer to L_Σ, it also satisfies all the literals in Y'. Hence, $h_{\varphi'}(\bar{x}') \models Y'$ in G_Σ, a contradiction. One might think that it is possible that $h_{\varphi'}(\bar{x}') \not\models X'$ in G_φ, since G_Σ and G_φ carry different attribute values. However, a contradiction to that of $h_{\varphi'}(\bar{x}') \models X'$ in G_Σ can be derived analogously, using the argument that the transformation preserves the satisfiability as above.

Finally, to see $|G_\Sigma| \leq 3(|\Sigma|+1)^5$, observe the following: (i) Graph G_Σ has at most $|\Sigma|$ nodes and edges, since they are inherited from their counterparts in G_φ. (ii) There are at most $|\Sigma|$ labels in G_Σ, and unboundedly large ones are replaced by l_Σ with $|l_\Sigma| \leq |\Sigma|$. (iii) The total size of attribute values is at most $3|\Sigma|^2(|\Sigma|+1)^3$. Putting these together, the size $|G_\Sigma|$ of G_Σ is at most $3(|\Sigma|+1)^5$.

Upper bound. We give an Σ_2^p algorithm to check whether a given set Σ of NGDs is satisfiable.

(1) Guess a graph G such that $|G| \leq 3(|\Sigma|+1)^5$, an NGD φ in Σ, and a mapping h_φ from V_Q to V, where V_Q is the set of nodes in the pattern Q of φ.
(2) Check whether h_φ is a match of Q in G; if so, continue; otherwise, reject the current guess.
(3) Check whether $G \models \Sigma$; if so, return true; otherwise, reject the current guess.

The correctness of the algorithm follows from the small model property. For its complexity, step (2) is in PTIME, which follows from the definition of matches; step (3) is in coNP by Theorem 5.1 (to be proved shortly). Therefore, the algorithm is in Σ_2^p and so is the satisfiability problem for NGDs.

Lower bound. We show that the satisfiability problem for NGDs is Σ_2^p-hard by reduction from the generalized subset sum problem, denoted by GSSP, which is Σ_2^p-complete [62]. GSSP is to decide, given an m-component vector $\bar{u}_1$ and a n-component vector $\bar{u}_2$ of integers, an integer w, whether $\exists\bar{v}_1\forall\bar{v}_2(\bar{u}_1^T\cdot\bar{v}_1 + \bar{u}_2^T\cdot\bar{v}_2 \neq w)$. Here $\bar{v}_1$ (respectively, $\bar{v}_2$) is an m (respectively, n)-component vector of Boolean values, i.e., 0 or 1, $\bar{u}_i^T$ is the transpose of $\bar{u}_i$, and $\bar{u}_i^T\cdot\bar{v}_j$ is the inner product of $\bar{u}_i$ and $\bar{v}_j$ $(i,j \in [1,2])$.

Given $\bar{u}_1 = (u_1,\ldots,u_m)$, $\bar{u}_2 = (u'_1,\ldots,u'_n)$ and w, we construct a set Σ of three NGDs such that Σ is satisfiable if and only if $\exists\bar{v}_1\forall\bar{v}_2(\bar{u}_1^T\cdot\bar{v}_1 + \bar{u}_2^T\cdot\bar{v}_2 \neq w)$. To encode the existential semantic of vector $\bar{v}_1$, we use an NGD to ensure that there are m nodes carrying A-attributes with Boolean values. The universal semantic of vector $\bar{v}_2$ is encoded by using wildcards in the pattern to arbitrarily match two nodes with values 0 and 1 of another attribute B. More specifically, Σ is constructed as follows:

(1) The pattern $Q[x_1,\ldots,x_m,y_0,y_1,z_1,\ldots,z_n,z] = (V_Q,E_Q,L_Q,\mu)$ is shared by all NGDs in Σ; it is given as follows: (a) $V_Q = \{v_i \mid i\in[1,m]\}\cup\{v'_0,v'_1,v'_2\}\cup\{v''_i \mid i\in[1,n]\}$; (b) $E_Q = \emptyset$; (c) $L_Q(v_i) = \tau_i$ $(i\in[1,m])$, $L_Q(v'_0) = \gamma_0$, $L_Q(v'_1) = \gamma_1$, $L_Q(v'_2) = \chi$, $L_Q(v''_i) =$ "_" $(i\in[1,n])$; and (d) $\mu(x_i) = v_i$ $(i\in[1,m])$, $\mu(y_0) = v'_0$, $\mu(y_1) = v'_1$, $\mu(z) = v'_2$, and $\mu(z_i) = v''_i$ $(i\in[1,n])$.

That is, Q has $m+n+3$ isolated nodes, in which n nodes are labeled wildcard "_" that can match any label in the data graph, and the other $m+3$ nodes carry distinct labels.

(2) The first NGD of Σ encodes the Boolean values of $\bar{v}_1$ and is defined as $\varphi_1 = Q[x_1,\ldots,x_m,y_0,y_1,z_1,\ldots,z_n,z](\emptyset \to (|2\times x_1.A-1|=1)\wedge\cdots\wedge(|2\times x_m.A-1|=1))$.

Intuitively, it assures that there are m nodes having A-attributes with Boolean values.

(3) The second NGD of Σ is defined as $\varphi_2 = Q[x_1,\ldots,x_m,y_0,y_1,z_1,\ldots,z_n,z](\emptyset \to (y_0.B=0)\wedge(y_1.B=1)\wedge(z.C=1))$, which assures two distinct nodes carrying distinct Boolean values 0 and 1 for their B-attributes, respectively. It also enforces the value of C-attribute to be 1.

(4) The third NGD φ_3 encodes vectors $\bar{u}_1$ and $\bar{u}_2$; it is defined as $\varphi_3 = Q[x_1,\ldots,x_m,y_0,y_1,z_1,\ldots,z_n,z]((|2\times z_1.B-1|=1)\wedge\cdots\wedge(|2\times z_n.B-1|=1)\wedge(A'+B'=w) \to (z.C=2))$, where $A' = x_1.A\times u_1+\cdots+x_m.A\times u_m$, and $B' = z_1.B\times u'_1+\cdots+z_n.B\times u'_n$.

Intuitively, φ_3 is also used for checking the condition of GSSP. To see this, observe that (a) the instantiated $x_1.A,\ldots,x_m.A$ can be assigned to $\bar{v}_1$ when Σ has a model G, since Q must have a match in G; (b) variables $z_1,\ldots,z_n$ can be mapped to nodes labeled γ_1 or γ_2 in an arbitrary way by the semantic of wildcard "_"; moreover, there exist two such nodes having B-attributes with

values 0 and 1, respectively; therefore, the instantiated $z_1.B, \ldots, z_n.B$ for all the matches of Q can be regarded as the set of all n-component vectors of Boolean values that encode the universal semantic of $\bar{v}_2$; and (c) the instantiated $z.C$ is enforced to be 1 by φ_2 and contradicts to $z.C = 2$ when the condition $A' + B' = w$ in φ_3 holds, which encodes the negation of $\exists \bar{v}_1 \forall \bar{v}_2 (\bar{u}_1^T \cdot \bar{v}_1 + \bar{u}_2^T \cdot \bar{v}_2 \neq w)$.

We next prove that Σ is satisfiable if and only if $\exists \bar{v}_1 \forall \bar{v}_2 (\bar{u}_1^T \cdot \bar{v}_1 + \bar{u}_2^T \cdot \bar{v}_2 \neq w)$ holds for $\bar{u}_1$ and $\bar{u}_2$.

($\Rightarrow$) First assume that Σ is satisfiable. Then there exists a graph G such that $G \models \Sigma$, and Q has a match h_Q in G. Based on h_Q, we define Boolean vector $\bar{v}_1 = (h_Q(x_1).A, \ldots, h_Q(x_m).A)$. This is well-defined, since $G \models \varphi_1$, which ensures that each $h_Q(x_i)$ has A-attribute of Boolean value. It remains to prove that $\forall \bar{v}_2 (\bar{u}_1^T \cdot \bar{v}_1 + \bar{u}_2^T \cdot \bar{v}_2 \neq w)$ for this $\bar{v}_1$. Suppose by contradiction that there exists a Boolean vector $\bar{v}_2 = (t'_1, \ldots, t'_n)$ such that $\bar{u}_1^T \cdot \bar{v}_1 + \bar{u}_2^T \cdot \bar{v}_2 = w$. We show that there exists another match h'_Q of Q in G such that $h'_Q(\bar{x}, \bar{y}, \bar{z}) \not\models \Sigma$, a contradiction to $G \models \Sigma$ above. More specifically, h'_Q is such defined that (a) $h'_Q(x_i) = h_Q(x_i)$ for $i \in [1, m]$, (b) $h'_Q(y_0) = h_Q(y_0)$, $h'_Q(y_1) = h_Q(y_1)$, $h'_Q(z) = h_Q(z)$, and (c) $h'_Q(z_i) = h_Q(y_0)$ if t'_i of $\bar{v}_2$ is 0, and otherwise $h'_Q(z_i) = h_Q(y_1)$. One can verify that h'_Q is a match of Q in G, since variable z_i ($i \in [1, n]$) can be mapped to any node by its label of wildcard "_".

We now show that $h'_Q(\bar{x}, \bar{y}, \bar{z}) \not\models \Sigma$ by contradiction. Assume that $h'_Q(\bar{x}, \bar{y}, \bar{z}) \models \Sigma$. One can see that $h'_Q(z).C = 1$ by $G \models \varphi_2$. Moreover, we can verify that $h'_Q(z).C = 2$, a contradiction, because (a) $h'_Q(z_i).B$ (for $i \in [1, n]$), i.e., $h_Q(y_0).B$ or $h_Q(y_1).B$, is a Boolean value that equals t'_i of $\bar{v}_2$, which is guaranteed by φ_2 and the construction of h'_Q, (b) $h'_Q(A' + B') = w$ by the assumption of $\bar{v}_2$, and (c) $G \models \varphi_3$.

($\Leftarrow$) Conversely, assume that $\exists \bar{v}_1 \forall \bar{v}_2 (\bar{u}_1^T \cdot \bar{v}_1 + \bar{u}_2^T \cdot \bar{v}_2 \neq w)$. Based on such a vector $\bar{v}_1 = (s'_1, \ldots, s'_m)$, we construct a model $G = (V, E, L, F_A)$ of Σ, and hence show that Σ is satisfiable, where (a) $V = \{v_1^M, \ldots, v_m^M, v_0^N, v_1^N, v^T\}$; (b) $E = \emptyset$; (c) $L(v_i^M) = \tau_i$ for $i \in [1, m]$, $L(v_0^N) = \gamma_0$, $L(v_1^N) = \gamma_1$, $L(v^T) = \chi$; and (d) $F_A(v_i^M).A = s'_i$ for $i \in [1, m]$, $F_A(v_0^N).B = 0$, $F_A(v_1^N).B = 1$, and $F_A(v^T).C = 1$.

We next show that G is a model of Σ. Observe the following: (a) By the definition of G, it is easy to see that Q has a match h in G and $h(x_i).A = s'_i$ for $i \in [1, m]$. (b) Since for each node v_i^M ($i \in [1, m]$) labeled τ_i in G, $F_A(v_i^M).A$ is a Boolean value, $F_A(v_0^N).B = 0$, $F_A(v_1^N).B = 1$, and $F_A(v^T).C = 1$ for the nodes labeled γ_0, γ_1, and τ, respectively, we have that $G \models \varphi_1$ and $G \models \varphi_2$. It remains to show that $G \models \varphi_3$. Suppose by contradiction that $G \not\models \varphi_3$. Then there exists a match h_Q of Q in G such that $h_Q(\bar{x}, \bar{y}, \bar{z}) \models (|2 \times z_1.B - 1| = 1) \wedge \ldots \wedge (|2 \times z_n.B - 1| = 1) \wedge (A' + B' = w)$. Since $h_Q(\bar{x}, \bar{y}, \bar{z}) \models (A' + B' = w)$, one can verify that there exists an n-component Boolean vector $\bar{v}_2 = (t'_1, \ldots, t'_n)$, where $t'_i = h_Q(z_i).B$ for $i \in [1, n]$, such that $\bar{u}_1^T \cdot \bar{v}_1 + \bar{u}_2^T \cdot \bar{v}_2 = w$, which contradicts to the assumption that $\exists \bar{v}_1 \forall \bar{v}_2 (\bar{u}_1^T \cdot \bar{v}_1 + \bar{u}_2^T \cdot \bar{v}_2 \neq w)$ above. Indeed, since $h_Q(\bar{x}, \bar{y}, \bar{z}) \models |2 \times z_i.B - 1| = 1$, we know that $h_Q(z_i).B$ is a Boolean value and thus can be assigned to t'_i for $i \in [1, n]$, hence a contradiction.

The strong satisfiability problem for NGDs. We show that this problem is also Σ_2^p-complete. Similar to the proof for the satisfiability problem above, we start with a small model property.

The small model property. We show that if a set Σ of NGDs is strongly satisfiable, then there exists a model G_Σ such that $G_\Sigma \models \Sigma$, $|G_\Sigma| \leq 6(|\Sigma| + 1)^5$ and every pattern in Σ has a match in G_Σ.

By the definition of strong satisfiability, if Σ is strongly satisfiable, then there exists a graph $G = (V, E, L, F_A)$ such that $G \models \Sigma$ and for each pattern Q in Σ, there exists a match h_Q of Q in G. Based on these matches, we construct G_Σ as follows: (a) We first deduce a subgraph G' of G such that G' has at most $|\Sigma|$ nodes. (b) We then revise the attribute values and the labels in G' to obtain the model G_Σ of Σ such that $|G_\Sigma| \leq 6(|\Sigma| + 1)^5$, in which each attribute value is of bounded length.

(a) Subgraph G' is "induced" by matches of patterns from Σ, i.e., $G' = (V', E', L', F'_A)$, where

- $V' = \{h_Q(\bar{x}) \mid Q[\bar{x}](X \to Y) \in \Sigma\}$, where h_Q is the match of Q in G;
- $E' = \{(h_Q(v_1), h_Q(v_2)) \mid (v_1, v_2) \in E_Q, Q[\bar{x}](X \to Y) \in \Sigma\}$;
- L' is such defined: for $v \in V'$ (respectively, $e = (v_1, v_2) \in E'$), $L'(v) = L(v)$ (respectively, $L'(e) = L(e)$); and
- we define F'_A by taking attributes that only appear in Σ along the same lines as generating attributes for subgraph G_φ in the proof of the satisfiability problem.

This is well-defined, and we have that $G' \models \Sigma$, $|V'| \leq |\Sigma|$, and each node has at most $|\Sigma|$ attributes.

(b) Similar to the counterpart in the proof for the satisfiability problem above, we can normalize G' to get G_Σ such that $G_\Sigma \models \Sigma$ and $|G_\Sigma| \leq 6(|\Sigma| + 1)^5$. We omit the details here.

Upper bound. Based on the small model property, we give an Σ_2^p algorithm to check whether a given set Σ of NGDs is strongly satisfiable, which works as follows:

(1) Guess a graph $G = (V, E, L, F_A)$ such that $|G| \leq 6(|\Sigma| + 1)^5$, and for each NGD $\varphi = Q[\bar{x}](X \to Y)$ in Σ, guess a mapping h_φ from V_Q to V, where V_Q is the set of nodes in the pattern Q.
(2) Check whether each mapping h_φ is a match of Q in G for NGD $\varphi = Q[\bar{x}](X \to Y)$ in Σ; if so, continue; otherwise, reject the current guess.
(3) Check whether $G \models \Sigma$; if so, return true; otherwise, reject the current guess.

The correctness of the algorithm is assured by the small model property. For its complexity, step (2) is in PTIME for the definition of matches. Step (3) is in coNP by Theorem 5.1, to be given shortly. Therefore, the algorithm is in Σ_2^p and so is the strong satisfiability problem for NGDs.

Lower bound. We prove that the strong satisfiability problem for NGDs is Σ_2^p-hard also by reduction from GSSP. Given $\bar{u}_1 = (u_1, \ldots, u_m)$, $\bar{u}_2 = (u'_1, \ldots, u'_n)$ and w, we construct a set Σ of NGDs such that Σ is strongly satisfiable if and only if $\exists \bar{v}_1 \forall \bar{v}_2 (\bar{u}_1^T \cdot \bar{v}_1 + \bar{u}_2^T \cdot \bar{v}_2 \neq w)$ holds. The set Σ is the same as that in the proof of the satisfiability problem. Since all the NGDs in Σ have the same pattern Q, one can verify that Σ is satisfiable if and only if it is strongly satisfiable. Therefore, the lower bound proof for the satisfiability problem of NGDs coincides with the one for strong satisfiability.

The implication problem for NGDs. We now study the implication problem. Similar to the (strong) satisfiability problem, we first establish a small model property and then use it to prove the upper bound. After these, we show that the implication problem is Π_2^p-hard for NGDs.

The small model property. We prove that given a set Σ of NGDs and an NGD $\varphi = Q[\bar{x}](X \to Y)$, if $\Sigma \not\models \varphi$, then there exists a graph $G_{(\Sigma,\varphi)}$ such that $|G_{(\Sigma,\varphi)}| \leq 6(|\Sigma| + |\varphi| + 1)^5$, $G_{(\Sigma,\varphi)} \models \Sigma$ and $G_{(\Sigma,\varphi)} \not\models \varphi$.

Suppose that $\Sigma \not\models \varphi$. Then there is a graph $G = (V, E, L, F_A)$ such that $G \models \Sigma$, but $G \not\models \varphi$. By $G \not\models \varphi$, there exists a match h of Q in G such that $h(\bar{x}) \models X$, but $h(\bar{x}) \not\models Y$. Based on h, we build $G_{(\Sigma,\varphi)}$ as follows: (1) We first deduce a subgraph G_φ of G from h, such that G_φ has at most $|\varphi|$ nodes. (2) We then normalize the labels and attribute values in G_φ to derive $G_{(\Sigma,\varphi)}$ such that $|G_{(\Sigma,\varphi)}| \leq 6(|\Sigma| + |\varphi| + 1)^5$, i.e., $G_{(\Sigma,\varphi)}$ has a bounded size. Moreover, we show that $G_{(\Sigma,\varphi)} \models \Sigma$ and $G_{(\Sigma,\varphi)} \not\models \varphi$. The construction of $G_{(\Sigma,\varphi)}$ is similar to that given above for the satisfiability problem.

(1) We define G_φ as the subgraph of G "induced" by match h. Let $Q[\bar{x}] = (V_Q, E_Q, L_Q, \mu)$ be the graph pattern of φ. The graph $G_\varphi = (V_\varphi, E_\varphi, L_\varphi, F_A^\varphi)$ is defined as follows:

- $V_\varphi = \{h(v) \mid v \in V_Q\}$, i.e., it includes those nodes mapped from Q via h;
- $E_\varphi = \{(h(v), h(v')) \mid (v, v') \in E_Q\}$, i.e., it also includes those edges mapped from Q;
- the function L_φ is defined as $L_\varphi(v) = L(v)$ for each $v \in V_\varphi$, and $L_\varphi(e) = L(e)$ for each $e \in E_\varphi$;
- F_A^φ is defined in the same way as its counterpart for the satisfiability problem by including attributes that appear in NGDs Σ and φ.

The graph G_φ is well-defined, since $F_A^\varphi(\cdot)$ inherits values from $F_A(\cdot)$. One can verify that $|V_\varphi| \leq |\varphi|$, each node in G_φ has at most $|\Sigma| + |\varphi|$ attributes, $G_\varphi \models \Sigma$, but $G_\varphi \not\models \varphi$ by the definition of G_φ.

(2) We normalize the unboundedly large labels and attribute values in G_φ to construct $G_{(\Sigma,\varphi)}$ in the same manner as that in the satisfiability proof, in which large labels are substituted by a single label of small length and each attribute value in G_φ is replaced by a bounded-length solution to its corresponding variable in the linear programming problem $\mathsf{L}_{(\Sigma,\varphi)}$ constructed from G_φ. The only difference is that instantiated literals enforced by φ on G_φ is also processed in creating $\mathsf{L}_{(\Sigma,\varphi)}$.

We next show that $G_{(\Sigma,\varphi)}$ witnesses $\Sigma \not\models \varphi$, i.e., $G_{(\Sigma,\varphi)} \models \Sigma$ but $G_{(\Sigma,\varphi)} \not\models \varphi$, and it is of bounded size.

