2021 IEEE 37th International Conference on Data Engineering (ICDE)

# Explaining Missing Data in Graphs: A Constraint-based Approach

Qi Song*
*Amazon.com*
qison@amazon.com

Peng Lin
*Washington State University*
peng.lin@wsu.edu

Hanchao Ma
*Case Western Reserve University*
hxm382@case.edu

Yinghui Wu
*Case Western Reserve University*
*Pacific Northwest National Laboratory*
yxw1650@case.edu

***Abstract*—This paper introduces a *constraint-based approach* to clarify missing values in graphs. Our method capitalizes on a set $\Sigma$ of graph data constraints. An explanation is a sequence of operational enforcement of $\Sigma$ towards the recovery of interested yet missing data (*e.g.,* attribute values, edges). We show that constraint-based approach helps us to understand not only why a value is missing, but also how to recover the missing value. We study $\Sigma$-*explanation* problem, which is to compute the optimal explanations with guarantees on the informativeness and conciseness. We show the problem is in $\Delta_2^P$ for established graph data constraints such as graph keys and graph association rules. We develop an efficient bidirectional algorithm to compute optimal explanations, without enforcing $\Sigma$ on the entire graph. We also show our algorithm can be easily extended to support graph refinement within limited time, and to explain missing answers. Using real-world graphs, we experimentally verify the effectiveness and efficiency of our algorithms.**

***Index Terms*—Graphs, Data Constraints, Data Provenance**

## I. Introduction

Real-world graphs are incomplete [1]: attribute values of entities (nodes) and relations (edges) are often missing. Enhancing graphs from multiple data sources with entity matching and link prediction has been widely studied [1]. A desirable yet missing functionality is to clarify *why* certain expected data is missing in graph data, whether such data can be restored, and how. Such need is evident in knowledge fusion [2], user-centric data quality [3], and query suggestion [4].

Missing data in a graph $G$ can be captured by *data constraints for graphs* [5]–[9]. These data constraints identify node pairs $(v, v')$ in $G$ via *graph pattern matching* between a graph pattern $P$ and a graph $G$, and either enforce node equivalence or assert a missing edge between $v$ and $v'$. Consider the following established data constraints for graphs.

(1) *Key constraints* [6], [10], [11] have a general form of $P \rightarrow (u.\mathsf{id} = u'.\mathsf{id})$, and state that "*A pair of nodes $v$ and $v'$ in a graph $G$ should refer to the the real-world entity* (id)*, if they both match a graph pattern $P$ via graph pattern matching*".

For example, graph keys [6] state that "any pair of nodes in a graph that match to a designated pattern node in a graph pattern $P$ should refer to the same entity", where $P$ is defined by variants of subgraph isomorphism [6], [9], [11].

(2) Constraints that infer missing edges $P \rightarrow r(u, u')$ [5], [12]–[14] state that "*There is an edge between a pair of nodes $v$ and $v'$ in $G$ with label $r$, if $v$ and $v'$ both match $P$ via graph pattern matching*". Notable examples include AMIE [5] (where $P$ are Horn clauses), conditional patterns [14], and graph association rules [12], [13] where the matching semantics of $P$ are defined by variants of subgraph isomorphism.

*Can these data constraints be used to explain why specific data of interests is missing in a graph*? Naturally, data constraints may provide useful provenance information by inferring new data [15]. Consider the following example.

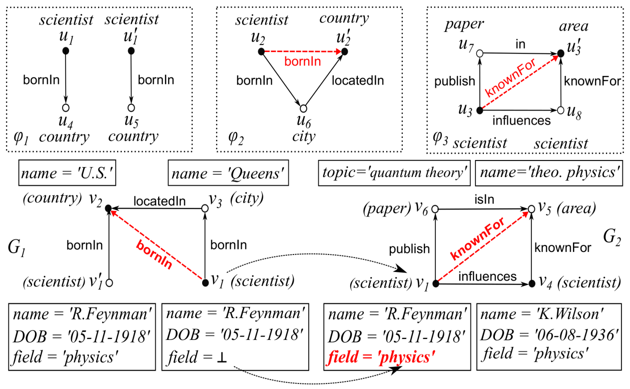


Fig. 1: Clarifying "*why the statement 'R.Feynman is known for physics' is missing?*" with graph data constraints.

**Example 1:** A fraction of a curated academic knowledge graph $G$ is illustrated in Fig 1 (excluding dashed edges and colored attributes). Each node (*e.g.,* $v_1$) refers to an entity with a label (*e.g.,* a type scientist), and a tuple that maps attributes to their values (*e.g.,*name = 'R.Feynman'). Each edge encodes a relation between two nodes, *e.g.,* 'R.Feynman' was born in (bornIn) 'U.S.'. For the ease of discussion, we decompose $G$ into components $G_1$ and $G_2$ with a shared node $v_1$.

There are two validated graph data constraints defined on $G$: a graph key $\varphi_1$ with graph pattern $P_1$, and a graph association rule $\varphi_2$ with pattern $P_2$ (Fig. 1), which state the following.

$\underline{\varphi_1}$ [6]: "*two **scientists** refer to the same person* ($u_1$.id = $u_1'$.id)*, if they have the same name, birth date, and were born in* (bornIn) *the same country*"($u_4$.val = $u_5$.val).

$\underline{\varphi_2}$ [12], [13]: "*a scientist* ($u_2$) *was born in a country* ($u_2'$) *if the scientist was born in a city* ($u_6$) *which is located in* (locatedIn) *that country* ($u_2'$)."

*This job was done prior to joining Amazon.

A missing edge $\mathsf{bornIn}(v_1, v_2)$ can be inferred by enforcing $\varphi_2$ in $G_1$. Indeed, $v_1$ and $v_2$ match the variables $u_2$ and $u'_2$ in $P_2$ via subgraph isomorphism, but *violate* $\varphi_2$: there is no relation bornIn between them. This verifies the missing birth country 'US' of $v_1$ given his birth city 'Queens'. □

Nevertheless, simply enforcing all the constraints in a static "batch" may fail to explain why particular data is missing.

**Example 2:** Consider a third graph association rule $\varphi_3$ with pattern $P_3$ (Fig. 1) with the following semantic:

> $\varphi_3$: *"a scientist ($u_3$) is **known for** an area ($u'_3$), if he published a paper ($u_7$) in the same area, and influenced a peer of his field ($u_3$.field = $u_8$.field) from that area."*

A user issues a (SPARQL) query $Q$ "SELECT ?scientist WHERE ?scientist KnownFor 'theo.physics' " to search for "*scientists who are known for theoretical physics*". As the answer $Q(G)$ only contains 'K.Wilson', the user is wondering why R.Feynman, a famous theoretical physicist in $G$ is not in $Q(G)$. It is desirable to clarify whether and how refining $G$ with new data can introduce 'R.Feynman' to $Q(G)$.

Observe that a missing edge KnownFor($v_1$ ('R. Feynman'), $v_5$('theo.physics')) could be inferred by enforcing $\varphi_3$, and "enables" R.Feynman to be an answer of $Q$ if inserted. Nonetheless, $\varphi_3$ is already satisfied by $G$: it is unknown whether $v_1$ ('R. Feynman') has the same field as $v_4$ ('K. Wilson') due to the unknown 'field' (marked as '$\perp$'). Thus $v_1$ fails to match $u_3$ on "influence in physics". In fact, no violation can be captured by $\varphi_3$ in $G$ as is.

On the other hand, the missing edge KnownFor($v_1$ ('R. Feynman'), $v_5$('theo.physics')) can be clarified by interleaving enforcement in a sequence that "evolves" $G$:

- Enforcing $\varphi_2$ explains an edge $\mathsf{bornIn}(v_1, v_2)$ is missing, and infers $v_1$'s birth country given his birth city.
- Inserting the missing edge $\mathsf{bornIn}(v_1, v_2)$ enables the enforcement of $\varphi_1$, which clarifies why $v_1$ and $v'_1$ *should* refer to the same scientist (given the same name, country and birthdate). It also infers a value 'physics' of $v_1$.field.
- The enriched node $v_1$ allows $\varphi_3$ to be enforced, as the 'influence' between physicists $v_1$ and $v_4$ are identified.