Indeed, $G_{(\Sigma,\varphi)} \models \Sigma$ can be verified analogously to its counterpart in the proof for satisfiability. Thus, we just prove that $G_{(\Sigma,\varphi)} \not\models \varphi$ by contradiction. Assume that $G_{(\Sigma,\varphi)} \models \varphi$. Then $h(\bar{x}) \models X$ and $h(\bar{x}) \models Y$ in $G_{(\Sigma,\varphi)}$ for the match h that was used in the creation of G_φ. Since $h(\bar{x}) \models X$ while $h(\bar{x}) \not\models Y$ in G_φ, there exists some literal $l = e_1 \otimes e_2$ in Y such that $h(\bar{x}) \not\models l$ in G_φ. Moreover, the instantiated l is involved in building the linear programming $\mathsf{L}_{(\Sigma,\varphi)}$. We assume *w.l.o.g.* that l is in the form of $e_1 > e_2$; the other cases can be proved similarly. By the definition of $\mathsf{L}_{(\Sigma,\varphi)}$, there exists a corresponding expression $e_1 \leq e_2$ in $\mathsf{L}_{(\Sigma,\varphi)}$, which is satisfied by any feasible solution to $\mathsf{L}_{(\Sigma,\varphi)}$. It follows that $h(\bar{x}) \not\models e_1 > e_2$ in $G_{(\Sigma,\varphi)}$, since the solutions to $\mathsf{L}_{(\Sigma,\varphi)}$ preserve the truth value of any original instantiated literal on G_φ. It contradicts to the assumption that $h(\bar{x}) \models Y$ in $G_{(\Sigma,\varphi)}$.

We now show that $|G_{(\Sigma,\varphi)}| \leq 6(|\Sigma| + |\varphi| + 1)^5$. Observe the following: (i) Graph $G_{(\Sigma,\varphi)}$ has at most $|\varphi|$ nodes and edges, since $G_{(\Sigma,\varphi)}$ uses the same sets of nodes and edges as G_φ. (ii) There are at most $|\varphi|$ many labels in $G_{(\Sigma,\varphi)}$, in which large ones are normalized with small length; and each node carries at most $|\Sigma| + |\varphi|$ attributes. (iii) The size of each attribute value in $G_{(\Sigma,\varphi)}$ is at most $6(|\Sigma| + |\varphi| + 1)^3$; this can be verified along the same lines as that in the proof of the satisfiability problem, leveraging the property of the bounded-length solution to linear programming. Putting these together, we have that the size $|G_{(\Sigma,\varphi)}|$ of $G_{(\Sigma,\varphi)}$ is at most $6(|\Sigma| + |\varphi| + 1)^5$.

Upper bound. Based on the small model property, we develop an Σ_2^p algorithm that given a set Σ of NGDs and an NGD $\varphi = Q[\bar{x}](X \to Y)$ checks whether $\Sigma \not\models \varphi$, as follows:

(1) Guess a graph $G = (V, E, L, F_A)$ such that $|G| \leq 6(|\Sigma| + |\varphi| + 1)^5$, and a mapping h_φ from V_Q to V, where V_Q denotes the set of nodes in the pattern Q.
(2) Check whether h_φ is a match of Q in G; if so, continue; otherwise, reject the current guess.
(3) Check whether $h_\varphi(\bar{x}) \models X$ and $h_\varphi(\bar{x}) \not\models Y$; if so, continue; otherwise, reject the current guess.
(4) Check whether $G \models \Sigma$; if so, return true; otherwise, reject the current guess.

The correctness of the algorithm is assured by the small model property. For its complexity, step (2) is in PTIME by the definition of matches. Step (3) is in PTIME, since $|X| + |Y| \leq |\varphi|$. Step (4) is in coNP (Theorem 5.1). Thus, the algorithm is in Σ_2^p, and the implication problem for NGDs is in Π_2^p.

Lower bound. We show that the implication problem is Π_2^p-hard by reduction from the complement of the GSSP (see GSSP in the proof of the satisfiability problem). Given two integral vectors $\bar{u}_1 = (u_1, \ldots, u_m)$ and $\bar{u}_2 = (u'_1, \ldots, u'_n)$, and another integer w, we construct a set Σ of NGDs and

another NGD φ such that $\Sigma \not\models \varphi$ if and only if $\exists \bar{v}_1 \forall \bar{v}_2(\bar{u}_1^T \cdot \bar{v}_1 + \bar{u}_2^T \cdot \bar{v}_2 \neq w)$. That is, we find a graph G "witnessing" $\Sigma \not\models \varphi$ when the condition in GSSP holds.

We borrow some constructions from the lower bound proof for the satisfiability problem. Recall the graph pattern $Q[x_1, \ldots, x_m, y_0, y_1, z_1, \ldots, z_n, z]$ and the third NGD φ_3 given there. We define $\Sigma = \{\varphi_3\}$, which encodes the two given vectors $\bar{u}_1$ and $\bar{u}_2$ and checks whether $\forall \bar{v}_2(\bar{u}_1^T \cdot \bar{v}_1 + \bar{u}_2^T \cdot \bar{v}_2 \neq w)$ for some fixed $\bar{v}_1$. The other NGD φ is defined as $\varphi = Q[x_1, \ldots, x_m, y_0, y_1, z_1, \ldots, z_n, z]((|2 \times x_1.A - 1| = 1) \wedge \cdots \wedge (|2 \times x_m.A - 1| = 1) \wedge (y_0.B = 0) \wedge (y_1.B = 1) \rightarrow (z.C = 2))$, to encode the possible vector $\bar{v}_1$, where the pattern Q of φ is the same as that of φ_3.

Observe that φ ensures that for any match h of Q in a graph G, if $h(x_i).A$ is a Boolean value for $i \in [1, m]$, $h(y_0).B = 0$ and $h(y_1).B = 1$, then $h(z).C$ must be 2. In addition, we can deduce 2^n many matches of Q in G for a given h by changing $h(z_i)$ ($i \in [1, n]$) to $h(y_0)$ or $h(y_1)$ arbitrarily. Moreover, for each such deduced match h', if $h'(A' + B') = w$ holds (i.e., $h'(x_1).A \times u_1 + \cdots + h'(x_m).A \times u_m + h'(z_1).B \times u_1' + \cdots + h'(z_n).B \times u_n' = w$), then $h'(z).C = 2$ as assured by φ_3 of Σ.

From these, we establish the relationship between the implication problem for the NGDs Σ and φ constructed above and GSSP and show that $\Sigma \not\models \varphi$ if and only if $\exists \bar{v}_1 \forall \bar{v}_2(\bar{u}_1^T \cdot \bar{v}_1 + \bar{u}_2^T \cdot \bar{v}_2 \neq w)$ holds.

($\Rightarrow$) Assume that $\Sigma \not\models \varphi$. We show that there exists an m-component vector $\bar{v}_1 = (s_1', \ldots, s_m')$ such that $\forall \bar{v}_2(\bar{u}_1^T \cdot \bar{v}_1 + \bar{u}_2^T \cdot \bar{v}_2 \neq w)$. By $\Sigma \not\models \varphi$, there exists a graph G such that $G \models \Sigma$ but $G \not\models \varphi$. Since $G \not\models \varphi$, there is a match h of Q in G such that $h(\bar{x}, \bar{y}, \bar{z}) \models (|2 \times x_1.A - 1| = 1) \wedge \cdots \wedge (|2 \times x_m.A - 1| = 1) \wedge (y_0.B = 0) \wedge (y_1.B = 1)$ and $h(\bar{x}, \bar{y}, \bar{z}) \not\models (z.C = 2)$. Based on h, we define the Boolean vector $\bar{v}_1$ such that $s_i' = h(x_i).A$ for $i \in [1, m]$. This is well-defined, since $h(x_i).A$ has value 0 or 1.

It remains to show that $\forall \bar{v}_2(\bar{u}_1^T \cdot \bar{v}_1 + \bar{u}_2^T \cdot \bar{v}_2 \neq w)$. Assume by contradiction that there exists a Boolean vector $\bar{v}_2 = (t_1', \ldots t_n')$ such that $\bar{u}_1^T \cdot \bar{v}_1 + \bar{u}_2^T \cdot \bar{v}_2 = w$. Then, we show that $h(z).C$ must be 2, which contradicts to $h(\bar{x}, \bar{y}, \bar{z}) \not\models (z.C = 2)$, as argued above. To see this, it suffices to apply φ_3 of Σ. That is, we construct a match h' of Q in G such that $h'(z) = h(z)$ and $h'(\bar{x}, \bar{y}, \bar{z}) \models (|2 \times z_1.B - 1| = 1) \wedge \cdots \wedge (|2 \times z_n.B - 1| = 1) \wedge (A' + B' = w)$. For if it holds, then $h'(z).C = h(z).C = 2$ by $G \models \Sigma$. The match h' is constructed as follows: (a) $h'(x_i) = h(x_i)$ for $i \in [1, m]$; (b) $h'(y_0) = h(y_0)$, $h'(y_1) = h(y_1)$, $h'(z) = h(z)$; and (c) $h'(z_i) = h(y_0)$ when $t_i' = 0$, or $h'(z_i) = h(y_1)$ when $t_i' = 1$ for $i \in [1, n]$. Since z_i's are labeled wildcards that can match any label, h' is also a match of Q in G. One can see that $|2 \times h'(z_i).B - 1| = 1$ for $i \in [1, n]$, since $h(y_0).B = 0$ and $h(y_1).B = 1$, and $h'(z_i).B$ takes the value from them. Moreover, one can verify that $h'(\bar{x}, \bar{y}, \bar{z}) \models (A' + B' = w)$ by the construction of h' and the assumption for $\bar{v}_1$ and $\bar{v}_2$ defined above. This leads to $h(z).C = 2$, a contradiction.

($\Leftarrow$) Conversely, assume that $\exists \bar{v}_1 \forall \bar{v}_2(\bar{u}_1^T \cdot \bar{v}_1 + \bar{u}_2^T \cdot \bar{v}_2 \neq w)$. Let $\bar{v}_1 = (s_1', \ldots, s_m')$ be such an integral vector. Based on $\bar{v}_1$, we construct a graph G such that $G \models \Sigma$ but $G \not\models \varphi$, i.e., G "witness" $\Sigma \not\models \varphi$. Graph G is similar to the pattern $Q = (V_Q, E_Q, L_Q, \mu)$, except that it includes a subset of nodes and carries attributes. More specifically, $G = (V, E, L, F_A)$ is defined as follows: $V = \{v \mid v \in V_Q, L_Q(v) \neq \_\}$, consisting of those nodes in V_Q that are not labeled wildcard; $E = E_Q$, i.e., the empty set $\emptyset$; $L(v) = L_Q(v)$ for each $v \in V$; and F_A is such defined that $F_A(v).A = s_i'$ if $L(v) = \tau_i$ for $i \in [1, m]$, $F_A(v).B = 0$ if $L(v) = \gamma_0$, $F_A(v).B = 1$ if $L(v) = \gamma_1$, and $F_A(v).C = 1$ if $L(v) = \chi$.

One can verify that $G \not\models \varphi$, since Q has a match in G, each node labeled τ_i ($i \in [1, m]$) in G carries Boolean attribute A, and the only node labeled χ in G carries C-attribute 1 instead of 2.

It remains to show that $G \models \Sigma$. Assume by contradiction that $G \not\models \Sigma$. Then there exists a match h of Q in G such that $h(\bar{x}, \bar{y}, \bar{z}) \models (A' + B' = w)$ and $h(\bar{x}, \bar{y}, \bar{z}) \models (|2 \times z_i.B - 1| = 1)$ for $i \in [1, n]$. We now define an n-component Boolean vector $\bar{v}_2 = (h(z_1).B, \ldots, h(z_n).B)$; this is well-defined, since $h(z_i).B$'s are Boolean values. By the definitions of graph G and match h, we have that $h(x_i).A = s_i'$ for $i \in [1, m]$. Hence, one can verify that $h(A' + B') = \bar{u}_1^T \cdot \bar{v}_1 + \bar{u}_2^T \cdot \bar{v}_2$. Thus, $\bar{u}_1^T \cdot \bar{v}_1 + \bar{u}_2^T \cdot \bar{v}_2 = w$, which contradicts to the assumption that $\forall \bar{v}_2(\bar{u}_1^T \cdot \bar{v}_1 + \bar{u}_2^T \cdot \bar{v}_2 \neq w)$. □

One might think that the complexity comes from interactions between arithmetic operations and comparison predicates. This is not the case: The lower bounds still hold when either arithmetic expressions or built-in predicates are present, not necessarily both.

Corollary 4.3. *For* NGDs, *the satisfiability, strong satisfiability, and implication problems remain* Σ_2^p*-complete,* Σ_2^p*-complete, and* Π_2^p*-complete, respectively, even in the absence of either (a) arithmetic operations or (b) comparison predicates* $\neq, <, \leq, >, \geq$.

Proof. (a) We first consider the case of NGDs without arithmetic operations. For the upper bounds, observe that all algorithms given in the proof of Theorem 4.2 are still in Σ_2^p, as it is still in PTIME to check whether (i) a guessed mapping h is a match; (ii) $G \models \Sigma$; and (iii) $h(\bar{x}) \models X$ and $h(\bar{x}) \not\models Y$.

The lower bounds for the strong satisfiability and implication problems follow from their counterparts for graph denial constraints (GDCs) without id literals, which are known to be Σ_2^p-hard and Π_2^p-hard, respectively [31]. Indeed, these GDCs are the same as NGDs in the absence of arithmetic.

It remains to show the lower bound for the satisfiability problem for NGDs without arithmetic operations. We prove the Σ_2^p-hardness by reduction from the strong satisfiability problem of GDCs without id literals. Given a set Σ_1 of GDCs in the absence of id literals, we build a set Σ of NGDs such that Σ_1 is strongly satisfiable if and only if Σ is satisfiable. Suppose that Σ_1 consists of the following GDCs (without the id literals): $\varphi_1' = Q_1[\bar{x}_1](X_1 \rightarrow Y_1), \ldots, \varphi_n' = Q_n[\bar{x}_n](X_n \rightarrow Y_n)$. We construct n NGDs in the absence of arithmetic operations such that these NGDs share the same pattern Q, which is a combination of $Q_1, \ldots, Q_n$. Meanwhile, for each $\varphi_i' = Q_i[\bar{x}_i](X_i \rightarrow Y_i)$ in Σ_1, where pattern $Q_i = (V_{Q_i}, E_{Q_i}, L_{Q_i}, \mu_{Q_i})$, Σ includes one NGD such that the constraint $X_i \rightarrow Y_i$ is enforced only with the pattern nodes V_{Q_i} of Q_i in Q. More specifically, Σ is constructed as follows:

(1) The graph pattern $Q[\bar{x}_1, \ldots, \bar{x}_n] = (V_Q, E_Q, L_Q, \mu)$ is shared by all NGDs in Σ. It is defined as the union of $Q_1, \ldots, Q_n$. That is, $V_Q = V_{Q_1} \cup \ldots \cup V_{Q_n}$; $E_Q = E_{Q_1} \cup \ldots \cup E_{Q_n}$; $L_Q(v) = L_{Q_i}(v)$ if $v \in V_{Q_i}$; and similarly for $L_Q(e)$ with $e \in E_Q$; and $\mu(x_i) = \mu_{Q_i}(v)$ if x_i is from Q_i. To simplify the discussion, we assume *w.l.o.g.* that Q_i and Q_j are disjoint for all $i, j \in [1, n]$ and $i \neq j$. In general, Q is built as the disjoint union of the patterns $Q_1, \ldots, Q_n$ without affecting the arguments below.

(2) For each $\varphi_i' = Q_i[\bar{x}_i](X_i \rightarrow Y_i)$ in Σ_1, Σ includes NGD $\varphi_i = Q[\bar{x}](X_i \rightarrow Y_i)$. Since we require that the patterns in Σ_1 are disjoint, $X_i \rightarrow Y_i$ is also the only constraint associated with the nodes in Q_i.

We show that these make a reduction, i.e., Σ_1 is strongly satisfiable if and only if Σ is satisfiable.

($\Rightarrow$) Suppose that Σ_1 is strongly satisfiable and that G is such a model of Σ_1. Then $G \models \Sigma_1$, and there exists a match of Q_i in G for all $i \in [1, n]$. We prove that G also witnesses the satisfiability of Σ. As Q contains disjoint patterns $Q_1, \ldots, Q_n$, we can readily verify that Q also has a match in G.

We now prove that $G \models \Sigma$ by contradiction. If $G \not\models \Sigma$, then there exists an NGD $\varphi_i = Q[\bar{x}](X_i \rightarrow Y_i)$ in Σ and a match h of Q in G such that $h(\bar{x}) \models X_i$, but $h(\bar{x}) \not\models Y_i$. By the definition of Σ, there exists a corresponding GDC $\varphi_i' = Q_i[\bar{x}_i](X_i \rightarrow Y_i)$ in Σ_1. We show that $G \not\models \varphi_i'$, which contradicts $G \models \Sigma_1$ and hence $G \models \Sigma$ follows. By the definition of Q, we can deduce a match h_i of Q_i in G as follows: for each $x \in \bar{x}_i$, $h_i(x) = h(x)$. Here $\bar{x}_i$ is the list of variables in Q_i. Since X_i and Y_i are literals defined on the vertices from V_{Q_i}, $h(\bar{x}) \models X_i$, and $h(\bar{x}) \not\models Y_i$, we know that $h_i(\bar{x}_i) \models X_i$ and $h_i(\bar{x}_i) \not\models Y_i$. That is, $h_i(\bar{x}_i) \not\models (X_i \rightarrow Y_i)$. Therefore, $G \not\models \varphi_i'$.

($\Leftarrow$) Suppose that Σ is satisfiable and graph G is such a model of Σ. Then $G \models \Sigma$, and there exists a match h of Q in G. We show that G witnesses the strong satisfiability of Σ_1. As argued above, there exists a match of Q_i in G for any $i \in [1, n]$. It remains to show that $G \models \Sigma_1$.

We show that $G \models \Sigma_1$ by contradiction. If $G \not\models \Sigma_1$, then there exists a GDC $\varphi_i' = Q_i[\bar{x}_i](X_i \rightarrow Y_i)$ and a match h_1 of Q_i in G such that $h_1(\bar{x}_i) \models X_i$, but $h_1(\bar{x}_i) \not\models Y_i$. By the definition of

Σ, $\varphi_i = Q[\bar{x}](X_i \to Y_i)$ is an NGD included in Σ. It suffices to show that $G \not\models \varphi_i$. For if it holds, then it contradicts the assumption that $G \models \Sigma$. By the definition of Q, we can deduce a match h_2 of Q in G as follows: when $x \in x_i$, $h_2(x) = h_1(x)$; otherwise, $h_2(x) = h(x)$. Here h is the match derived by the assumption of $G \models \Sigma$. Because the satisfiability of X_i and Y_i depends solely on the attributes at $h_2(x)$ for $x \in \bar{x}_i$, we know that $h_2(\bar{x}) \models X_i$ and $h_2(\bar{x}) \not\models Y_i$, i.e., $h_2(\bar{x}) \not\models (X_i \to Y_i)$. Hence, $G \not\models \varphi_i$.

(b) We next show that the satisfiability, strong satisfiability, and implication problems are Σ_2^p-complete, Σ_2^p-complete, and Π_2^p-complete, respectively, for NGDs in the absence of comparison predicates. To see the lower bound, observe that the encodings used in the lower bound proofs of Theorem 4.2 do not use any comparison predicates for NGDs, i.e., the only built-in predicate involved is =. The upper bounds can be verified in the same way as in (a) above. □

4.2 Impact of Non-linear Arithmetic Expressions

One might want to support arithmetic expressions that are not linear, defined as

$$e \ ::= \ t \mid |e| \mid e + e \mid e - e \mid e \times e \mid e \div e.$$

That is, expression e is built up from terms c and $x.A$ by closing them under arithmetic operators, *not necessarily of degree at most 1.* A literal is defined as $e_1 \otimes e_2$ as before, where e_1 and e_2 are arithmetic expressions of $Q[\bar{x}]$, and $\otimes$ is one of $=, \neq, <, \leq, >, \geq$.

This extension, however, makes the static analyses undecidable, even for NGDs with literals of a bounded degree. The undecidability justifies our choice of linear arithmetic expressions for NGDs.

Theorem 4.4. *The satisfiability, strong satisfiability, and implication problems become undecidable for* NGDs *with non-linear arithmetic expressions, even when*

- *no arithmetic expressions in the* NGDs *have degree above 2,*
- *and none of* $\neq, <, \leq, >, \geq$ *predicate is present.*

Proof. We study the three problems one-by-one, starting from the satisfiability problem.

Satisfiability. We show that the satisfiability problem becomes undecidable for extended NGDs. It is verified by reduction from Hilbert's 10th problem, denoted by HTP, which is undecidable [45, 55]. HTP is to decide, given a polynomial Diophantine equation in the form of $\sum_{i=1}^{n} a_i y_1^{n_{1,i}} \cdots y_m^{n_{m,i}} = 0$, where $a_1, \ldots, a_n$ are integer coefficients and $n_{1,i}, \ldots, n_{m,i}$ are non-negative integer exponents for each $i \in [1, n]$, whether there exists a feasible solution of integers for $(y_1, \ldots, y_m)$.

Given a Diophantine equation, we construct a set $\Sigma = \{\varphi\}$ of extended NGDs such that Σ is satisfiable if and only if the equation has a solution of integers. We encode the semantics of polynomials in a recursive manner by using literals of a single NGDφ with built-in predicate = only, and none of the arithmetic expressions in φ has degree above 2.

We start by illustrating the idea of recursive encoding with an example. Consider a polynomial of $3y_1y_2^5$. We first use a literal $x'_{1,0}.A = 3$ to encode the coefficient 3. We then encode the exponentiation y_1 and y_2^5. The former is simply expressed by another integer variable $x^1_{1,1}.A$, while the latter is encoded recursively leveraging three literals $x^5_{2,4}.A = x^2_{2,2}.A \times x^3_{2,3}.A$, $x^3_{2,3}.A = x^2_{2,2}.A \times x^1_{2,1}.A$, and $x^2_{2,2}.A = x^1_{2,1}.A \times x^1_{2,1}.A$. That is, we encode y_2^5 by decomposing it into y_2^3 and y_2^2 of smaller exponents, which are also encoded as literals with integer variables $x^3_{2,3}.A$ and $x^2_{2,2}.A$, respectively. Indeed, the exponentiation of y_i^j has a corresponding integer variable $x^j_{i,k}.A$ in the encoding (if exists). We finally encode the entire polynomial, also following a recursive strategy. We decompose each polynomial into a sub-expression followed by a suffix of single exponentiation and encode these two separately. For instance, we encode $3y_1y_2^5$ with two literals $x'_{1,2}.A = x'_{1,1}.A \times x^5_{2,4}.A$ and

$x'_{1,1}.A = x'_{1,0}.A \times x^1_{1,1}.A$, i.e., $3y_1y_2^5$ is split into $3y_1$ and suffix y_2^5, which are expressed by $x'_{1,1}.A$ and $x^5_{2,4}.A$, respectively. Taken together, $3y_1y_2^5$ is expressed as integer variable $x'_{1,2}.A$ eventually.

Formally, for each polynomial $a_i y_1^{n_{1,i}} \cdots y_m^{n_{m,i}} (i \in [1,n])$ in the given equation, (1) coefficient a_i is encoded by a single literal $x'_{i,0}.A = a_i$; (2) each exponentiation $y_j^{n_{j,i}} (j \in [1,m])$ is encoded in terms of the encoding of $y_j^{\lfloor n_{j,i}/2 \rfloor}$ and $y_j^{\lceil n_{j,i}/2 \rceil}$ recursively with a literal $x^{n_{j,i}}_{j,k}.A = x^{\lfloor n_{j,i}/2 \rfloor}_{j,l}.A \times x^{\lceil n_{j,i}/2 \rceil}_{j,p}.A$; and (3) the entire polynomial $a_i y_1^{n_{1,i}} \cdots y_m^{n_{m,i}}$ is recursively encoded through the encoding of sub-expression $a_i y_1^{n_{1,i}} \cdots y_{m-1}^{n_{m-1,i}}$ and exponentiation $y_m^{n_{m,i}}$ with a literal $x'_{i,m}.A = x'_{i,m-1}.A \times x^{n_{m,i}}_{m,k}.A$, where $x'_{i,m-1}.A$ denotes the corresponding integer variable of the sub-expression. One can verify that each exponentiation $y_j^{n_{j,i}}$ can be encoded by using at most $2\lceil \log_2 n_{j,i} \rceil$ literals, and the encoding of the entire polynomial needs m more literals. Thus, the total size of the encoding is polynomial in the size of the given equation, yielding a PTIME reduction.

Based on the recursive encoding, we define an extended NGD $\varphi = Q[\bar{x}](\emptyset \rightarrow (Z_1 \wedge Z_2))$, where Z_1 includes literals for encoding the polynomials as described above, and Z_2 checks the existence of integer solutions to the equation. Let Σ consist of φ only. More specifically, φ is defined as follows:
(1) The graph pattern $Q[\bar{x}] = (V_Q, E_Q, L_Q, \mu)$ is such defined that (a) $V_Q = \{v'_{i,0} \mid i \in [1,n]\} \cup \{v^{N_{j,i}}_{i,j} \mid i \in [1,m], j \in [1,K_i]\} \cup \{v'_{i,j} \mid i \in [1,n], j \in [1,m]\} \cup \{v''\}$, where K_i refers to the number of variables introduced to encode all the exponentiations of base y_i in the equation for each $i \in [1,m]$, $N_{j,i}$'s ($i \in [1,m], j \in [1,K_i]$) indicate their exponents, i.e., $x^{N_{j,i}}_{i,j}$ is the corresponding integer variable of exponentiation $y_i^{N_{j,i}}$ in the recursive encoding; indeed, the nodes are split into four groups to encode the coefficients, the exponentiation, the polynomials, and the equation, respectively; (b) $E_Q = \emptyset$; (c) all nodes in V_Q are associated with distinct labels excluding wildcard by L_Q; and (d) for each $i \in [1,n]$, $\mu(x'_{i,0}) = v'_{i,0}$; for each $i \in [1,m]$ and $j \in [1,K_i]$, $\mu(x^{N_{j,i}}_{i,j}) = v^{N_{j,i}}_{i,j}$; and for each $i \in [1,n]$ and $j \in [1,m]$, $\mu(x'_{i,j}) = v'_{i,j}$ and $\mu(x'') = v''$, which maps variables from $\bar{x}$ to V_Q.

(2) Z_1 is the set of all literals introduced for recursively encoding the polynomials of the equation, including $x'_{i,0}.A = a_i$ to encode coefficient, $x^{N_{j,i}}_{i,j}.A = x^{\lfloor N_{j,i}/2 \rfloor}_{i,l}.A \times x^{\lceil N_{j,i}/2 \rceil}_{i,p}.A$ to encode exponentiation, and $x'_{i,j}.A = x'_{i,j-1}.A \times x^{n_{j,i}}_{j,k}.A$ to encode the polynomial.

(3) $Z_2 = (x''.A = x'_{1,m}.A + x'_{2,m}.A + \cdots + x'_{n,m}.A) \wedge (x''.A = 0)$.

Observe that the extended NGDφ ensures the instantiation of integer variables, i.e., values of attribute A, must satisfy all the literals enforced by the recursive encoding, and it enforces a feasible solution to the given Diophantine equation by Z_2. Based on this, we next show that Σ is satisfiable if and only if the given Diophantine equation has an integer solution.

($\Rightarrow$) First assume that Σ is satisfiable. Then there exists a graph G such that $G \models \Sigma$ and Q has a match h in G. We show that $(h(x^1_{1,1}).A, \ldots, h(x^1_{m,1}).A)$ is a feasible solution to the equation, in which attribute $x^1_{i,1}.A$ must exist by the definition of satisfiability. Assume by contradiction that $\sum_{i=1}^n a_i (h(x^1_{1,1}).A)^{n_{1,i}} \cdots (h(x^1_{m,1}).A)^{n_{m,i}} \neq 0$. Since $h(\bar{x}) \models (\emptyset \rightarrow (Z_1 \wedge Z_2))$, we have that $h(x'').A = h(x'_{1,m}).A + \cdots + h(x'_{n,m}).A = \sum_{i=1}^n a_i (h(x^1_{1,1}).A)^{n_{1,i}} \cdots (h(x^1_{m,1}).A)^{n_{m,i}} = 0$, a contradiction. This can be verified by repeatedly replacing $h(x'_{i,j}).A$ with $h(x'_{i,j-1}).A \times h(x^{n_{j,i}}_{j,k}).A$, $h(x^{N_{j,i}}_{i,j}).A$ with $h(x^{\lfloor N_{j,i}/2 \rfloor}_{i,l}).A \times h(x^{\lceil N_{j,i}/2 \rceil}_{i,p}).A$, and $h(x'_{i,0}).A$ with a_i, until only a_i's and $h(x^1_{j,1}).A$'s are left. This is well-defined, since h satisfies all the literals in Z_1 that are introduced to encode the computation of polynomials.

($\Leftarrow$) Conversely, assume that the given equation has a solution $(b_1, \ldots, b_m)$ of integers. Based on this solution, we build a graph G such that $G \models \Sigma$ and there exists a match of Q in G. We define

$G = (V, E, L, F_A)$ as follows: (1) $V = V_Q$, $E = E_Q$ and $L = L_Q$, i.e., it takes the same nodes, edges, and labels as in the pattern Q; (2) F_A is such defined that (a) for each $i \in [1, n]$, $F_A(v'_{i,0}).A = a_i$; (b) for each $i \in [1, m]$ and each $j \in [1, K_i]$, $F_A(v_{i,j}^{N_{j,i}}).A = b_i^{N_{j,i}}$; (c) for each $i \in [1, n]$ and each $j \in [1, m]$, $F_A(v'_{i,j}).A = a_i b_1^{n_{1,i}} \dots b_j^{n_{j,i}}$; and (d) $F_A(v'').A = \sum_{i=1}^{n} a_i b_1^{n_{1,i}} \dots b_m^{n_{m,i}}$. Intuitively, G is the same as the graph pattern Q except the associated attributes, which are assigned values of the coefficients, exponentiation, polynomials, and sum of polynomials in the given equation when variables are instantiated by the feasible solution $(b_1, \dots, b_m)$. Since G has the same topological structure as that in Q, and all nodes carry distinct labels, we know that there only exists a single match h of Q in G.

It remains to show that $h(\bar{x}) \models Z_1 \wedge Z_2$. Suppose by contradiction that there exists a literal l in Z_1 or Z_2 such that $h(\bar{x}) \not\models l$. Observe the following: (a) By the definition of F_A, we have that $F_A(v'_{i,0}).A = a_i$, $F_A(v_{i,j}^{N_{j,i}}).A = F_A(v_{i,l}^{\lfloor N_{j,i}/2 \rfloor}).A \times F_A(v_{i,p}^{\lceil N_{j,i}/2 \rceil}).A$ and $F_A(v'_{i,j}).A = F_A(v'_{i,j-1}).A \times F_A(v_{j,k}^{n_{j,i}}).A$. Thus, h satisfies all the literals in Z_1. (b) Since $F_A(v'').A = F_A(v'_{i,m}).A + \dots + F_A(v'_{n,m}).A$, we have that $h(\bar{x}) \models (x''.A = x'_{1,m}.A + x'_{2,m}.A + \dots + x'_{n,m}.A)$. Hence, the only literal that is not satisfied by h is $x''.A = 0$. As a result, $F_A(v'').A = \sum_{i=1}^{n} a_i b_1^{n_{1,i}} \dots b_m^{n_{m,i}} \neq 0$, contradicting the assumption that $(b_1, \dots, b_m)$ is a feasible solution to the Diophantine equation.

Strong satisfiability. The proof for the satisfiability problem above suffices to verify the undecidability of the strong satisfiability problem, since the set Σ used in the reduction there consists of a single extended NGD. Hence, the satisfiability and strong satisfiability problems of Σ coincide.

Implication. We show that the implication problem for extended NGDs is undecidable by reduction from the complement of HTP. Given an equation $\sum_{i=1}^{n} a_i y_1^{n_{1,i}} \cdots y_m^{n_{m,i}} = 0$, we build a set Σ of extended NGDs and a NGD φ_1 such that $\Sigma \not\models \varphi_1$ if and only if the equation has a solution.

Recall the graph pattern Q and the extended NGD $\varphi = Q[\bar{x}](\emptyset \rightarrow (Z_1 \wedge Z_2))$ defined in the proof of the satisfiability problem above. Let Σ consist of one NGD $\varphi_2 = Q[\bar{x}](\emptyset \rightarrow (Z_1 \wedge Z'_2))$, where Z'_2 is obtained from Z_2 by removing literal $x''.A = 0$ to encode the computation of polynomials as in the proof for satisfiability; and $\varphi_1 = Q[\bar{x}](x''.A = 0 \rightarrow x''.B = 1)$ to check the existence of solutions. We next show that $\Sigma \not\models \varphi_1$ if and only if $\sum_{i=1}^{n} a_i y_1^{n_{1,i}} \cdots y_m^{n_{m,i}} = 0$ has an integer solution.

($\Rightarrow$) First suppose that $\Sigma \not\models \varphi_1$. We show that there exists an integer solution $(b_1, \dots, b_m)$ to the equation. By $\Sigma \not\models \varphi_1$, there exists a graph G such that $G \models \Sigma$ but $G \not\models \varphi_1$. Since $G \not\models \varphi_1$, there is a match h of Q in G such that $h(x'').A = 0$ and $h(x'').B \neq 1$. Based on h, we define $(b_1, \dots, b_m)$ such that $b_i = h(x_{i,1}^1).A$ for each $i \in [1, m]$. It remains to show that $(b_1, \dots, b_m)$ is a feasible solution, which can be proved by contradiction using similar arguments to that in the proof of satisfiability, i.e., applying the rules of encoding recursively to express $h(x'').A$ by a_i's and $h(x_{j,1}^1.A)$'s.

($\Leftarrow$) Conversely, suppose that the equation has an integer solution of $(b_1, \dots, b_m)$. We construct a graph G' such that $G' \models \Sigma$, but $G' \not\models \varphi_1$. Recall the graph $G = (V, E, L, F'_A)$ constructed in the proof of the satisfiability problem above. Graph $G' = (V, E, L, F'_A)$ is the same as G except that an additional B-attribute value 2 is associated with node v'', i.e., $F'_A(v'').B = 2$. Since $G \models \varphi$ and φ_2 is obtained from φ by removing one literal, $G' \models \Sigma$. Moreover, by the construction of G', we have that there exists only one match h of Q in G. It remains to show that $G' \not\models \varphi_1$, i.e., $h(\bar{x}) \not\models \varphi_1$. Assume by contradiction that $h(\bar{x}) \models \varphi_1$. Then $h(x'').A \neq 0$, since $h(x'').B = F'_A(v'').B = 2$. However, $h(x'').A$ must be 0 as argued in the proof of the satisfiability problem, hence a contradiction. □

5 DETECTING ERRORS WITH NGDS

We have seen that NGDs provide uniform rules for capturing inconsistencies in graphs, numeric or not (Section 3). We next study error detection and incremental detection in graphs by using NGDs as data quality rules. We also establish the (parameterized) complexity of these two problems.

5.1 Detecting Inconsistencies in Graphs

Given an NGD $\varphi = Q[\bar{x}](X \rightarrow Y)$ and a graph G, we say that a match $h(\bar{x})$ of Q in G is a *violation* of φ if $G_h \not\models \varphi$, where G_h is the subgraph induced by $h(\bar{x})$. For a set Σ of NGDs, we denote by $\mathsf{Vio}(\Sigma, G)$ the set of all violations of NGDs in G, i.e., $h(\bar{x}) \in \mathsf{Vio}(\Sigma, G)$ if there exists an NGD φ in Σ such that $h(\bar{x})$ is a violation of φ in G. That is, $h(\bar{x})$ violates at least one NGD in Σ.

The *error detection problem*, also known as the *validation problem*, is stated as follows:

- *Input*: A set Σ of NGDs and a graph G.
- *Output*: The set $\mathsf{Vio}(\Sigma, G)$ of violations.

That is, when NGDs are used as data quality rules, it is to find all inconsistent entities in a graph.

Its decision version is to decide, given a set Σ of NGDs and a graph G, whether $\mathsf{Vio}(\Sigma, G) = \emptyset$.

It is known that the validation problem for GFDs is coNP-complete [33]. The good news is that the problem gets no harder for NGDs, despite their increased expressive power.

Corollary 5.1. *The validation problem for* NGDs *remains* coNP*-complete.*

Proof. We first show that the validation problem is in coNP, and then show that it is coNP-hard.

Upper bound. We use an NP algorithm to check, given graph G and set Σ of NGDs, whether $G \not\models \Sigma$.

(1) Guess an NGD $\varphi = Q[\bar{x}](X \rightarrow Y)$ in Σ, and a mapping h from Q to G.
(2) Check whether h is a match of Q in G; if so, continue; otherwise, reject the current guess.
(3) Check whether $h(\bar{x}) \models X$ but $h(\bar{x}) \not\models Y$; if so, return true; otherwise, reject the current guess.

The correctness of the algorithm follows from the semantics of NGDs. For its complexity, step (2) is in PTIME, which follows from the definition of matches. Step (3) is also in PTIME, since $|X| + |Y| \leq |\Sigma|$. Thus, the algorithm is in NP, and the validation problem is in coNP.

Lower bound. Since GFDs are a special case of NGDs, and the validation problem for GFDs is coNP-complete [31], we have that the validation problem for NGDs is also coNP-hard. □

Parameterized complexity. We now study the fixed-parameter tractability of the validation problem. An instance of a *parameterized problem* $\mathcal{P}$ is a pair (x, k), where x is an input as in the conventional complexity theory, and k is a parameter that characterizes the structure of x. A parameterized problem $\mathcal{P}$ is called *fixed-parameter tractable*, denoted by FPT, if there exists a computable function g, a constant c, and an algorithm $\mathcal{A}$ such that given an instance (x, k) of P with parameter k, $\mathcal{A}$ takes $O(g(k)|x|^c)$ time to solve x. Intuitively, if k is small, then it is feasible to solve the problem efficiently although $g(k)$ could be exponential (see, e.g., Reference [35], for details).

For a set Σ of NGDs, we denote by k the size of the largest pattern node set in Σ, i.e., $k = \max\{|\bar{x}| \mid Q[\bar{x}](X \rightarrow Y) \in \Sigma\}$. For a graph G, denote by d the maximum degree of the nodes in G.

We parameterize the validation problem as follows:

- Input: A set Σ of NGDs and a graph G.
- Parameters: $k = \max\{|\bar{x}| \mid Q[\bar{x}](X \rightarrow Y) \in \Sigma\}$, and the maximum degree d of G.
- Question: Does $G \models \Sigma$?

We show that the parameterized validation problem is also nontrivial.

Theorem 5.2. *With parameters k and d, the validation problem is (1) co-*W[2]*-hard for* NGDs*; but (2) it is in* FPT *for* NGDs *defined with connected patterns.*

Here W[2] is the set of parameterized problems that can be FPT-reducible to the parameterized dominating set problem [5]. It is conjectured that FPT problems are properly contained in W[2]. Thus, the validation problem for NGDs remains nontrivial even when the parameters k and d are small. The problem for NGDs seems to be harder than for GFDs, which is shown to be W[1]-hard [27], where W[1] consists of parameterized problems that can be FPT-reducible to the parameterized independent set problem [22]. It is conjectured that W[1] is properly contained in W[2] [5].

The good news is that the parameterized validation problem is in FPT for NGDs defined with connected patterns. For an NGD $\varphi = Q[\bar{x}](X \to Y)$, its pattern Q is *connected* if for any two nodes x and y in Q there exists an undirected path between x and y. We find that most NGDs in practice are defined with connected patterns. For example, all patterns in Figure 2 are connected.

Proof. (1) We show the co-W[2]-hardness by reduction from the complement of the k-set cover problem, which is known W[2]-complete [16]. The k-set cover problem is to decide, given a finite family of sets $S = \{S_1, S_2, \ldots, S_n\}$ and a positive integer k, whether there exists a subset $S' \subseteq S$ such that S' has at most k sets and the union of sets in S' contains all elements in $S_1 \cup \cdots \cup S_n$.