An explanation for "why" $\mathsf{knownFor}(v_1, v_5)$ is missing can be characterized by a sequence of "modifications" of $G$ due to the enforcement of corresponding constraints $\{\varphi_2, \varphi_1, \varphi_3\}$. It also clarifies "how" to recover $\mathsf{knownFor}(v_1, v_5)$. This helps users to track and diagnose the refinement of graphs. □

Although desirable, clarifying *specific missing values* is nontrivial, due to the high computational complexity of enforcing the graph data constraints and the dynamic enforcement when $G$ is large. These call for efficient solutions that are optimized to clarify specific missing values.

**Contribution & Organization**. We propose a *constraint-based approach* to clarify missing data in graphs, and study the problem complexity, measurements and feasible algorithms.

*(1) Formal Characterization* (Section III). Given a graph $G$ and data constraints $\Sigma$ of $G$, we introduce $\Sigma$-*explainability* to characterize whether a designated missing value can be derived via sequential enforcement of $\Sigma$ ("an explanation"). These sequences ensure more informative graphs, and eventually all terminate at a unique graph up to graph homomorphic equivalence (Section III-B). As such, they clarify missing data with responsible constraints ("*Why*") and operators ("*how*").

*(2) Quality measures* (Section III-C). We quantify the quality of constraint-based explanations with two measures: informativeness gain (amount of new values introduced to $G$), and conciseness. Based on the measures, we study $\Sigma$-*explanation* problem to compute concise and informative explanations for specified missing data of interests. We show that deciding $\Sigma$-explanability is complement to the validation problem of $\Sigma$, and is nontrivial (*e.g.,* $\Delta_2^P$ for graph keys). We also show that it is doable given an oracle that efficiently detects violations.

*(3) Bidirectional Search* (Section IV). We develop an algorithm BiExp for $\Sigma$-explanation. The algorithm uses bidirectional search to explore applicable actions (forward search) and virtual "enablers" (backward search) requested in order to derive missing data. We develop pruning techniques to reduce unnecessary exploration of enforcement.

*(4) Extensions* (Section V). We also extend BiExp for two applications. The first variant clarifies missing values in query results for subgraph queries. It incorporates query processing into a bidirectional search of BiExp and has the same worst-case time cost. The second, in support of online graph refinement, can progressively construct explanations for specific values of interests, and incur a bounded cost on enforcing constraints (*e.g.,* time). It also guarantees a competitive ratio $6 \cdot ln|V| + ln(\frac{c_u}{c_l}) + 1$, where $[c_l, c_u]$ is the range of the cost for enforcing a single data constraint in $\Sigma$.

*(5) Experimental study* (Section VI). Using real-world graphs, we experimentally verify the efficiency and effectiveness of our constraint-based algorithms. (1) We find our algorithms effectively exploit data constraints to clarify the missing data of interests that may not be inferred by batch enforcement. (2) The bidirectional search and optimization techniques effectively reduce the cost. For example, BiExp improves its unoptimized counterparts by 2.1 times, and is 3.1 and 7.3 times faster than the counterparts using forward and backward search alone, respectively. (3) Our algorithms effectively clarify specific missing values with bounded enforcement cost and for subgraph queries, as verified by our case study.

**Related work**. We categorize the related work as follows.

*Graph refinement*. Various link prediction methods have been studied to infer missing links in graphs [1], including association analysis [5], [13], supervised link prediction [2], and embedding methods [1]. Graph entity resolution [16] aims to infer node equivalence by enforcing keys [6], [10], association rules [17] or via interactive inference [18]. Graph identification [19] uses probabilistic soft logic to infer identical nodes and missing facts in knowledge graphs. These methods are optimized to refine the entire graph. In contrast, our method enforces data constraints to infer a necessary amount of data towards user-specified values, for which enforcing all the missing values is neither practical nor necessary.

Graph data repairing. Functional dependencies for graphs (GFDs) [7], [20], among rule-based graph repairing [3], [21], [22] uses constraints and data quality rules to capture and remove inconsistencies in graphs. For example, Chase for graphs [20] captures a sequence of quotient graphs induced by equivalent classes of nodes upon the enforcement of GFDs. Subset repairs and prioritized repairs based on Pareto optimality [23] aim to derive best replacement of values that can improve the overall inconsistencies. Our work differs from these work. (1) Rather than repairing the entire graph, we aim to infer necessary data and responsible constraints to clarify user-specific values. Our non-destructive approach enriches missing values to ensure more informative graph instances. (2) We use Chase to characterize constraint-based explanations. Unlike Chase that resolve inconsistent values with a preferred value [7], [20], our process preserves values with union semantics to preserve possible explanations.

*Why provenance*. Why-provenance has been studied for relation data [24], [25] and graphs [4]. Data provenance (or Why-provenance) tracks the data that contributes to query answers [24]. Query provenance (or "Why-not" questions) identifies operators that lead to desired answers [4]. Both approaches assume that the data of interests are already in the database, but cannot clarify missing data that is not in the data. Constraints are used to explain certain answers in incomplete databases [26], yet feasible algorithms are not discussed especially for graph data. Data provenance with functional dependencies [15], [27] apply static enforcement of data constraints *e.g.,* inclusion dependencies to derive non-operational explanations as possible values. These work do not consider operational enforcement of data constraints that involve graph patterns and their dynamic enforcement, which are needed to clarify specific missing values in graphs.

## II. Graphs and Data Constraints

We start with a notion of graphs with missing values.

**Graphs**. A graph $G = (V, E, L, F, X)$ contains a finite set of nodes $V$, a set of edges $E \subseteq V \times V$, and a set of associated variables $X$ to denote missing values.

(1) Each node $v \in V$ has a label $L(v)$ (*e.g.,* type) and an identifier id that refers to a real world entity (*e.g.,* URI). It carries a tuple $F(v)$ that is defined on a set of attributes $\mathcal{A}$. The value of an attribute $v.A$ in the tuple $F(v)$ is either a constant $a$ from a finite set $\mathcal{C}$, or a distinct *node variable* $x_{A_v}$ ='$\perp$' (marked null) from $X$ that represents a missing value.

(2) Each edge $r(v, v') \in E$ carries an edge label $r \in \mathcal{C}$ (*e.g.,* a relation name). There is an *edge variable* $x^r_{vv'} \in X$ with a value'$\perp$' for each $r(v, v') \notin E$ ($x^r_{vv'}$ = 1 if $r(v, v')$ belongs to the missing fraction of $G$; $x^r_{vv'}$ = 0 otherwise).

We assume $G$ has a complete set of nodes $V$. Each variable $x \in X$ from $G$ denotes a missing *element*. An element is

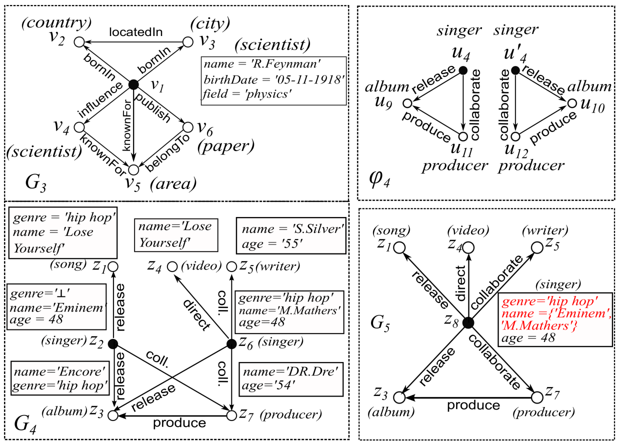


Fig. 2: Graphs and instances (a case of NE).

either an attribute-value pair $(v.A, a)$ that specifies a value $a$ for attribute $v.A$, or an edge $r(v, v')$.