Given S and k, suppose that $\mathcal{E} = S_1 \cup S_2 \cup \cdots \cup S_n = \{a_1, \ldots, a_m\}$. We construct two positive integers k_1 and d, a graph G, and a set Σ of NGDs such that $k_1 = \max\{|\bar{x}| \mid Q[\bar{x}](X \to Y) \in \Sigma\}$, d is the maximum degree of nodes in G, and $G \not\models \Sigma$ if and only if there exists a k-set cover of S. Intuitively, we will use (a) n isolated nodes in G to represent $S_1, \ldots, S_n$, (b) attributes $A_1, \ldots, A_m$ in G to encode elements in $\mathcal{E}$, and (c) literals in Σ to check set cover. More specifically, we define $k_1 = k$ and $d = 0$, i.e., all nodes in G are isolated. We construct G and Σ as follows:

Graph. The graph $G = (V, E, L, F_A)$ is defined as follows: (a) $V = \{v_1, \ldots, v_n\}$, i.e., one node for each set in S; (b) $E = \emptyset$, i.e., all nodes are isolated; (c) for each node $v_i \in V$, $L(v_i) = \tau_i$; and (d) for each $v_i \in V$, assume that $S_i = \{a_{i_1}, \ldots, a_{i_l}\}$; then $F_A(v_i).A_{i_1} = 1, \ldots,$ and $F_A(v_i).A_{i_l} = 1$; and for all the other elements $a_t \notin S_i$, $F_A(v_i).A_t = 0$. That is, each attribute represents one element in $\mathcal{E}$; moreover, if the element is in S_i, the corresponding attribute value is 1; otherwise, the value is 0. Note that, since all nodes in G are isolated, the maximum degree of the nodes in G is 0, i.e., $d = 0$.

NGDs. The set Σ consists of a single NGD $\varphi = Q[\bar{x}](X \to Y)$ to verify the k-set cover, where (a) Q consists of k isolated pattern nodes $x_1, \ldots, x_k$, which are labeled with wildcard "_"; (b) X consists of the following literals: $(x_1.A_1 + \cdots + x_k.A_1 > 0) \wedge \cdots \wedge (x_1.A_m + \cdots + x_k.A_m > 0)$, i.e., attribute A_i ($i \in [1, m]$) is required to have value 1 in at least one of nodes $x_1, \ldots, x_k$; hence, when a match of Q satisfies these literals, we can deduce a k-set cover of S from the mappings of $x_1, \ldots, x_k$; (c) Y consists of literals $x_1.A = 1 \wedge x_1.A = 2$, i.e., no match of Q can satisfy Y. Observe that, (i) since Q has k pattern nodes, $|Q| \leq k_1$; and (ii) although there can be less than k sets in a k-set cover S' and Q has k nodes, it suffices to use only one NGD φ, since multiple pattern nodes in Q can be mapped to the same node in G by the homomorphism semantics of pattern matching.

It remains to show that $G \not\models \Sigma$ if and only if there exists a k-set cover of S.

($\Rightarrow$) Suppose that $G \not\models \Sigma$. We show that there exists a k-set cover of S. Since $G \not\models \Sigma$, there exists a match h of Q in G such that $h(\bar{x}) \models X$ but $h(\bar{x}) \not\models Y$. Let $h(x_1) = v'_1, \ldots, h(x_k) = v'_k$. From $h(\bar{x}) \models X$, we can deduce sets $S'_1, \ldots, S'_k$ by using the nodes $v'_1, \ldots, v'_k$, i.e., each S'_i refers to v'_i. Since $h(\bar{x}) \models X$, by the definition of X, for each attribute A_i, at least one node v'_j ($j \in [1, k]$) satisfies that $v'_j.A_i = 1$, i.e., each element a_i is in at least one of $S'_1, \ldots, S'_k$. Thus, $S'_1, \ldots, S'_k$ form a k-set cover of S.

($\Leftarrow$) Conversely, assume that there exists a k-set cover $S'_1, \ldots, S'_k$ of S. We show that $G \not\models \Sigma$ by constructing the following match h of Q in G: $h(x_1) = v'_1, \ldots, h(x_k) = v'_k$. Here $v'_1, \ldots, v'_k$ are nodes denoting $S'_1, \ldots, S'_k$ in G. We verify that $h(\bar{x}) \models X$ but $h(\bar{x}) \not\models Y$ as follows: (a) For $h(\bar{x}) \models X$, since

$S'_1, \ldots, S'_k$ is a set cover of S, each element $a_i \in \mathcal{E}$ is in one of $S'_1, \ldots, S'_k$; by the construction of G, we know that $h(x_1).A_i + \cdots + h(x_k).A_i > 0$. Hence, $h(\bar{x}) \models X$. (b) Since Y is $x_1.A = 1 \wedge x_1.A = 2$, no match can satisfy Y; therefore, $h(\bar{x}) \not\models Y$. Putting these together, we know that $G \not\models \Sigma$.

(2) Given a graph $G = (V, E, L, F_A)$ and a set Σ of NGDs defined with connected patterns, we provide the following algorithm to check whether $G \models \Sigma$:

(a) for each node $v \in V$, construct its k-neighbor $G_k(v)$ (see Section 6.1 for the definition);
(b) for each $G_k(v)$, check whether $G_k(v) \models \Sigma$; if so, return true; otherwise, return false.

For the correctness, since all patterns in Σ are connected and the maximum degree of the nodes in G is d, for each NGD $Q[\bar{x}](X \to Y)$ in Σ, every match of Q only involves the nodes in the k-neighbor $G_k(v)$ of some node v in G. So it suffices to verify whether $G_k(v) \models \Sigma$ for each v in V. For the complexity, step (a) is in $O(d^k|G|)$ time, since it only extracts $G_k(v)$ for each node v. For step (b), it is in $O(d^{k^2}|\Sigma||G|)$ time, since (i) the maximum degree of nodes in G is d, and $G_k(v)$ consists of at most d^k nodes; (ii) for each NGD in Σ, it has at most d^{k^2} matches; and (iii) there exist at most $|\Sigma|$ NGDs. Putting these together, the algorithm is in $O(d^{k^2}|\Sigma||G|)$ time. Thus, the parameterized validation problem is in FPT with k and d for NGDs defined with connected patterns. □

Detection algorithms. Using GFDs as data quality rules, algorithms have been developed for error detection [33]. The algorithms are *parallel scalable*, i.e., they guarantee to reduce the runtime of parallel algorithms relative to a yardstick sequential algorithm when using more processors (see Section 6.1). Hence, they can scale with real-life graphs by adding resources when the graphs grow big. The experimental study of Reference [33] has validated the scalability and efficiency of the algorithms.

A close examination of the algorithms of Reference [33] reveals that they can be readily extended to NGDs. Indeed, for the algorithms to work with NGDs on a graph G that is fragmented and distributed across processors, the only change involves local checking of NGDs in each fragment of G by adding arithmetic and comparison calculations; the generation of matches of graph patterns, which dominates the cost of the algorithms, remains unchanged. The workload estimation and balancing strategies of Reference [33] remain intact for NGDs. These strategies make the algorithms parallel scalable. As a result, the algorithms remain parallel scalable when they employ NGDs instead of GFDs.

Hence, parallel scalable algorithms are in place to uniformly detect semantic inconsistencies in real-life graphs, numeric or not, by using NGDs as data quality rules.

5.2 Incremental Error Detection

Error detection is costly in large G, and real-life graphs are frequently updated. This highlights the need for studying incremental error detection: We compute $\mathrm{Vio}(\Sigma, G)$ once, and then we incrementally compute $\mathrm{Vio}(\Sigma, G \oplus \Delta G)$ in response to frequent updates ΔG to G. This is more efficient than recomputing $\mathrm{Vio}(\Sigma, G \oplus \Delta G)$ starting from scratch when ΔG is small, as often found in practice.

We define a unit update as edge insertion (insert e) or deletion (delete e), which can simulate certain modifications. An insertion possibly introduces new nodes carrying labels, attributes, and values drawn from alphabets Γ, Θ, and U (Section 2), respectively, while deletions just remove the links, leaving the nodes otherwise unaffected. We formalize the problem as follows: We consider *batch update* ΔG consisting of a sequence of insertions and deletions of edges. Denote by

$$\Delta\mathrm{Vio}^+(\Sigma, G, \Delta G) = \mathrm{Vio}(\Sigma, G \oplus \Delta G) \setminus \mathrm{Vio}(\Sigma, G),$$
$$\Delta\mathrm{Vio}^-(\Sigma, G, \Delta G) = \mathrm{Vio}(\Sigma, G) \setminus \mathrm{Vio}(\Sigma, G \oplus \Delta G),$$
$$\Delta\mathrm{Vio}(\Sigma, G, \Delta G) = (\Delta\mathrm{Vio}^+(\Sigma, G, \Delta G), \Delta\mathrm{Vio}^-(\Sigma, G, \Delta G)),$$

denoting new errors introduced by ΔG, removed by ΔG, and their combination, respectively.

The *incremental error detection problem* can now be stated as follows:

- *Input*: Graph G, set Σ of NGDs, and batch update ΔG to G.
- *Output*: The changes $\Delta\mathsf{Vio}(\Sigma, G, \Delta G)$ to $\mathsf{Vio}(\Sigma, G)$.

We do not require $\mathsf{Vio}(\Sigma, G)$ as part of the input, since the set may be exponential in size and is costly to store. Its decision problem is to decide whether $\Delta\mathsf{Vio}(\Sigma, G, \Delta G) = \emptyset$.

It is not surprising that the incremental error detection problem is nontrivial.

Theorem 5.3. *It is* coNP*-complete to decide, given a set* Σ *of* NGDs*, a graph* G*, and a batch update* ΔG*, whether* $\Delta\mathsf{Vio}(\Sigma, G, \Delta G)$ *is empty, even when both* G *and* ΔG *have constant sizes.*

Proof. We start by giving an NP algorithm to check whether $\Delta\mathsf{Vio}(\Sigma, G, \Delta G)$ is not empty, and then show that the problem is coNP-hard even when both G and ΔG have constant sizes.

Upper bound. We provide an NP algorithm to check, given a set Σ of NGDs, a graph G, and batch update ΔG, whether $\Delta\mathsf{Vio}(\Sigma, G, \Delta G)$ is not empty, as follows:

(1) Guess two NGDs $\varphi_1 = Q_1[\bar{x}_1](X_1 \to Y_1)$ and $\varphi_2 = Q_2[\bar{x}_2](X_2 \to Y_2)$ in Σ, a mapping h_1 of Q_1 in G, and a mapping h_2 of Q_2 in $G \oplus \Delta G$.
(2) Check whether (a) h_1 is a match of Q_1 in G such that $h_1(\bar{x}_1) \models X_1$ and $h_1(\bar{x}_1) \not\models Y_1$ and (b) h_1 is not a match in $G \oplus \Delta G$; if so, return true; otherwise, continue.
(3) Check whether (a) h_2 is a match of Q_2 in $G \oplus \Delta G$, such that $h_2(\bar{x}_2) \models X_2$ and $h_2(\bar{x}_2) \not\models Y_2$, and (b) h_2 is not a match in G; if so, return true; otherwise, reject the guess.

The correctness of the algorithm follows from the definition of $\Delta\mathsf{Vio}(\Sigma, G, \Delta G)$; more specifically, steps (2) and (3) take care of edge deletions and insertions, respectively.

For its complexity, steps (2) and (3) are in PTIME, since $|X_1| + |Y_1| \leq |\Sigma|$ and $|X_2| + |Y_2| \leq |\Sigma|$. Therefore, the algorithm is in NP, and the incremental error detection with NGDs is in coNP.

Lower bound. We show that it is coNP-hard to decide whether $\Delta\mathsf{Vio}(\Sigma, G, \Delta G)$ is empty when both G and ΔG are of constant sizes. It is verified by reduction from the complement of the 3-colorability problem, which is to decide, given an undirected graph G, whether there exists a proper 3-coloring γ of G such that for each edge (u, v) in G, $\gamma(u) \neq \gamma(v)$. The problem is known to be NP-complete [59].

Given an undirected G, we construct a graph G', a set Σ of NGDs, and batch update $\Delta G'$, such that $\Delta\mathsf{Vio}(\Sigma, G', \Delta G')$ is not empty if and only if G has a proper 3-coloring. Intuitively, we use G' to encode the three colors, and Σ to encode the structure of G and to check the existence of 3-colorings.

(1) The graph $G' = (V', E', L', F'_A)$ is defined as follows:

- $V' = \{v'_1, v'_2, v'_3, v'_4\}$;
- $E' = \{(v'_1, v'_2), (v'_2, v'_1), (v'_2, v'_3), (v'_3, v'_2), (v'_3, v'_1), (v'_1, v'_3)\}$, which makes a 3-clique;
- $L'(v'_1) = r$, $L'(v'_2) = g$, and $L'(v'_3) = b$ for three colors, $L'(v'_4) = \chi$; for each edge $e \in E'$, $L'(e) = \tau$;
- $F'_A(v'_1).A = 1$, $F'_A(v'_2).A = 1$, and $F'_A(v'_3).A = 1$.

Intuitively, G' includes a 3-clique, in which each node represents a color. The extra node v_4 is an isolated node, which will be involved in a violation only after G' is changed (see below).

(2) The set Σ consists of a single NGD $\varphi = Q[x_1, \ldots, x_n, x_{n+1}](\emptyset \to (x_1.A = 3))$, where $n = |V|$, i.e., the number of nodes in G, and the pattern $Q[x_1, \ldots, x_n, x_{n+1}] = (V_Q, E_Q, L_Q, \mu)$ is defined as:

- $V_Q = V \cup \{v_{n+1}\} = \{v_1, \ldots, v_n, v_{n+1}\}$, i.e., it takes nodes of G and an isolated node v_{n+1};
- $E_Q = \{(u, v), (v, u) \mid (u, v) \in E\} \cup \{(v_i, v_{n+1}) \mid v_i \in V\}$, i.e., each undirected edge (u, v) in G is encoded with two directed edges (u, v) and (v, u), and all nodes in V are linked to v_{n+1};
- $L_Q(v_i) =$ "_" $(i \in [1, n])$, and $L_Q(v_{n+1}) = \chi$; for each edge $e \in E_Q$, $L_Q(e) = \alpha$, if e is incident to node v_{n+1}; otherwise, it is labeled τ;
- for each node $v_i (i \in [1, n + 1])$ in V_Q, $\mu(x_i) = v_i$.

(3) The update $\Delta G'$ is defined as three edge insertions of (v_1', v_4'), (v_2', v_4'), and (v_3', v_4') labeled α.

Intuitively, Σ is used to encode the structure of G and verify possible 3-colorings, while the extra node v_{n+1} and edges directing to it are introduced to ensure that $\Delta\text{Vio}(\Sigma, G', \Delta G')$ is not empty for the batch update $\Delta G'$ defined above. Note that both G' and $\Delta G'$ are of constant sizes.

We can show that $\Delta\text{Vio}(\Sigma, G', \Delta G')$ is not empty if and only if G has a proper 3-coloring. Indeed, $\text{Vio}(\Sigma, G')$ is empty before edge insertions, since Q does not have any match in G'. To see this, observe that (a) all nodes in Q are connected to v_{n+1} except itself, which is labeled χ, and (b) there is no edge directing to v_4' in G', which is the only node labeled χ, i.e., the only possible match of v_{n+1}, in G'. Thus, it suffices to show that $G' \oplus \Delta G' \not\models \Sigma$, i.e., $\text{Vio}(\Sigma, G' \oplus \Delta G')$ is not empty if and only if G has a proper 3-coloring, which can be easily verified by contradiction. We omit the details. □

Parameterized complexity. Analogous to the validation problem, we next study the parameterized complexity of the incremental error detection problem, which was not studied in Reference [27].

Theorem 5.4. *With parameters k and d, the incremental error detection problem is (1) co-*W[2]*-hard for* NGDs*, but (2) it is in* FPT *for* NGDs *defined with connected patterns.*

Proof. (1) We show that the incremental error detection problem is co-W[2]-hard by reduction from the validation problem for NGDs, which is shown co-W[2]-hard in Theorem 5.2.

Given a set Σ of NGDs and a graph G, we construct another set Σ_1 of NGDs and batch update ΔG to G such that $\text{Vio}(\Sigma, G) \neq \emptyset$ if and only if $\Delta\text{Vio}(\Sigma_1, G, \Delta G) \neq \emptyset$. Intuitively, (a) we introduce a specific pattern edge (x_i, x_j) labeled τ to each pattern in Σ to form Σ_1, where τ is a label not appearing in G and Σ, and pattern nodes x_i and x_j are labeled wildcard "_"; and moreover, (b) ΔG consists of the insertion of an edge (v, v') labeled τ. In this way, (i) Σ_1 cannot be applied on G and $\text{Vio}(\Sigma_1, G) = \emptyset$, since G does not contain any edge labeled τ; and (ii) after the insertion of (v, v') to G, we can verify that Σ can be applied on G if and only if Σ_1 can be applied on $G \oplus \Delta G$. Since $\text{Vio}(\Sigma_1, G) = \emptyset$, we know that $\text{Vio}(\Sigma, G) \neq \emptyset$ if and only if $\Delta\text{Vio}(\Sigma_1, G, \Delta G) \neq \emptyset$.

(2) Given a graph G, a set Σ of NGDs defined with connected patterns, and batch update ΔG, the FPT algorithm for the validation problem in Theorem 5.2 still works for incremental error detection. Thus, we check whether $\Delta\text{Vio}(\Sigma, G, \Delta G) = \emptyset$ as follows: (1) compute $\text{Vio}(\Sigma, G)$ and $\text{Vio}(\Sigma, G \oplus \Delta G)$ using the FPT algorithm above; and (2) compute $\text{Vio}(\Sigma, G \oplus \Delta G) \setminus \text{Vio}(\Sigma, G)$ and $\text{Vio}(\Sigma, G) \setminus \text{Vio}(\Sigma, G \oplus \Delta G)$, and check whether any of these is nonempty; if so, return false; otherwise, return true.

The correctness of the algorithm is immediate. For its complexity, observe that given a graph G and a set Σ of NGDs, the algorithm of Theorem 5.2 runs in $O(d^{k^2}|\Sigma||G|)$ time to compute $\text{Vio}(\Sigma, G)$. Then step (1) is in $O(d^{k^2}|\Sigma|(|G| + |\Delta G|))$ time. For step (2), the numbers of violations in $\text{Vio}(\Sigma, G \oplus \Delta G)$ and $\text{Vio}(\Sigma, G)$ are bounded by the number of all possible matches, which amount to $O(d^{k^2}|\Sigma|(|G| + |\Delta G|))$. Thus, the algorithm is in $O(d^{k^2}|\Sigma|(|G| + |\Delta G|))$ time, and the incremental error detection problem is in FPT with k and d for NGDs defined with connected patterns. □

In the following, we focus on (parallel) algorithms to incrementally detect inconsistencies in graphs by using NGDs. The algorithms complement the batch algorithms of Reference [33] for NGDs used as data quality rules. As remarked earlier, we are not aware of prior work on incremental error detection in graphs, and the static workload partitioning strategy of Reference [33] hampers the parallel scalability of the batch algorithms achieved there when being incrementalized directly.

6 INCREMENTAL DETECTION ALGORITHMS

Despite the challenges noted in Theorem 5.3, we develop two algorithms to incrementally detect errors in graphs with NGDs, and we show that the algorithms have certain performance guarantees.

We first review the performance guarantees (Section 6.1). We then present a sequential incremental error detection algorithm (Section 6.2), followed by a parallel algorithm (Section 6.3).

To simplify the discussion, we focus on NGDs defined with graph patterns Q that are connected. The algorithms can be readily extended to process NGDs that are defined with possibly disconnected patterns. More specifically, one can first compute (candidate) partial violations, by finding matches of distinct connected components in the patterns, following the same update-driven approach to be given shortly in this section. These partial matches are then combined to check attribute dependencies that go across multiple connected components to identify violations.

6.1 Performance Guarantees

We first review two characterizations of (parallel) incremental error detection algorithms.

(1) Locality. We first adapt a criterion from Reference [28]. (a) In a graph G, a node v' is *within d hops* of v if $\text{dist}(v, v') \leq d$ by taking G as an undirected graph, where $\text{dist}(v, v')$ is the shortest distance between v and v' in G. (b) We denote by $V_d(v)$ the set of all nodes in G that are within d hops of v. (c) The *d-neighbor* of v, denoted by $G_d(v)$, is the subgraph of G induced by $V_d(v)$ (see Section 2).

The *diameter* d_Q of a pattern Q is the maximum $\text{dist}(v, v')$ for all nodes v and v' in Q. For a set Σ of NGDs, *the diameter* d_Σ of Σ is the maximum diameter d_Q for all patterns Q that appear in Σ.

We say that an incremental error detection algorithm $\mathcal{A}$ is *localizable* if given a set Σ of NGDs, a graph G, and batch update ΔG to G, its cost is determined only by the size $|\Sigma|$ of NGDs and the sizes of the d_Σ-neighbors of those nodes on the edges of ΔG.

The notion of *localizable* was proposed in Reference [28] and has proven effective for graph queries by reducing computation on a (large) graph to small areas surrounding ΔG. We specialize it to incremental validation to compute $\Delta\text{Vio}(\Sigma, G, \Delta G)$ by checking only the d_Σ-neighbors $G_{d_\Sigma}(v)$ for nodes v that appear in ΔG. In practice, $G_{d_\Sigma}(v)$ is often small. Indeed, (a) Q is typically small, e.g., 98% of real-life patterns have radius 1 [39], which also indicate patterns in rules [38]; and (b) G is sparse, e.g., the average node degree is 14.3 in social graphs [15].

(2) Parallel scalability. The second criterion is adapted from [50], which has been widely used in practice to characterize the effectiveness of parallel algorithms. Consider a yardstick sequential algorithm $\mathcal{A}$ for incremental error detection, with its cost $t(|G|, |\Sigma|, |\Delta G|)$ measured in the sizes of graph G, set Σ of NGDs, and batch update ΔG.

A parallel algorithm $\mathcal{A}_p$ for incremental error detection is said to be *parallel scalable relative* to yardstick $\mathcal{A}$ if its parallel running time by using p processors can be expressed as follows:

$$T(|G|, |\Sigma|, |\Delta G|, p) = O\left(\frac{t(|G|, |\Sigma|, |\Delta G|)}{p}\right),$$

Algorithm IncDect
Input: Graph G, set Σ of NGDs and batch update ΔG=(ΔG^+, ΔG^-).
Output: The set $\Delta\text{Vio}(\Sigma, G, \Delta G)$ of violations.
1. initialize $\Delta\text{Vio}^+(\Sigma, G, \Delta G) := \emptyset$; $\Delta\text{Vio}^-(\Sigma, G, \Delta G) := \emptyset$;
2. **for each** NGD φ in Σ **do**
3. $(M^+_\varphi, M^-_\varphi) := \text{IncMatch}(\varphi, G, \Delta G)$;
4. $\Delta\text{Vio}^+(\Sigma, G, \Delta G) := \Delta\text{Vio}^+(\Sigma, G, \Delta G) \cup M^+_\varphi$;
5. $\Delta\text{Vio}^-(\Sigma, G, \Delta G) := \Delta\text{Vio}^-(\Sigma, G, \Delta G) \cup M^-_\varphi$;
6. **return** $(\Delta\text{Vio}^+(\Sigma, G, \Delta G), \Delta\text{Vio}^-(\Sigma, G, \Delta G))$;

Fig. 4. Algorithm IncDect.

where $p \ll |G|$, i.e., the number of processors used is much smaller than the sizes of real-life graphs G, as commonly found in the real world.

Intuitively, parallel scalability measures speed up over sequential algorithms by parallelization. It is a relative measure *w.r.t.* a yardstick algorithm $\mathcal{A}$. A parallel scalable $\mathcal{A}_p$ "linearly" reduces the running time of $\mathcal{A}$ when p increases. Hence, a parallel scalable algorithm is able to scale with large graphs G by adding processors as needed. It makes incremental detection feasible by increasing p.

6.2 A Sequential Localizable Algorithm

Below, we develop an exact localizable sequential algorithm, denoted by IncDect. Given a set Σ of NGDs, a graph G, and batch update ΔG, IncDect computes $\Delta\text{Vio}(\Sigma, G, \Delta G)$ with a single processor. Algorithm IncDect incrementalizes subgraph matching by following *update-driven evaluation*, and checks attribute dependencies with linear arithmetic expressions and comparison predicates.

Subgraph matching. We start by reviewing the general framework of subgraph matching. A number of subgraph matching algorithms have been developed for graphs, mostly following a backtracking-based procedure Match_n [53]. Given a pattern Q and a graph G, Match_n first identifies a set $C(u)$ of candidate matches for each pattern node u in Q. Then its main subroutine SubMatch_n recursively expands partial solution M by matching one pattern node of Q with a node of G, where M is a set of node pairs (u, v) indicating that v matches pattern node u. Subgraph homomorphism algorithms [36, 61] can also be characterized by the generic Match_n and SubMatch_n.

More specifically, given a partial solution M, SubMatch_n selects a pattern node u that is not yet matched and refines $C(u)$ following certain matching order selection and pruning strategies. For each refined candidate v in $C(u)$, it checks whether v can make a valid match of u by inspecting the correspondence between edges adjacent to u in Q and those edges connected to v in G. The qualified node pair (u, v) is added to M, and SubMatch_n is called recursively for further expansion until all the pattern nodes are matched. The partial solution M is restored when SubMatch_n backtracks.

Algorithm. IncDect (outlined in Figure 4) incrementalizes batch Match_n to process G, Σ, and $\Delta G = (\Delta G^+, \Delta G^-)$, where ΔG^+ and ΔG^- include all $\text{insert}(v, v')$ and $\text{delete}(v, v')$, respectively. (1) It starts with $\Delta\text{Vio}^+(\Sigma, G, \Delta G) = \emptyset$ and $\Delta\text{Vio}^-(\Sigma, G, \Delta G) = \emptyset$ (line 1). (2) For each NGD $\varphi = Q[\bar{x}](X \to Y)$ in Σ, it invokes a procedure IncMatch revised from Match_n to expand $\Delta\text{Vio}^+(\Sigma, G, \Delta G)$ (respectively, $\Delta\text{Vio}^-(\Sigma, G, \Delta G)$) with matches $h(\bar{x})$ of Q in $G \oplus \Delta G$ (respectively, G) such that (a) $h(u) = v$ and $h(u') = v'$ for some $(u, u') \in E_Q$ and $\text{insert}(v, v')$ in ΔG^+ (respectively, $\text{delete}(v, v')$ in ΔG^-) and (b) $h(\bar{x}) \not\models X \to Y$, in which the sets of these matches are denoted by M^+_φ and M^-_φ, respectively (lines 2–5).

Intuitively, edge insertions may introduce new violations and hence expand $\Delta\text{Vio}^+(\Sigma, G, \Delta G)$, but do not remove existing ones; however, deletions expand $\Delta\text{Vio}^-(\Sigma, G, \Delta G)$ only. IncMatch computes the newly added (respectively, removed) violations; this is done by identifying those matches that have nodes connected by edges in ΔG^+ (respectively, ΔG^-) and violate the attribute dependency.

Procedure IncMatch. We next give details of IncMatch and its subroutine IncSubMatch for processing an NGD $Q[\bar{x}](X \rightarrow Y)$. Following *update-driven* evaluation, we extend procedures Match_n and SubMatch_n to conduct (1) initial partial solution selection; (2) candidates filtering; and (3) arithmetic and comparison calculations.

(1) Given a graph pattern Q of NGD φ, procedure IncMatch first finds out whether each edge (v, v') in ΔG is a candidate match of some pattern edge (u, u') in Q by checking the labels. This is in contrast to its batch counterpart Match_n, which searches candidates in the entire graph G. If (v, v') makes a candidate, it forms an initial partial solution $h_{\text{up}}(u, u') = (v, v')$, referred to as an *update pivot* of Q *triggered by* unit update of edge (v, v'). IncMatch then expands $h_{\text{up}}(u, u')$ by recursively invoking IncSubMatch as in Match_n to compute update-driven violations $h(\bar{x})$.

(2) In each call, IncSubMatch searches candidates from the neighbors of those nodes that are already in a partial solution, starting from the update pivot. Each time, IncSubMatch picks a pattern node that is connected to some already matched ones. For a match $h(\bar{x})$ of Q in the updated graph $G \oplus \Delta G$ to be included in $\Delta\text{Vio}^+(\Sigma, G, \Delta G)$, (a) it must be expanded from an update pivot of Q triggered by unit edge insertion; and (b) there exist no v and v' in $h(\bar{x})$ such that $h(u) = v$ and $h(u') = v'$ for any $(u, u') \in E_Q$ while $\text{delete}(v, v')$ is in ΔG^-. Therefore, it leaves out edges in ΔG^- when retrieving candidates to expand the solutions from update pivots triggered by edge insertions. Similarly, it does not consider edges $\text{insert}(v, v')$ in ΔG^+ when expanding $\Delta\text{Vio}^-(\Sigma, G, \Delta G)$.

As an optimization strategy, IncMatch also marks the combination of multiple update pivots in partial solutions to prevent the same match from being enumerated more than once.

(3) The validation of literals with linear arithmetic expressions is performed by applying candidate pruning in IncSubMatch. More specifically, it evaluates a literal l in X as long as all the variables in l are instantiated, i.e., every variable that occurs in l is already matched or is being matched by the candidates under process, and prunes those when l is evaluated to be false. Literals in Y are handled similarly except that candidates contributing to true evaluations are pruned. Indeed, only matches $h(\bar{x})$ that satisfy $h(\bar{x}) \models X$ and $h(\bar{x}) \not\models Y$ are returned as violation.

Finally, those matches expanded from update pivots triggered by edge insertions (respectively, deletions) and violating $X \rightarrow Y$, referred to as *update-driven violations*, are returned by IncMatch and added to $\Delta\text{Vio}^+(\Sigma, G, \Delta G)$ (respectively, $\Delta\text{Vio}^-(\Sigma, G, \Delta G)$) by algorithm IncDect.

Example 6.1. Suppose that the edge (NatWest Help, 1) is deleted from graph G_4 of Figure 1. Given NGD φ_4 of Example 3.1, IncDect calls IncMatch to detect update-driven violations. It first finds that the deleted edge is a candidate match of (x, s_1) in Q_4. That is, an update pivot $h_{\text{up}}(x, s_1) =$ (NatWest Help, 1) is built. IncMatch then expands $h_{\text{up}}(x, s_1)$ recursively by inspecting the neighbors of candidate matches until all pattern nodes of Q_4 are matched. For instance, node 22,000 in G_4 is the only candidate match for pattern node m_1. Finally, it returns violation $h_{\text{up}}(\bar{x})$ to be removed, which includes all the nodes of G_4 and indicates that NatWest_Help is a fake account.

Besides delete(NatWest Help, 1), suppose that four edges are inserted into graph G_4 to specify that another account NatWest_Help_1 has one following and two followers and refers to company NatWest with status 0. Given this batch update, algorithm IncDect computes the same violation to be removed as above. Indeed, there are no newly introduced violations, since all matches expanded from update pivots triggered by edge insertions are pruned by literal validation.

Analysis. The correctness of algorithm IncDect is warranted by the following: The violations in $\Delta\text{Vio}^+(\Sigma, G, \Delta G)$ (respectively, $\Delta\text{Vio}^-(\Sigma, G, \Delta G)$) are matches of pattern Q in graph $G \oplus \Delta G$ (respectively, G) that contain inserted (respectively, deleted) edges of ΔG and violate attribute dependency $X \to Y$ for an NGD $Q[\bar{x}](X \to Y)$ in Σ, i.e., update-driven violations found by IncMatch.

IncDect runs in $O(|\Sigma||G_{d_\Sigma}(\Delta G)|^{|\Sigma|})$ time, where $G_{d_\Sigma}(\Delta G)$ denotes the union of d_Σ-neighbors of nodes involved in ΔG. Hence, it is localizable. Indeed, (a) the computation performed by each invocation of procedure IncMatch is confined in the d_Σ-neighbors of an unit update in ΔG. (b) Checking linear arithmetic expressions incurs much less cost than candidate selection in matching.

6.3 A Parallel Scalable Algorithm

Algorithm IncDect takes exponential time in the worst case. It is costly if the set Σ of NGDs or update ΔG is large or graph G is dense. This motivates us to develop algorithm PIncDect, which is parallel scalable relative to IncDect, to reduce response time by adding more processors when needed.

Overview. Algorithm PIncDect works with p processors $S_1, \ldots, S_p$ on graph G, which is partitioned via edge-cut [10] or vertex-cut [47]. In a nutshell, PIncDect finds update pivots of patterns in Σ triggered by unit updates and distributes partial solutions as work units to p processors. Then each processor handles its workload and identifies violations *in parallel*, driven by updates like in IncDect.

There are two challenges: (1) The d_Σ-neighbor of a node may reside in different fragments. (2) The workloads of some processors may be skewed, since (a) the workload assignment may be unbalanced; and (b) some work unit may take much longer, e.g., when accessing a large d_Σ-neighbor.

Note that work stealing and shedding [14, 42] cannot solve (b) by re-assigning work units. In a nutshell, both work stealing and work shedding are techniques to rebalance workload among workers. More specifically, after a worker has finished its computation, work stealing tries to assign to it some work units from busy workers; in contrast, work shedding identifies slow workers earlier using progress-report messages and then reassigns their workload to other workers. These techniques cannot solve stragglers of incremental error detection, since they have to move work units as a whole; when it comes to NGDs, a work unit is decided by the d_Σ-neighbor of some node and could be costly. When a work unit is redistributed to an idle processor S_i by work stealing or shedding, the chances are that S_i becomes a new straggler if it takes much longer to process the unit compared to others. As a consequence, the parallel response time is not improved at all.

To cope with this, PIncDect does the following: It finds and distributes the *candidate neighborhood* of each update pivot. Then all processors interact with each other asynchronously to expand and verify partial solutions by accessing candidate neighborhoods only. To reduce skewness, PIncDect (a) splits and parallelizes the *work unit* of filtering and verifying a candidate, based on cost estimation, and (b) periodically redistributes partial solutions (work units) to be expanded from busy processors to those with light loads. This makes PIncDect parallel scalable relative to IncDect.

Candidate neighborhood. Similar to IncDect, initially PIncDect checks whether each unit update of edge (v, v') in ΔG triggers an update pivot $h_{\text{up}}(u, u') = (v, v')$ for some pattern nodes u and u' in Q at each processor. It then identifies the d_Q-neighbor of v in $G \oplus \Delta G^+$, i.e., the *candidate neighborhood* $N_C(h_{\text{up}}(u, u'))$ for $h_{\text{up}}(u, u')$. When node v is involved in multiple update pivots, only the union of their candidate neighborhoods is extracted. The processors coordinate to extract

such an area when it is fragmented by notifying each other of the remaining size to be explored via messages passed through crossing edges in edge cut or entry and exit nodes in vertex cut.

All processors broadcast the data extracted such that the union $N_C(\Delta G, \Sigma)$ of candidate neighborhoods for update pivots is replicated at each processor. We find that $N_C(\Delta G, \Sigma)$ is often much smaller than graph G when ΔG and Σ are small, as commonly found in practice.

Moreover, for each node v in $N_C(\Delta G, \Sigma)$, PIncDect evenly "partitions" its adjacency list $v.\mathsf{adj}$ by annotating local partition (instead of physically breaking it up), such that each processor S_i holds a *partial copy* $v.\mathsf{adj}_i$. The update pivots are also evenly partitioned into p disjoint sets such that each processor S_i maintains one set BVio_i as its *workload*. A partial solution to be expanded is a *work unit*.

Parallel validation. All processors expand partial solutions to find update-driven violations in parallel. For each partial solution in BVio_i, processor S_i expands it by matching a pattern node that is not matched yet until a complete match (violation) is found. This is done by *candidate filtering* followed by *verification*. It adopts a hybrid processing strategy to split and parallelize skewed work units. Algorithm PIncDect also periodically balances workloads across p processors to reduce skewed workloads by generating and distributing a large number of work units.

We next give the insights of the two steps for expanding partial solutions, which dominate the cost of algorithm PIncDect.

Candidate filtering. Consider $h_{\mathsf{up}}(u_0, \ldots, u_k)$ in BVio_i, a partial solution for Q to be expanded at certain processor S_i. The next pattern node to be matched is u_{k+1} such that it is connected to u_r in Q for $r \in [0, k]$. The candidates for u_{k+1} are selected from the neighbors of $h_{\mathsf{up}}(u_r)$, just like in procedure IncSubMatch (Section 6.2). Here PIncDect estimates the *sequential cost* as $|h_{\mathsf{up}}(u_r).\mathsf{adj}|$, and the *parallel cost* as $C(k+1) + |h_{\mathsf{up}}(u_r).\mathsf{adj}|/p$, for expanding the partial solution by matching u_{k+1}, where C is a constant referred to as *communication latency*, and $C(k+1)$ denotes the broadcasting cost. It conducts expansion at processor S_i directly by inspecting candidates from $h_{\mathsf{up}}(u_r).\mathsf{adj}$ if the sequential cost is less than the parallel one. Otherwise, $h_{\mathsf{up}}(u_0, \ldots, u_k)$ is broadcast to all the processors and is expanded in parallel by checking the partial copy $h_{\mathsf{up}}(u_r).\mathsf{adj}_j$ reserved at each S_j for $j \in [1, p]$. This allows us to reduce a skewed work unit with large adjacency lists.

Verification. After $h_{\mathsf{up}}(u_0, \ldots, u_k)$ is expanded with u_{k+1} at processor S_i, PIncDect checks the edges between the candidate $h_{\mathsf{up}}(u_{k+1})$ and other matches $h_{\mathsf{up}}(u_0), \ldots, h_{\mathsf{up}}(u_k)$ to verify the validity of the expansion. It may split the verification work. Here the sequential cost is estimated as $|h_{\mathsf{up}}(u_{k+1}).\mathsf{adj}|$ and the parallel cost is $C(k+2) + |h_{\mathsf{up}}(u_{k+1}).\mathsf{adj}|/p$. If the parallel cost is smaller, it broadcasts $h_{\mathsf{up}}(u_0, \ldots, u_k, u_{k+1})$ to check at each S_j by using its partial copy $h_{\mathsf{up}}(u_{k+1}).\mathsf{adj}_j$. The results of checking are then sent back to S_i for assembling to decide the qualification of the partial solution, which are added to BVio_i for further expansion if qualified, unless it makes a complete match of Q.

Workload balancing. The workload of a processor S_i is *skewed* if its local set BVio_i of update pivots contains far more work units than the others at the same time. This happens even if we start with evenly distributed update pivots, since different partial solutions may trigger radically different number of new work units. We define *the skewness of* S_i as $\frac{\|\mathsf{BVio}_i\|}{\mathsf{avg}_{t\in[1,p]}\|\mathsf{BVio}_t\|}$.

To handle this, PIncDect checks the skewness of processors at a time interval intvl. If the skewness of S_i exceeds a threshold η (3 in experiments), it evenly distributes the work units in BVio_i to those S_j's having skewness below η' (0.7 in experiments), extending BVio_j's. The processors are allowed to send and receive work units at any time without being blocked by synchronization barriers.

Algorithm PIncDect
Input: A fragmented graph G across p processors $S_1, \ldots S_p$,
a set Σ of NGDs, and a batch update ΔG.
Output: The set $\Delta\text{Vio}(\Sigma, G, \Delta G)$ of violations.
1. **for each** unit update of (v, v') in ΔG and pattern edge (u, u') in Q of NGD $\varphi = Q[\bar{x}](X \rightarrow Y) \in \Sigma$ **having** $L_Q(u) = L(v)$, $L_Q(u') = L(v)$, **and** $L_Q(u, u') = L(v, v')$ **do**
2. construct update pivot $h_{\text{up}}(u, u') = (v, v')$;
3. identify the d_{Q_u}-neighbor of v;
4. construct $N_C(\Delta G, \Sigma)$ in parallel and replicate at all processors;
5. evenly partition adjacency lists and work units;
6. invoke PIncMatch(BVio_i) at processor S_i for all $i \in [1, p]$;
7. **repeat**
8. periodically balance workload across p processors at intvl;
9. **until** all S_i's return Vio_i;
10. $\Delta\text{Vio}(\Sigma, G, \Delta G) := \bigcup_i \text{Vio}_i$;
11. **return** $\Delta\text{Vio}(\Sigma, G, \Delta G)$;

Procedure PIncMatch */*executed at each S_i in parallel*/*
Input: Workload BVio_i.
Output: The set Vio_i of local violations.
1. $\text{Vio}_i := \emptyset$;
2. **while** there **exists** a partial solution to be expanded **do**
3. **for each** $h_{\text{up}}(u_0, \ldots, u_k) \in \text{BVio}_i$ by matching u_{k+1} with neighbors of $h_{\text{up}}(u_r)$ **do**
4. remove $h_{\text{up}}(u_0, \ldots, u_k)$ from BVio_i;
5. **if** $|h_{\text{up}}(u_r).\text{adj}| \leq C(k+1) + |h_{\text{up}}(u_r).\text{adj}|/p$ **then**
6. expand $h_{\text{up}}(u_0, \ldots, u_k)$ at S_i;
7. **else** broadcast $h_{\text{up}}(u_0, \ldots, u_k)$ and expand it in parallel;
8. **for each** $h_{\text{up}}(u_0, \ldots, u_k, u_{k+1})$ to be verified at S_i **do**
9. **if** $|h_{\text{up}}(u_{k+1}).\text{adj}| \leq C(k+2) + |h_{\text{up}}(u_{k+1}).\text{adj}|/p$ **then**
10. verify $h_{\text{up}}(u_0, \ldots, u_{k+1})$ at S_i;
11. **else** broadcast $h_{\text{up}}(u_0, \ldots, u_{k+1})$ and verify in parallel;
12. **if** $h_{\text{up}}(u_0, \ldots, u_k, u_{k+1})$ is a valid partial solution **then**
13. **if** it is a complete match **then** add it to Vio_i;
14. **else** add it to BVio_i;
15. **return** Vio_i;

Fig. 5. Algorithm PIncDect.

Algorithm. Putting these together, we present the main driver of algorithm PIncDect in Figure 5. It first identifies the candidate neighborhood for each update pivot (lines 1–3) and replicates the union of candidate neighborhoods at all processors (line 4). The update pivots are also evenly distributed (line 5). Then PIncDect invokes procedure PIncMatch at each processor S_i with initial workload BVio_i in parallel for $i \in [1, p]$ (line 6). It periodically balances workload (line 8) until all processors complete their work (line 9). At this point, PIncDect collects local violations Vio_i's from all processors. The union of all Vio_i's is $\Delta\text{Vio}(\Sigma, G, \Delta G)$ (line 10) and is hence returned (line 11).

At each processor S_i, procedure PIncMatch expands a partial solution by filtering candidate matches (lines 3–7), followed by parallel verification (lines 8–11). Both steps split skewed work units by applying the hybrid processing strategy based on cost estimation, as described earlier.

The local violations Vio_i and workload BVio_i are updated accordingly (lines 12–14). It returns Vio_i when no work units remain in BVio_i, i.e., when S_i finishes its workload (line 15).

Example 6.2. Consider a graph G revised from G_4 of Figure 1 by including additional 98 accounts NatWest_Help_i for $i \in [1, 98]$, where each one has one following and two followers and refers to company NatWest with status 1. Assume that graph G is fragmented and distributed across four processors. Recall NGD φ_4 and edge deletion delete(NatWest Help, 1) from Example 6.1. After generating update pivot $h_{\text{up}}(x, s_1)$ as in Example 6.1, algorithm PIncDect identifies in parallel $N_C(h_{\text{up}}(x, s_1))$, which is the 4-neighbor of node NatWest Help. This subgraph is replicated at four processors. Moreover, the adjacency lists are evenly "partitioned" by annotating partial copies. For instance, each processor maintains a partial copy of 25 nodes (i.e., accounts) for the adjacency list of the company node NatWest. Then it expands $h_{\text{up}}(x, s_1)$ to find update-driven violations.

Suppose that a partial solution $h_{\text{up}}(x, s_1, m_1, n_1, w)$ is to be expanded at processor S_j, where w is mapped to node NatWest, and the next pattern node to be matched is y. Then it is broadcast by S_j, and PIncDect expands it in parallel at each processor by mapping y to node NatWest_Help_i for some $i \in [1, 98]$ or NatWest_Help using the partial copies maintained for the adjacency list of node NatWest. Here the estimated parallel cost 30 is less than the sequential cost 100 when communication latency C is assumed to be 1; thus, parallel computation is favored.

Consider another partial solution of $h_{\text{up}}(x, s_1, m_1, n_1, w, y)$ to be expanded at processor S_j. Algorithm PIncDect expands it locally at S_j with the entire adjacency list of $h_{\text{up}}(y)$, since the size of $h_{\text{up}}(y).\text{adj}$, i.e., sequential cost of 4, is less than the estimated parallel cost.

Finally, a total of 99 violations are identified and added to $\Delta\text{Vio}^-(\Sigma, G, \Delta G)$, in which NatWest_Help_i and NatWest_Help are validated to be fake for each $i \in [1, 98]$.

Theorem 6.3. Given a set Σ of NGDs*, a graph G, batch update ΔG, p processors, when $p < |G_{d_\Sigma}(\Delta G)|$,* PIncDect *runs in $O(|\Sigma||G_{d_\Sigma}(\Delta G)|^{|\Sigma|}/p)$ time, i.e.,* PIncDect *is parallel scalable relative to* IncDect.

Proof. Obviously, identifying the candidate neighborhoods for update pivots triggered by update ΔG and NGDs Σ takes $O(|G_{d_\Sigma}(\Delta G)|)$ time. We next analyze the cost for parallel expansion. The total time for candidate filtering in processing partial solutions of size k is at most $N_k(Ck + |G_{d_\Sigma}(\Delta G)|/p)$, and their corresponding verification needs at most $N_{k+1}(C(k + 1) + |G_{d_\Sigma}(\Delta G)|/p))$ time, where N_k denotes the number of partial matches of size k. Moreover, it inspects partial solutions with size less than $|V_\Sigma|$, where V_Σ denotes the set of all pattern nodes in Σ. Hence, parallel expansion takes at most $\sum_{k=2}^{|V_\Sigma|-1} \big(N_k(Ck + \frac{|G_{d_\Sigma}(\Delta G)|}{p}) + N_{k+1}(C(k + 1) + \frac{|G_{d_\Sigma}(\Delta G)|}{p})\big) < \frac{4C|\Sigma|(1-|G_{d_\Sigma}(\Delta G)|^{|\Sigma|-1})|G_{d_\Sigma}(\Delta G)|^2}{(1-|G_{d_\Sigma}(\Delta G)|)p} = O(\frac{|\Sigma||G_{d_\Sigma}(\Delta G)|^{|\Sigma|}}{p})$ time when $p < |G_{d_\Sigma}(\Delta G)|$, which dominates the cost of PIncDect. This verifies the parallel scalability of algorithm PIncDect relative to the sequential IncDect. □

7 FINDING TOP-RANKED ERRORS

The intractability of incremental error detection with NGDs motivates us to identify special cases that can be solved in polynomial time. In practice, one often wants to find errors with high importance or interestingness quickly. Hence, below, we develop an algorithm for (incremental) detection of top-ranked inconsistencies with a particular type of NGDs in edge-weighted graphs.

We consider *star-like* NGDs as in Reference [71]. A NGD $\varphi = Q[\bar{x}](X \to Y)$ is called *star-like* if its pattern Q is star-shaped, i.e., there exists a *root* node r_Q in Q such that all the edges of Q are incident to it. As observed in References [39, 44], many real-life graph patterns are star-shaped,

and so are the patterns in NGDs, e.g., patterns Q_1 and Q_2 in NGDs φ_1 and φ_2 of Example 3.1, respectively.

Scoring function. The preference model for violations, i.e., matches $h(\bar{x})$ in edge-weighted graph G that violate an NGD, is specified by a scoring function $W(\cdot)$ that sums the weights of those edges connecting the nodes in $h(\bar{x})$. That is, $W(h) = \sum_{e \in E_{G_h}} w(e)$, where E_{G_h} denotes the edge set of the subgraph G_h of G induced by $h(\bar{x})$ and each edge e in G carries a positive weight value $w(e)$ from range $[0, 1]$, indicating the certainty of the relationship, i.e., the link it establishes.

Since real-life knowledge bases often contain incorrect links [11], the users are often more interested in those errors having higher total confidence of the links. It is very likely that these inconsistencies are critical and hence are given high priority for inspection or fixing.

Based on the scoring function $W(\cdot)$, we next present two algorithms for detection and incremental detection of top-ranked errors, respectively. It should be remarked that our proposed algorithms also work for the scoring functions defined with conventional similarity metrics [71].

Detection of top-ranked inconsistencies. Given a set Σ of star-like NGDs, a graph G, and a positive integer $K \ll |G|$, the *top-ranked error detection problem* is to find the violations $\mathrm{Vio}(\Sigma, G, K)$ in the union $\bigcup_Q M(Q, G, K)$ of the top-K match sets of the patterns Q from Σ in G. Here $M(Q, G, K)$ is a set of matches of cardinality K such that for each match $h \in M(Q, G, K)$, $W(h) \geq W(h')$ for all other matches h' of Q that are not in $M(Q, G, K)$; and $\mathrm{Vio}(\Sigma, G, K)$ denotes the set of violations of NGDs in Σ that appear in the union $\bigcup_Q M(Q, G, K)$ of the top-K match sets.

A sequential algorithm can be readily deduced to compute $\mathrm{Vio}(\Sigma, G, K)$ in $O(|\Sigma||G|)$ time by extending procedure StarK of Reference [71], also denoted as StarK. It works as follows: (1) It invokes StarK [71] to find the top-K match set $M(Q, G, K)$ for each pattern Q from Σ. (2) It checks each match h in $\bigcup_Q M(Q, G, K)$ and includes it in $\mathrm{Vio}(\Sigma, G, K)$ if h violates an attribute dependency in Σ.

We now parallelize StarK, denoted as PRDect and shown in Figure 6. Similar to the setting of Section 6.3, PRDect is deployed on a graph G that is fragmented and distributed across p processors $S_1, \ldots, S_p$, among which a designated S_c works as the coordinator. It takes as input the graph G, a set Σ of star-like NGDs, and a positive integer K. It deduces $\mathrm{Vio}(\Sigma, G, K)$ in parallel. For each pattern Q in Σ, PRDect maintains two structures at coordinator S_c (lines 1–2): (1) a set $\mathcal{H}_Q$ to store the top-K match set of Q; and (2) a priority queue $\mathcal{R}_Q$ to keep track of *potential* top-K matches to be checked. The computation of $\mathcal{H}_Q$ is also controlled at S_c (lines 3–7), based on which PRDect returns the violations in the union of top-K match sets $\mathcal{H}_Q$ for different Q (line 8).

After initializing $\mathcal{H}_Q$ and $\mathcal{R}_Q$, the coordinator S_c first posts pattern Q and integer K to all the processors in procedure InitMatchK, which returns a gathered set of matches that are computed by procedure PRMatch at different processors (see below). These matches are candidate top-K matches of Q and are hence inserted into $\mathcal{R}_Q$ (line 3). It next expands $\mathcal{H}_Q$ iteratively until K matches are included or there is no potential top-K match in $\mathcal{R}_Q$ (line 4). Each time it adds to $\mathcal{H}_Q$ a match h that is popped from $\mathcal{R}_Q$ with the best score; it sends match h and pattern Q as messages to some processor S_j via procedure NextMatch and inserts the received next best matches induced by h to queue $\mathcal{R}_Q$ (lines 5–7). The next best matches are indeed derived by using the same match lists as that for h at processor S_j in procedure PRMatch (see details below).

Upon receiving a message M from coordinator S_c, processor S_i invokes procedure PRMatch to compute a set $\mathcal{H}_i$ of matches and sends it back to S_c. The set $\mathcal{H}_i$ is composed of either the potential top-K matches of pattern Q (lines 2–6) or the next top matches induced by some particular match h (lines 7–8). Here all the processors work in parallel synchronously guided by the coordinator S_c.

(1) When the message M contains pattern Q and an integer K, for each candidate match v of the root r_Q of Q in the local graph G_i, PRMatch builds a match list $L_v[e]$ for every pattern edge

Algorithm PRDect
Input: Fragmented G, set Σ of star-like NGDs and integer K.
Output: The set $\mathsf{Vio}(\Sigma, G, K)$ of top-K violations.
*/*executed at coordinator S_c */*
1. **for each** pattern Q that appears in Σ **do**
2. initialize set $\mathcal{H}_Q := \emptyset$; priority queue $\mathcal{R}_Q := \emptyset$;
3. add the matches in $\mathsf{InitMatchK}(Q, K, G)$ to $\mathcal{R}_Q$;
4. **while** $\mathcal{H}_Q$ has less than K matches **and** $\mathcal{R}_Q \neq \emptyset$ **do**
5. pop the best match h from $\mathcal{R}_Q$;
6. $\mathcal{H}_Q := \mathcal{H}_Q \cup \{h\}$;
7. add $\mathsf{NextMatch}(h, Q, G)$ to $\mathcal{R}_Q$;
8. **return** the violations in the union of $\mathcal{H}_Q$'s as $\mathsf{Vio}(\Sigma, G, K)$;

Procedure PRMatch */*executed at each processor S_i in parallel*/*
Input: Message M and local graph G_i.
Output: The set $\mathcal{H}_i$ of matches.
1. $\mathcal{H}_i := \emptyset$;
2. **if** M includes the pattern Q and integer K **then**
3. **for each** node v in G_i *s.t.* $L(v) = L_Q(r_Q)$ **do**
4. compute match list *w.r.t.* v for the pattern edges in Q;
5. retrieve the best match of Q *w.r.t.* v and add it to $\mathcal{H}_i$;
6. remove those that do not have the best K scores from $\mathcal{H}_i$;
7. **else if** M consists of a match h and a pattern Q **then**
8. find the next best matches *w.r.t.* $h(r_Q)$ and add them to $\mathcal{H}_i$;
9. **return** $\mathcal{H}_i$;

Fig. 6. Algorithm PRDect.

$e = (r_Q, u)$ or $e = (u, r_Q)$ in Q*w.r.t.* node v (lines 3–4). The entries in $L_v[e]$ are in the form of $\langle e', w(e')\rangle$, where e' is an edge from G_i that is a candidate match of pattern edge e and is incident to v. It adds to $\mathcal{H}_i$ a match of Q that is assembled by those candidates e' having best scores $w(e')$ in all $L_v[e]$, i.e., the best match *w.r.t.* v (line 5). Then the set $\mathcal{H}_i$ is further refined by reserving only K matches with the best K scores, which represents the potential top-K matches of Q at processor S_i (line 6). Note that when graph G is partitioned via vertex-cut, procedure PRMatch only maps root r_Q to primary copies of the nodes in S_i, and S_i may fetch data from other processors to identify candidates.

(2) If there exists a match h and a pattern Q in message M, PRMatch retrieves the match lists pertaining to h, i.e., h is assembled by using the entries in these lists. Each such list $L_Q[e]$ is sorted in advance with K entries based on the best K scores. It computes the set $\mathcal{H}_i$ of matches by moving the index for one node v of h in its corresponding sorted match list to the next position while keeping the others unchanged (line 8). To achieve this, each match is associated with its node indices in match lists. For instance, if the edges involved in h have indices, i.e., positions of $(7, 6, 3)$ in the sorted match lists, then three matches are deduced with edges having the indices of $(8, 6, 3)$, $(7, 7, 3)$, and $(7, 6, 4)$ from the same lists, respectively. That is, each such match differs from h in only one node and has a "slightly" smaller score, i.e., $\mathcal{H}_i$ includes the next matches induced by h.

Example 7.1. Recall the star-like NGD φ_1 from Example 3.1. Consider a graph G_f fragmented across two processors such that pattern edge $e_1 = (x, y)$ (respectively, $e_2 = (x, z)$) in Q_1 of φ_1 is mapped to candidates with scores, i.e., weights 0.4 and 0.6 (respectively, 0.5 and 0.8) at processor S_1 and 0.7 and 0.2 (respectively, 0.6 and 0.2) at S_2. Assume that $K = 2$ and the root x of Q_1 has only one match in each processor. Using procedure PRMatch, PRDect first finds potential top-3 matches

at the two processors, which are two matches h_1 and h_2 with scores 1.4 and 1.3, respectively. It then adds h_1 to the top-3 match set $\mathcal{H}_{Q_1}$ at the coordinator and computes the next best matches h_3 and h_4 at S_1 induced by h_1 with scores 1.2 and 1.1, respectively. Eventually the top-3 match set includes h_1, h_2, and h_3.

We have the following for the parallel scalability of PRDect, which is defined analogously to the notion given in Section 6.1, removing the parameter $|\Delta G|$:

Proposition 7.2. *If G is evenly partitioned, i.e., $|G_i| = O(|G|/p)$ for $i \in [1, p]$ and $Kp < \sqrt{|G|}$, algorithm* PRDect *takes $O(|\Sigma||G|/p)$ time and is parallel scalable relative to* StarK.

Proof. Since graph G is evenly partitioned, it needs at most $O(|\Sigma||G|/p)$ time to construct the match lists in parallel by using procedure PRMatch. Finding the potential top-K match sets takes $O(|\Sigma||G|/p + K \log K)$ time by PRMatch, since the discovery of the sorted best K values from a set of n values can be done in $O(n + K \log K)$ time [13]. The total time for computing the next best matches via PRMatch is bounded by $O(|\Sigma|(|G|/p + K^2 \log K))$, in which K iterations are conducted for each pattern from Σ. There are at most $Kp|\Sigma| + (K-1)|\Sigma|$ (respectively, $K|\Sigma|$) matches inserted to (respectively, popped from) the priority queues at the coordinator S_c, which takes $O(K|\Sigma|(p + \log K + \log |\Sigma|))$ time in total. Putting these together, algorithm PRDect runs in $O(|\Sigma|(|G|/p + K^2 \log K + Kp + K \log |\Sigma|)) = O(|\Sigma||G|/p)$ time when $Kp < \sqrt{|G|}$, and hence is parallel scalable relative to StarK. □

Incremental detection of top-ranked errors. Given a set Σ of star-like NGDs, a graph G, a positive integer $K \ll |G|$, and batch update ΔG to G, the *incremental top-ranked error detection problem* is to derive the difference $\Delta\mathrm{Vio}(\Sigma, G, \Delta G, K)$ between the violations $\mathrm{Vio}(\Sigma, G, K)$ and $\mathrm{Vio}(\Sigma, G \oplus \Delta G, K)$, which involves the newly introduced violations and deleted ones. The major challenge introduced by this problem includes the following: (1) each unit update may *both* introduce new top-ranked violations and remove existing top-ranked ones; and (2) it is no longer localizable, since the updated edges can trigger changes to the violations with edges that are far from them.

We next incrementalize algorithm PRDect, denoted by PIncRDect, to compute $\Delta\mathrm{Vio}(\Sigma, G, \Delta G, K)$. As shown in Figure 7, its input has graph G, star-like NGDs Σ, and integer K as for PRDect, and it takes as additional input the batch update ΔG and some auxiliary information, which consists of the top-K match sets $\mathcal{H}_Q$'s, priority queues $\mathcal{R}_Q$'s, and match lists $L_Q[e]$'s at different processors. All these are of polynomial-size and can be easily obtained when running PRDect on graph G.

The key idea behind PIncRDect is that (a) when some matches in the original top-K match sets involve deleted edges, they must be replaced by other new matches; and (b) while the inserted edges help generate new matches with better scores than the ones in the top-K match sets, the match sets should be adjusted to incorporate these new matches.

More specifically, for each pattern Q from Σ, algorithm PIncRDect first collects into set ΔG_Q candidate matches for pattern edges in Q at coordinator S_c (lines 1–3), i.e., update pivots (see Section 6.2). It next transmits the edges in nonempty ΔG_Q to the corresponding processors in which they reside by procedure UpdateMatchK (lines 4–5), which accumulates the *new* potential top-K matches of Q computed at those processors via procedure PIncRMatch (see below). PIncRDect inserts the new matches into queue $\mathcal{R}_i$, removes the old matches that involve deleted edges in ΔG_Q from the top-K match set $\mathcal{H}_Q$, and adds them to set $\Delta\mathcal{H}_Q$ as removed ones, where $\Delta\mathcal{H}_Q$ records the changes to $\mathcal{H}_Q$ (lines 5–6). It then updates the top-K match set $\mathcal{H}_Q$ in a way similar to that of PRMatch (lines 7–10) and tracks the changes in $\Delta\mathcal{H}_Q$ (line 11). The difference is that here some old matches h' in $\mathcal{H}_Q$ are replaced by the new matches popped from $\mathcal{R}_Q$ with better scores (line 9), and

Algorithm PIncRDect
Input: Fragmented graph G, set Σ of star-like NGDs, integer K, batch update ΔG and auxiliary information.
Output: The set $\Delta\text{Vio}(\Sigma, G, \Delta G, K)$ of violations.
*/*executed at coordinator S_c */*
1. **for each** pattern Q that appears in Σ **do**
2. initialize set $\Delta\mathcal{H}_Q := \emptyset$; $\Delta G_Q := \emptyset$;
3. **for each** (v, v') in ΔG that is a candidate match of a pattern edge in Q **do** add (v, v') to ΔG_Q;
4. **if** ΔG_Q is not empty **then**
5. add the matches in UpdateMatchK$(Q, K, G, \Delta G_Q)$ to $\mathcal{R}_Q$;
6. remove those matches that involve ΔG_Q from $\mathcal{H}_Q$ and add them as removed matches to $\Delta\mathcal{H}_Q$;
7. **while** $\mathcal{H}_Q$ has less than K matches, $\mathcal{R}_Q \neq \emptyset$ **or** there is a match in $\mathcal{R}_Q$ having better score than $h' \in \mathcal{H}_Q$ **do**
8. pop the best match h from $\mathcal{R}_Q$;
9. $\mathcal{H}_Q := (\mathcal{H}_Q \setminus \{h'\}) \cup \{h\}$;
10. add NextMatch(h, Q, G) to $\mathcal{R}_Q$;
11. add h (resp. h') to $\Delta\mathcal{H}_Q$ as new (resp. removed) match;
12. **return** the violations in the union of $\Delta\mathcal{H}_Q$'s;

Procedure PIncRMatch */*executed at each S_i in parallel*/*
Input: Message M and local graph G_i.
Output: The set $\mathcal{H}_i$ of matches.
1. $\mathcal{H}_i := \emptyset$;
2. **if** M contains pattern Q, integer K and updates ΔG_Q^i **then**
3. update the match lists *w.r.t.* nodes in ΔG_Q^i for Q;
4. **for each** node v in $G_i \oplus \Delta G_Q^i$ *s.t.* $L(v) = L_Q(r_Q)$ **do**
5. compute the best match of Q *w.r.t.* v that involves inserted edges in ΔG_Q^i and add it to $\mathcal{H}_i$;
6. remove those that do not have the best K scores from $\mathcal{H}_i$;
7. **else if** M has a match h and a pattern Q **then**
8. find the next best matches *w.r.t.* $h(r_Q)$ and add them to $\mathcal{H}_i$;
9. **return** $\mathcal{H}_i$;

Fig. 7. Algorithm PIncRDect.

procedure NextMatch returns the next best matches derived by procedure PIncRMatch at different processors (line 10). The violations in the union of $\Delta\mathcal{H}_Q$'s are finally returned at S_c (line 12).

Procedure PIncRMatch is invoked at each processor S_i in parallel synchronously when a message M is received. If M includes pattern Q, integer K, and updates ΔG_Q^i, it adjusts the match lists *w.r.t.* the nodes involved in the received updates and computes a set $\mathcal{H}_i$ of new potential top-K matches by using the updated match lists (lines 2–6). These matches must involve the inserted edges in ΔG_Q^i and have the best K scores among such new matches. When a match h and a pattern Q are sent to S_i, procedure PIncRMatch finds the (new) next best matches $\mathcal{H}_i$ induced by h along the same lines as that in procedure PRMatch of algorithm PIncRDect (lines 7–8).

Example 7.3. Continuing with Example 7.1, consider an inserted edge e_1' at processor S_1 and assume that e_1' is a candidate match of pattern edge e_1 with score 1. Given these, procedure PIncRMatch is invoked at S_1 to derive the new potential top-3 matches, which include match h_1' with score 1.8. Since h_1' has better score than h_3 in the original top-3 match set $\mathcal{H}_{Q_1}$, h_3 is replaced

by h'_1. Note that h'_1 involves the newly inserted edge e'_1. Algorithm PIncRDect then computes the new next match h'_2 induced by h'_1 at S_1, whose score is 1.5. It hence replaces h_2 with h'_2 in $\mathcal{H}_{Q_1}$, and the top-3 match set for Q_1 finally becomes $\{h'_1, h'_2, h_1\}$ after the edge insertion.

Complexity. Observe that (1) discovering update pivots needs $O(|\Sigma||\Delta G|)$ time; (2) updating match lists and computing new potential top-K match sets at all processors take $O(|\Sigma|(\max_{i\in[1,p]}|\Delta G_i| + K\log K))$ time, where ΔG_i denotes the input updates that reside at processor S_i; (3) the next best matches can be derived in at most $O(|\Sigma|\max_{i\in[1,p]}(|G_i| + |\Delta G_i|K\log K))$ time; and (4) maintaining priority queues at the coordinator needs $O(|\Delta G||\Sigma|(p + \log K + \log|\Sigma|))$ time.

One can verify that when G is evenly partitioned and $\max(K, |\Delta G|) < \sqrt{|G|}/p$, the cost of PIncRMatch is dominated by that for computing the next best matches, and PIncRMatch is parallel scalable relative to a sequential counterpart that conducts all the operations at a single processor.

8 EXPERIMENTAL STUDY

Using real-life and synthetic graphs, we evaluated the impact of (1) the size $|\Delta G|$ of updates; (2) the size $|G|$, the density η_G, and the degree distribution of graphs; (3) the complexity of sets Σ of NGDs; (4) the integer K for PTIME algorithms (Section 7); and (5) the number p of processors for parallel processing, as well as the factors of communication latency C and time interval intvl for work unit splitting and workload balancing, on our (parallel) algorithms for incremental (top-ranked) error detection; and (6) the effectiveness of NGDs in catching inconsistencies in graphs.

Experimental setting. We used three real-life graphs: (a) DBpedia [3], a knowledge base with 28M entities of 200 types and 33.4M edges of 160 types; (b) YAGO2, an extended knowledge graph of YAGO [65] with 3.5M nodes of 13 types and 7.35M edges of 36 types; and (c) Pokec [1], a social network with 1.63M nodes of 269 types and 30.6M links of 11 types. We compared DBpedia and YAGO2 with Freebase [70], a more reliable knowledge base with 1.9B triples. If an edge (i.e., fact) occurs in both DBpedia (or YAGO2) and Freebase, we assigned weight 1 to it for certainty; otherwise, 0.5 was assigned. The edge weights for Pokec were assigned randomly. The density (defined as $\frac{|E|}{|V|\cdot(|V|-1)}$) is 6.5×10^{-7}, 6×10^{-7}, and 1.1×10^{-5}, and the average diameter of connected components is 4.8, 4.0, and 5.2, for DBpedia, YAGO2, and Pokec, respectively.

We also generated synthetic graphs $G = (V, E, L, F_A)$, where (a) node set V and edge set E were created by random graph generator and power-law graph generator of GTgraph [54], and each edge was given a random weight from $[0, 1]$; (b) labeling L was drawn from an alphabet $\mathcal{L}$ of 500 symbols; and (c) F_A assigned a set Γ_l of five active attributes for each node labeled l, and each attribute $A \in \Gamma_l$ drew its value from 1K values.

Three groups of synthetic graphs G were generated as follows: (a) We used random graph generator to create G with different $|V|$ and $|E|$, up to 80M and 100M, respectively, such that the density $\eta_G \le 10^{-6}$. (b) Using the same generator, we varied $|V|$ and $|E|$ while keeping $|V| + |E|$ in a fixed range to construct graphs with different η_G, ranging from 10^{-6} to 10^{-2}. (c) Fixing $|V|$ as 30M and $|E|$ as 60M, we set different distribution parameters for power-law graph generator to create synthetic graphs with power-law and uniform degree distribution, respectively.

NGDs. We discovered NGDs from 1/4 fraction of each graph using the method described in Section 3. These NGDs are strongly satisfiable. We picked a set Σ of 100 meaningful and diverse NGDs for each graph from the discovered ones, such that at least 90% of them have different patterns, including trees, stars, DAGs (directed acyclic graphs), and cyclic graphs. They carry patterns of diameters from 1 to 6, and 1 to 4 literals, with linear arithmetic expressions of lengths from 1 to 10.

ΔG. Updates ΔG to G were randomly generated, controlled by the size $|\Delta G|$ and a ratio γ of edge insertions to deletions. The ratio γ is 1 unless stated otherwise, i.e., the size $|G|$ remains unchanged.

Algorithms. In Java, we implemented (1) sequential algorithm IncDect (Section 6.2) vs. Dect, an error detection algorithm with NGDs, which is an extension of the algorithm developed for GFDs [33]; (2) parallel algorithm PIncDect (Section 6.3) vs. PDect, an extension of the parallel detection algorithm in Reference [33] to NGDs; (3) parallel $\text{PIncDect}_{\text{ns}}$, $\text{PIncDect}_{\text{nb}}$, and $\text{PIncDect}_{\text{NO}}$, variants of PIncDect with no work unit splitting, no workload balancing, and neither of the two, respectively, and (4) parallel PIncRDect vs. PRDect (Section 7), the PTIME parallel incremental and batch algorithms to detect top-ranked errors, and their sequential counterparts IncStarK and StarK, respectively.

We deployed the algorithms on a cluster of up to 20 machines, each with 32 GB DDR4 RAM and two 1.90 GHz Intel(R) Xeon(R) E5-2609 CPU, running 64-bit CentOS7 with Linux kernel 3.10.0. Each experiment was run five times and the average is reported here.

Experimental results. We next report our findings. The graphs were fragmented using METIS [2]. We took Synthetic G created by random graph generator with 30M nodes and 60M edges as default. When testing the efficiency of finding top-ranked errors, we applied a set of star-like NGDs that is selected from Σ and Σ_c and has the same cardinality as Σ. Here Σ_c is a set of 100 star-like NGDs discovered from each graph, in which their patterns have similar sizes to those in Σ. We fixed $K = 600$ for PTIME algorithms, the communication latency $C = 60$, time interval intvl $= 45s$, and the number of processors $p = 8$ for parallel algorithms unless stated otherwise.

Exp-1: Effectiveness of incremental error detection. We first evaluated the incremental algorithms against their batch counterparts. Fixing $\|\Sigma\| = 50$ and diameter $d_\Sigma = 5$, we varied the size $|\Delta G|$ of updates from 5% up to 40% of the size $|G|$ of graphs G in 5% increments. The results are reported in Figures 8(a)–8(d) over DBpedia, YAGO2, Pokec, and Synthetic, respectively (y-axis in logarithmic scale). From the experimental results, we find the following:

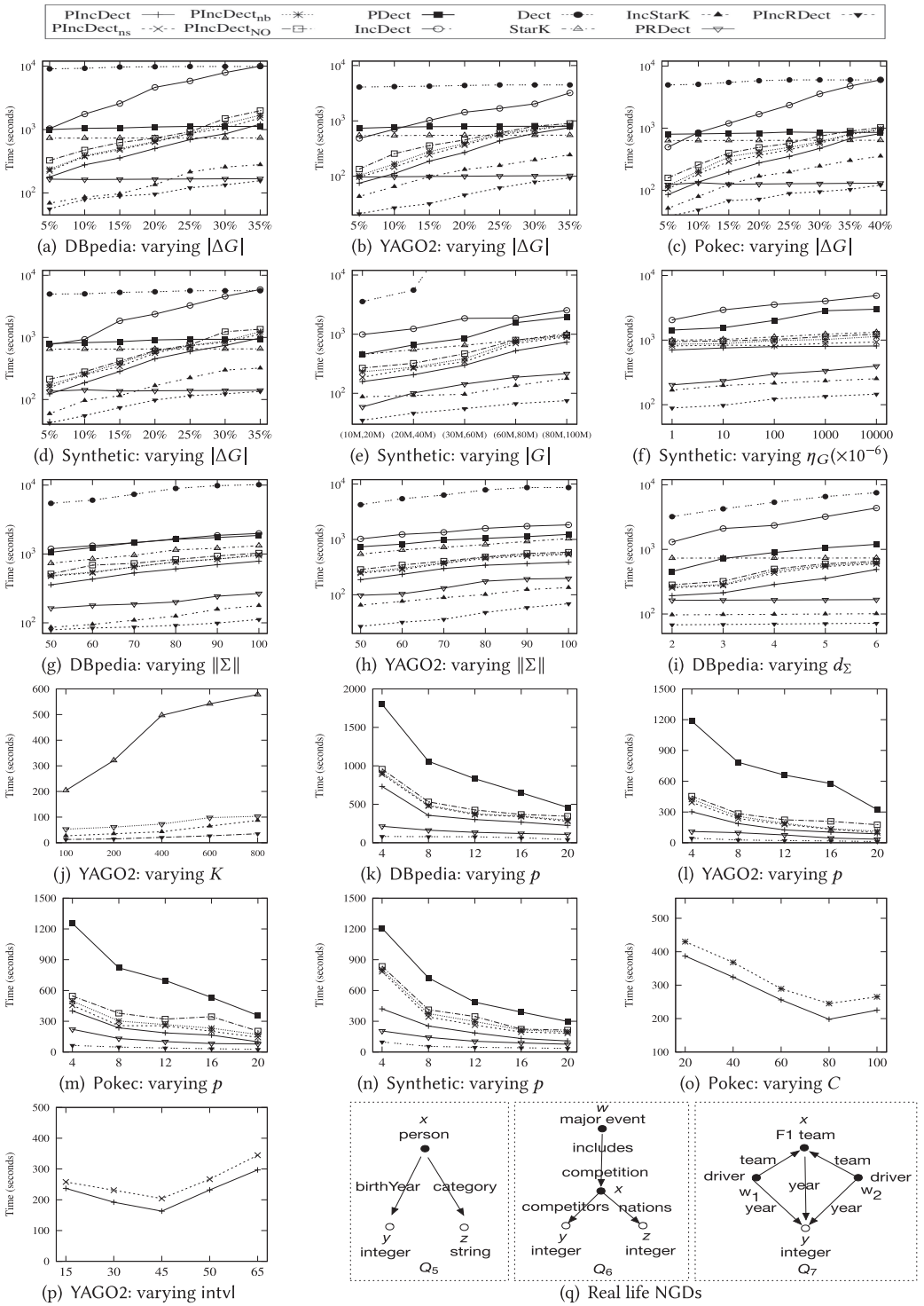
(a) When $|\Delta G|$ varies from 5% to 25% of $|G|$, IncDect is 8.8 to 1.7 (respectively, 8.5 to 2.6, 9.8 to 2.6, and 6.6 to 1.7) times faster than Dect over the four graphs; PIncDect outperforms PDect by 5.6 to 1.6 (respectively, 9.8 to 1.8, 9.4 to 2.5, and 5.6 to 1.6) times; and PIncRDect improves PRDect by 3.5 to 1.4 (respectively, 3.2 to 1.3, 4.6 to 1.6, and 3.2 to 1.2) times. Incremental PIncDect, IncDect, and PIncRDect beat their batch counterparts even when $|\Delta G|$ is 33% of $|G|$. These justify the need for incremental error detection.

(b) On average, PIncDect outperforms $\text{PIncDect}_{\text{ns}}$, $\text{PIncDect}_{\text{nb}}$, and $\text{PIncDect}_{\text{NO}}$ by 1.29, 1.33, and 1.61 times on DBpedia (respectively, 1.31, 1.43, 1.81 on YAGO2, 1.33, 1.45, 1.81 on Pokec, and 1.27, 1.36, 1.5 on Synthetic G) in the same setting. This verifies the effectiveness of our hybrid workload balancing strategy. It suggests that workload balancing should be combined with work unit splitting.

(c) The larger $|\Delta G|$ is, the slower all incremental algorithms are, while the batch Dect, PDect, StarK, and PRDect are indifferent to $|\Delta G|$, as expected. In all cases, PIncRDect performs the best.

(d) Incremental error detection is feasible in practice: PIncDect takes 693 s on DBpedia when $|\Delta G|$ is 25% of $|G|$, IncDect takes 5,840 s, PIncRDect takes 121 s, and IncStarK needs 214 s, as opposed to 1,121 s (respectively, 9,878 s, 166 s, 736 s) by PDect (respectively, Dect, PRDect, StarK).

(e) All incremental algorithms are insensitive to the ratio γ of edge insertions to deletions, which is verified by varying the ratio γ (results not shown).



(a) DBpedia: varying $|\Delta G|$
(b) YAGO2: varying $|\Delta G|$
(c) Pokec: varying $|\Delta G|$
(d) Synthetic: varying $|\Delta G|$
(e) Synthetic: varying $|G|$
(f) Synthetic: varying $\eta_G(\times 10^{-6})$
(g) DBpedia: varying $\|\Sigma\|$
(h) YAGO2: varying $\|\Sigma\|$
(i) DBpedia: varying d_Σ
(j) YAGO2: varying K
(k) DBpedia: varying p
(l) YAGO2: varying p
(m) Pokec: varying p
(n) Synthetic: varying p
(o) Pokec: varying C
(p) YAGO2: varying intvl
(q) Real life NGDs

Fig. 8. Experimental results.

(f) We find that the cost of checking linear arithmetic expressions is negligible (not shown). This confirms Corollary 5.1, i.e., NGDs do not make the validation problem harder than GFDs.

Exp-2: Diversity of graphs. We evaluated the impact of the size, density, and degree distribution of the graphs using Synthetic G. We fixed $|\Delta G|$ as 15% of $|G|$ and used the same NGDs as in Exp-1.

Varying $|G|$. We varied $|G|$, i.e., $(|V|, |E|)$, from (10M, 20M) to (80M, 100M). Figure 8(e) shows that (a) all the algorithms take longer when graph G gets larger, as expected, (b) incremental algorithms are less sensitive to the size $|G|$ than their batch counterparts, and (c) PIncRDect does the best among all, which is consistent with the results of Exp-1.

Varying η_G. Fixing $|V| + |E|$ in the range of $[90M, 110M]$, we varied the density η_G from 10^{-6} to 10^{-2}. Figure 8(f) tells us that (a) all algorithms take longer when graph G gets denser, as expected, and (b) incremental error detection is feasible on dense graphs, e.g., PIncDect and PIncRDect take 812 s and 146 s when $\eta_G = 10^{-2}$, respectively.

Degree distribution. Fixing $|G|$ as (30M, 60M), we tested all the algorithms on two synthetic graphs having pow-law and uniform degree distribution, respectively. We find that the difference between the response time of each algorithm on these two graphs is at most 13.5% of the slower one, and the difference for incremental algorithms is even smaller (not shown), i.e., the degree distribution of the graphs has little impact on the performance of incremental detection of inconsistencies.

Exp-3: Complexity of NGDs. We also evaluated the impact of the complexity of sets Σ of NGDs. We fixed $|\Delta G| = 15\%|G|$ in this set of experiments.

Varying $\|\Sigma\|$. Fixing $d_\Sigma = 5$, we varied the cardinality $\|\Sigma\|$ from 50 to 100 (our industry collaborator uses at most 95 rules [12]). As shown in Figures 8(g) and 8(h) on DBpedia and YAGO2, respectively, (a) the more NGDs are in Σ, the longer time is taken by all the algorithms, as expected, and (b) PIncDect, IncDect, PIncRDect, and IncStarK scale well with $\|\Sigma\|$.

The results on Pokec and Synthetic are consistent (not shown).

Varying d_Σ. Fixing $\|\Sigma\| = 50$, we varied d_Σ from 2 to 6. Figure 8(i) shows that all algorithms take longer over larger d_Σ on DBpedia except those that find top-ranked errors. This is consistent with our analysis that the costs of our localizable incremental algorithms increase when d_Σ gets larger. Nonetheless, PIncDect is feasible with real-life NGDs, e.g., it takes 489 s on DBpedia when $d_\Sigma = 6$, as opposed to 1,197 s by PDect and 7,532 s by Dect. PIncRDect, PRDect, StarK, and IncStarK are almost indifferent to d_Σ, since they only use the star-like NGDs, whose patterns are of diameters 1 or 2. The results on YAGO2, Pokec, and Synthetic are consistent.

Exp-4: Impact of K. This set of experiments evaluated the impact of parameter K on the algorithms for finding top-ranked inconsistencies. Fixing $|\Delta G| = 15\%|G|$, Figure 8(j) reports the result over YAGO2 using the NGDs of Exp-1. We can see that (a) all the algorithms take longer time for larger K, since more matches are required to be identified and checked; nonetheless, they are efficient in practice, e.g., PRDect and PIncRDect take 103 s and 35 s when $K = 800$, respectively. (b) PIncRDect (respectively, IncStarK) improves the performance of PRDect (respectively, StarK) by 3.6 (respectively, 8.6) times, on average. The results on DBpedia, Pokec, and Synthetic are consistent and hence not shown.

Exp-5: Scalability of parallel algorithms. Using the same NGDs as in Exp-1 and fixing $|\Delta G| = 15\%|G|$ for all the graphs, we evaluated the scalability of parallel algorithms (a) PIncDect versus PDect, $\text{PIncDect}_{\text{ns}}$, $\text{PIncDect}_{\text{nb}}$, and $\text{PIncDect}_{\text{NO}}$, and (b) PIncRDect versus PRDect by varying the number p of processors, the communication latency C, and the time interval intvl.

Varying p. Fixing $C = 60$ and intvl = 45s, we varied p from 4 to 20. As shown in Figures 8(k), 8(l), 8(m), and 8(n) over DBpedia, YAGO2, Pokec, and Synthetic, respectively, when p changes from 4 to 20, (a) parallel algorithms PIncDect, PDect, PIncRDect, and PRDect perform much better and are on average 3.7, 3.8, 2.7, and 2.6 times faster than IncDect, Dect, IncStarK, and StarK, respectively, and (b) PIncDect consistently outperforms PDect, $\text{PIncDect}_{\text{ns}}$, $\text{PIncDect}_{\text{nb}}$, and $\text{PIncDect}_{\text{NO}}$: On average, it is 2.47 to 3.14, 1.32 to 1.37, 1.44 to 1.53, and 1.53 to 1.72 times better, respectively, validating the effectiveness of the hybrid workload partition strategy. Moreover, work unit splitting or workload balancing alone does not work very well, as verified by the gap between the performance of PIncDect and that of $\text{PIncDect}_{\text{nb}}$ and $\text{PIncDect}_{\text{ns}}$, respectively.

Varying C. Fixing $p = 8$ and intvl = 45s, we evaluated the impact of communication latency parameter on PIncDect and $\text{PIncDect}_{\text{nb}}$ by tuning C from 20 to 100 in 20 increments. As shown in Figure 8(o) over Pokec, PIncDect performs the best when C is 80, taking 198 s. On one hand, PIncDect favors parallel computation with smaller C to split work units; on the other hand, PIncDect has a bias towards local computation with larger latency C to reduce the communication cost. The results on DBpedia, YAGO2, and Synthetic are consistent and hence are not shown.

Varying intvl. Fixing $p = 8$ and $C = 60$, we varied intvl from 15 s to 65 s in 15 s increments to evaluate the impact of intervals for monitoring workloads on PIncDect and $\text{PIncDect}_{\text{ns}}$. As shown in Figure 8(p) on YAGO2, the "optimal" intvl is 45 s for PIncDect. Similar to the latency C, while smaller intvl helps workload balancing, it incurs more communication cost. Hence, we need to strike a balance. The results on DBpedia, Pokec, and Synthetic G are consistent.

Exp-6: Effectiveness study. We manually examined the NGDs Σ discovered from real-life graphs to ensure that our picked ones are correct. The NGDs captured 415, 212, and 568 errors in DBpedia, YAGO2, and Pokec, respectively, ranging from wrong (numeric) values to structural errors. Note that errors were found in the rest 3/4 of each graph, which was not used for NGD discovery.

Real-world NGDs. Below are the NGDs shown in Figure 8(q), along with the real errors they caught.

NGD_1 is $Q_5[\bar{x}](y.\text{val} < 1800 \rightarrow z.\text{val} \neq \text{"living people"})$, stating that any person with birth year before 1800, i.e., aged over 210, can no longer be categorized as living people. It identifies an error in DBpedia that a living person John Macpherson was born in 1713.

NGD_2 is $Q_6[\bar{x}](w.\text{type} = \text{"Olympic"} \rightarrow z.\text{val} \leq y.\text{val})$, which states that the number of participating nations in an Olympic event should not be larger than the number of competitors, i.e., each athlete represents at most one nation. It detects that 24 athletes representing 34 countries participated in the Women's Sailboard Competition at the 1992 Summer Olympics, in DBpedia.

NGD_3 is $Q_7[\bar{x}](\emptyset \rightarrow x.\text{numberOfWins} \geq w_1.\text{numberOfWins} + w_2.\text{numberOfWins})$. This NGD states that in the Formula One racing, the total number of competitions won by two drivers is no larger than the number of competitions won by the team they represent during the same year. In DBpedia, it caught that Sebastian Vettel and Max Verstappen won one competition in 2016; however, their team Scuderia Ferrari won none of the races. In fact, Max did not race for Ferrari in 2016. This shows that NGDs also help us detect the erroneous links.

Error detection accuracy. We next evaluated the accuracy for error detection with NGDs on YAGO2. Besides the errors detected by our picked set Σ of NGDs, we also injected noise to YAGO2 by sampling $\alpha\%$ of nodes from the match candidates of the patterns from Σ in this graph and changing $\beta\%$ of either the attribute values associated with v or the labels of the edges incident to v for the sampled nodes v. We took care to ensure that our changes involve the attributes that appear in the attribute dependencies $X \rightarrow Y$ of the NGDs in Σ.

(α%, β%)	NGDs	GFDs	star-like NGDs
(5%, 10%)	84.6%	71.5%	54.1%
(10%, 20%)	78.7%	68.4%	50.4%
(20%, 40%)	70.3%	61.5%	41.8%
(30%, 60%)	61.5%	50.1%	33.7%

Fig. 9. Error detection accuracy.

The *accuracy* of error detection with NGDs is defined as $\frac{|V^{\text{NGD}} \cap V^e|}{|V^e|}$, where (a) V^e denotes the set of nodes that are involved in either the original errors captured or the noise introduced; and (b) V^{NGD} refers to all the nodes that are contained in the violations of NGDs after error injection. The accuracy for error detection with GFDs and star-like NGDs is similarly defined. It should be remarked that we adopted recall here to define the accuracy, since the precision is always 1 as long as all the NGDs are correct and are enforced on the graphs.

With various α% and β%, Figure 9 reports the accuracy of the NGDs, GFDs, and star-like NGDs in the discovered set Σ of cardinality 100. As shown there, (1) NGDs have the best accuracy in all cases. (2) All the graph dependencies perform better with smaller α% and β%, as expected. (3) The star-like NGDs and GFDs capture on average 60.5% and 80.5%, respectively, of the errors caught by NGDs.

Summary. We find the following: (1) Our incremental error detection algorithms scale well with $|\Delta G|$, $|G|$, η_G, $\|\Sigma\|$, d_Σ, and K. Algorithms IncDect and PIncDect outperform batch Dect from 6.7 to 2.1 times and from 52 to 13 times on average, respectively, when $|\Delta G|$ varies from 5% to 25% of $|G|$ over real-life and synthetic graphs. They perform better even when $|\Delta G|$ is up to 33% of $|G|$. (2) The incremental algorithms are less sensitive to $|G|$ than the batch algorithms, and they are able to deal with large-scale dense graphs. (3) Better still, parallel PIncDect scales well with the number p of processors used: Its runtime is improved by 3.7 times on average when p increases from 4 to 20. (4) Algorithms IncDect and PIncDect are feasible in practice: On real-life graphs, they take 1,659 s and 130 s on average (with $p = 20$), respectively. (5) The hybrid workload balancing strategy is effective: It helps improve the performance of PIncDect by 1.73 times on average and works well with large p. (6) Incremental detection of top-ranked errors is efficient: PIncRDect takes at most 30 s on average over real-life graphs when $K = 600$ and $p = 8$. Moreover, it catches top-ranked errors, which account for 60.5% of total errors, striking a balance between the accuracy and efficiency.

9 CONCLUSION

We have proposed a class of NGDs with linear arithmetic expressions and comparison predicates to catch semantic inconsistencies in graphs, numeric or not. We have justified NGDs by establishing the complexity of the satisfiability and implication analyses of NGDs and their extensions. We have also provided the (parameterized) complexity of validation and incremental validation problems for NGDs. We have developed the first parallel incremental algorithms to detect errors in graphs with provable performance guarantees, as well as parallel PTIME algorithms for detection and incremental detection of top-ranked inconsistencies. We have empirically verified that NGDs and the algorithms yield a promising tool for detecting errors in graph-structured data, *numeric or not*.

There is naturally much more to be done. One topic for future work is to extend NGDs by supporting aggregations. Another topic is to study graph repairing with NGDs.

REFERENCES

[1] Lubos Takac and Michal Zabovsky. 2012. Pokec Social Network. Retrieved from http://snap.stanford.edu/data/soc-pokec.html.

[2] George Karypis and Vipin Kumar. 2013. Metis. Retrieved from http://glaros.dtc.umn.edu/gkhome.

[3] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary G. Ives. 2015. DBpedia. Retrieved from http://wiki.dbpedia.org/Datasets.

[4] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases*. Addison-Wesley.

[5] Karl A. Abrahamson, Rodney G. Downey, and Michael R. Fellows. 1995. Fixed-parameter tractability and completeness IV: On completeness for W[P] and PSPACE analogues. *Ann. Pure Appl. Logic* 73, 3 (1995), 235–276.

[6] Foto N. Afrati, Chen Li, and Prasenjit Mitra. 2002. Answering queries using views with arithmetic comparisons. In *Proceedings of the PODS*.

[7] Foto N. Afrati, Chen Li, and Prasenjit Mitra. 2006. Rewriting queries using views in the presence of arithmetic comparisons. *Theor. Comput. Sci.* 368, 1–2 (2006), 88–123.

[8] Foto N. Afrati, Chen Li, and Vassia Pavlaki. 2008. Data exchange in the presence of arithmetic comparisons. In *Proceedings of the EDBT*.

[9] Waseem Akhtar, Alvaro Cortés-Calabuig, and Jan Paredaens. 2010. Constraints in RDF. In *Proceedings of the SDKB*.

[10] Konstantin Andreev and Harald Racke. 2006. Balanced graph partitioning. *Theor. Comput. Syst.* 39, 6 (2006), 929–939.

[11] Abdallah Arioua and Angela Bonifati. 2018. User-guided repairing of inconsistent knowledge bases. In *Proceedings of the EDBT*.

[12] Baidu. 2017. Personal Communication. http://www.baidu.com.

[13] Manuel Blum, Robert W. Floyd, Vaughan R. Pratt, Ronald L. Rivest, and Robert Endre Tarjan. 1973. Time bounds for selection. *J. Comput. Syst. Sci.* 7, 4 (1973), 448–461.

[14] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling multithreaded computations by work stealing. *J. ACM* 46, 5 (1999), 720–748.

[15] Paul Burkhardt and Chris Waring. 2013. *An NSA Big Graph Experiment*. Technical Report NSA-RD-2013-056002v1. U.S. National Security Agency.

[16] Marco Cesati. 2006. *Compendium of Parameterized Problems*. Technical report. Department of Computer Science, Systems, and Industrial Engineering, University of Rome Tor Vergata.

[17] Yang Chen, Sean Louis Goldberg, Daisy Zhe Wang, and Soumitra Siddharth Johri. 2016. Ontological pathfinding. In *Proceedings of the SIGMOD*.

[18] Jiefeng Cheng, Xianggang Zeng, and Jeffrey Xu Yu. 2013. Top-k graph pattern matching over large graphs. In *Proceedings of the ICDE*.

[19] Yiu-ming Cheung and Hong Jia. 2012. Unsupervised feature selection with feature clustering. In *Proceedings of the WI*.

[20] William Cook, Albertus M. H. Gerards, Alexander Schrijver, and Eva Tardos. 1986. Sensitivity theorems in integer linear programming. *Math. Prog.* 34, 3 (1986), 251–264.

[21] Alvaro Cortés-Calabuig and Jan Paredaens. 2012. Semantics of constraints in RDFS. In *Proceedings of the AMW*.

[22] Rod G. Downey and Michael R. Fellows. 1995. Fixed-parameter tractability and completeness II: On completeness for W[1]. *Theor. Comput. Sci.* 141, 1–2 (1995), 109–131.

[23] Grace Fan, Wenfei Fan, and Floris Geerts. 2014. Detecting errors in numeric attributes. In *Proceedings of the WAIM*.

[24] Wenfei Fan, Zhe Fan, Chao Tian, and Xin Luna Dong. 2015. Keys for graphs. *PVLDB* 8, 12 (2015), 1590–1601.

[25] Wenfei Fan and Floris Geerts. 2012. *Foundations of Data Quality Management*. Morgan & Claypool Publishers.

[26] Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. 2008. Conditional functional dependencies for capturing data inconsistencies. *ACM Trans. Datab. Syst.* 33, 2 (2008), 6:1–6:48.

[27] Wenfei Fan, Chunming Hu, Xueli Liu, and Ping Lu. 2018. Discovering graph functional dependencies. In *Proceedings of the SIGMOD*.

[28] Wenfei Fan, Chunming Hu, and Chao Tian. 2017. Incremental graph computations: Doable and undoable. In *Proceedings of the SIGMOD*.

[29] Wenfei Fan, Jianzhong Li, Nan Tang, and Wenyuan Yu. 2012. Incremental detection of inconsistencies in distributed data. In *Proceedings of the ICDE*.

[30] Wenfei Fan, Xueli Liu, Ping Lu, and Chao Tian. 2018. Catching numeric inconsistencies in graphs. In *Proceedings of the SIGMOD*.

[31] Wenfei Fan and Ping Lu. 2017. Dependencies for graphs. In *Proceedings of the PODS*.

[32] Wenfei Fan, Xin Wang, Yinghui Wu, and Jingbo Xu. 2015. Association rules with graph patterns. *Proc. VLDB Endow.* 8, 12 (2015), 1502–1513.

[33] Wenfei Fan, Yinghui Wu, and Jingbo Xu. 2016. Functional dependencies for graphs. In *Proceedings of the SIGMOD*.

[34] Sergio Flesca, Filippo Furfaro, and Francesco Parisi. 2010. Querying and repairing inconsistent numerical databases. *ACM Trans. Datab. Syst.* 35, 2 (2010), 14:1–14:50.

[35] Jörg Flum and Martin Grohe. 2006. *Parameterized Complexity Theory*. Springer.
[36] Fedor V. Fomin, Pinar Heggernes, and Dieter Kratsch. 2007. Exact algorithms for graph homomorphisms. *Theor. Comput. Sci.* 41, 2 (2007), 381–393.
[37] Enrico Franconi, Antonio Laureti Palma, Nicola Leone, Simona Perri, and Francesco Scarcello. 2001. Census data repair: A challenging application of disjunctive logic programming. In *Proceedings of the LPAR*.
[38] Luis Antonio Galárraga, Christina Teflioudi, Katja Hose, and Fabian Suchanek. 2013. AMIE: Association rule mining under incomplete evidence in ontological knowledge bases. In *Proceedings of the WWW*.
[39] Mario Arias Gallego, Javier D. Fernández, Miguel A. Martínez-Prieto, and Pablo de la Fuente. 2011. An empirical study of real-world SPARQL queries. In *Proceedings of the USEWOD Workshop*.
[40] Lukasz Golab, Howard J. Karloff, Flip Korn, Avishek Saha, and Divesh Srivastava. 2009. Sequential dependencies. *Proc. VLDB Endow.* 21, 1 (2009), 574–585.
[41] Ivana Grujic, Sanja Bogdanovic-Dinic, and Leonid Stoimenov. 2014. Collecting and analyzing data from E-government Facebook pages. In *Proceedings of the ICT Innovations*.
[42] Aaron Harlap, Henggang Cui, Wei Dai, Jinliang Wei, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. 2016. Addressing the straggler problem for iterative convergent parallel ML. In *Proceedings of the SoCC*.
[43] Binbin He, Lei Zou, and Dongyan Zhao. 2014. Using conditional functional dependency to discover abnormal data in RDF graphs. In *Proceedings of the SWIM*.
[44] Jiewen Huang, Daniel J. Abadi, and Kun Ren. 2011. Scalable SPARQL querying of large RDF graphs. *Proc. VLDB Endow.* 4, 11 (2011), 1123–1134.
[45] James P. Jones. 1980. Undecidable Diophantine equations. *Bull. Amer. Math. Soc.* 3, 2 (1980).
[46] Dmitri V. Kalashnikov, Laks V. S. Lakshmanan, and Divesh Srivastava. 2018. FastQRE: Fast query reverse engineering. In *Proceedings of the SIGMOD*.
[47] Mijung Kim and K. Selçuk Candan. 2012. SBV-Cut: Vertex-cut based graph partitioning using structural balance vertices. *Data Knowl. Eng.* 72 (2012), 285–303.
[48] Dimitris Kontokostas, Patrick Westphal, Sören Auer, Sebastian Hellmann, Jens Lehmann, Roland Cornelissen, and Amrapali Zaveri. 2014. Test-driven evaluation of linked data quality. In *Proceedings of the WWW*.
[49] Nick Koudas, Avishek Saha, Divesh Srivastava, and Suresh Venkatasubramanian. 2009. Metric functional dependencies. In *Proceedings of the ICDE*.
[50] Clyde P. Kruskal, Larry Rudolph, and Marc Snir. 1990. A complexity theory of efficient parallel algorithms. *Theor. Comput. Sci.* 71, 1 (1990), 95–132.
[51] Longbin Lai, Lu Qin, Xuemin Lin, and Lijun Chang. 2015. Scalable subgraph enumeration in MapReduce. *Proc. VLDB Endow.* 8, 10 (2015), 974–985.
[52] Georg Lausen, Michael Meier, and Michael Schmidt. 2008. SPARQLing constraints for RDF. In *Proceedings of the EDBT*.
[53] Jinsoo Lee, Wook-Shin Han, Romans Kasperovics, and Jeong-Hoon Lee. 2012. An In-depth comparison of subgraph isomorphism algorithms in graph databases. *Proc. VLDB Endow.* 6, 2 (2012), 133–144.
[54] Kamesh Madduri and David A. Bader. 2006. GTgraph. Retrieved from http://www.cse.psu.edu/ kxm85/software/GTgraph/.
[55] Yuri Matiyasevich. 1993. *Hilbert's 10th Problem*. The MIT Press.
[56] Douglas C. Montgomery, Elizabeth A. Peck, and G. Geoffrey Vining. 2012. *Introduction to Linear Regression Analysis*. John Wiley & Sons.
[57] Amelia Murray. 2016. Fake NatWest Twitter account targets customers to steal bank details. Retrieved from http://www.telegraph.co.uk/money/consumer-affairs/fake-natwest-twitter-account-targets-customers-to-steal-bank-det.
[58] Alexandros Ntoulas, Junghoo Cho, and Christopher Olston. 2004. What's new on the Web? The evolution of the Web from a search engine perspective. In *Proceedings of the WWW*.
[59] Christos H. Papadimitriou. 1994. *Computational Complexity*. Addison-Wesley.
[60] Nataliya Prokoshyna, Jaroslaw Szlichta, Fei Chiang, Renée J. Miller, and Divesh Srivastava. 2015. Combining quantitative and logical data cleaning. *Proc. VLDB Endow.* 9, 4 (2015), 300–311.
[61] Paweł Rzążewski. 2014. Exact algorithm for graph homomorphism and locally injective graph homomorphism. *Inf. Proc. Lett.* 114, 7 (2014), 387–391.
[62] Marcus Schaefer and Christopher Umans. 2002. Completeness in the polynomial-time hierarchy: A compendium. *SIGACT News* 33, 3 (2002), 32–49.
[63] Shaoxu Song and Lei Chen. 2011. Differential dependencies: Reasoning and discovery. *ACM Trans. Datab. Syst.* 36, 3 (2011), 16:1–16:41.
[64] Shaoxu Song, Hong Cheng, Jeffrey Xu Yu, and Lei Chen. 2014. Repairing vertex labels under neighborhood constraints. *Proc. VLDB Endow.* 7, 11 (2014), 987–998.
[65] Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. 2007. Yago: A core of semantic knowledge. In *Proceedings of the WWW*.

[66] Fabian M. Suchanek, Mauro Sozio, and Gerhard Weikum. 2009. SOFIE: A self-organizing framework for information extraction. In *Proceedings of the WWW*.
[67] Zhao Sun, Hongzhi Wang, Haixun Wang, Bin Shao, and Jianzhong Li. 2012. Efficient subgraph matching on billion node graphs. *Proc. VLDB Endow.* 5, 9 (2012), 788–799.
[68] Maksims Volkovs, Fei Chiang, Jaroslaw Szlichta, and Renée J. Miller. 2014. Continuous data cleaning. In *Proceedings of the ICDE*.
[69] Dominik Wienand and Heiko Paulheim. 2014. Detecting incorrect numerical data in DBpedia. In *Proceedings of the ESWC*.
[70] Wikipedia. 2019. Freebase. Retrieved from https://en.wikipedia.org/wiki/Freebase.
[71] Shengqi Yang, Fangqiu Han, Yinghui Wu, and Xifeng Yan. 2016. Fast top-k search in knowledge graphs. In *Proceedings of the ICDE*.
[72] Zhengwei Yang, Ada Wai-Chee Fu, and Ruifeng Liu. 2016. Diversified top-k subgraph querying in a large graph. In *Proceedings of the SIGMOD*.
[73] Yang Yu and Jeff Heflin. 2011. Extending functional dependency to detect abnormal data in RDF graphs. In *Proceedings of the ISWC*.

Received February 2019; revised September 2019; accepted February 2020