Valuation. Given a graph $G = (V, E, L, F, X)$, a *valuation* of $G$ is a function defined on $X$ such that there is at least one variable $x$='$\perp$' in $X$ valuated to: (1) a set of constants $C_{A_v} \subseteq \mathcal{C}$, if $x$ is a node variable, or (2) $x$=1, if $x$ is an edge variable. An *instance* of $G$ is a graph induced by a valuation. An instance can be a multigraph and may contain self-loops.

**Example 3:** An incomplete graph $G$ is illustrated in Fig. 1 (excluding the edge $\mathsf{bornIn}(v_1, v_2)$ in $G_1$). Each node $v_i$ has a distinct identifier id $= i$ ($i \in [1, 6]$). The set $X$ marks three missing elements: $X = \{x_{field_{v_1}} = \perp, x^{bornIn}_{v_1 v_2} = \perp, x^{knownFor}_{v_1 v_5} = \perp\}$, which encode a missing attribute value, and two missing edges. A valuation that sets $x_{field_{v_1}}$ = 'physics' (resp. $x^{bornIn}_{v_1 v_2}$ = 1) induces an instance of $G$ with new elements $v_1$.field = 'physics' (resp. $bornIn(v_1, v_2)$).

Similarly, Fig. 2 illustrates a graph $G_4$ of facts of entertainment, with a missing value genre at $z_2$. □

**Remarks**. We consider a weaker representation of incomplete graphs with marked nulls, similar to conditional tables [28]. A valuation can replace null with a set of possible values. This allows us to keep track of possible instances that may contribute to user-defined missing values (see Section III).

**Graph data constraints**. We next review data constraints for graphs. A data constraint $\varphi$ defined on a graph $G$ may

- enforce node equality (denoted as a NE constraint):

$$\mathcal{P} \rightarrow u_o.\mathsf{id} = u'_o.\mathsf{id}$$

- or assert a missing edge (denoted as an EG constraint):

$$\mathcal{P} \rightarrow \exists r(u_o, u'_o)$$

Here $\mathcal{P} = (P, \mathcal{L})$ (called a *pattern*) specifies a graph pattern $P(u_o, u'_o)$ and a set of literals $\mathcal{L}$ defined on attributes $\mathcal{A}$. (1) A *graph pattern* $P(u_o, u'_o)$ contains a set of nodes (variables) $V_P$ and edges $E_P$. Each node $u \in V_P$ (resp. edge $e \in E_P$) has a label $L_P(u)$ (resp. $L_P(e)$). The nodes $u_o$ and $u'_o$ in $V_P$ are two designated *entity nodes*. (2) Each literal $l \in \mathcal{L}$ is either a constant literal $u.A = a$ ($a$ is a constant), or a variable literal $u.A = u'.A'$ ($u, u' \in V_P$, $A, A' \in \mathcal{A}$).

*Matches*. We characterize graph data constraints with *graph homomorphism* [29], which subsumes several common semantics [5]–[8], [11] and takes into account the labels of nodes and edges. Consider $\varphi$ with pattern $\mathcal{P}$=$(P, \mathcal{L})$.

(1) A *matching* from graph pattern $P(u_o, u'_o)$ to $G$ is a function $h$ from $V_P$ to $V$, such that (a) for each edge $(u, u') \in E_P$, $(h(u), h(u')) \in E$, $L_P(u) = L(h(u))$, $L_P(u') = L(h(u'))$, and (b) $L_P(h(u), h(u')) = L((u, u'))$. A pair of nodes $(v, v')$ is a *match* of $P$ induced by $h$, if $h(u_o) = v$ and $h(u'_o) = v'$.

(2) A pair of nodes $(v, v')$ is a *match* of $\varphi$, if it is a match of $P(u_o, u'_o)$ induced by a matching $h$ from $P(u_o, u'_o)$ to $G$, and $h$ satisfies $\mathcal{L}$, *i.e.,* for each literal $l \in \mathcal{L}$, $h(u).A = a$ (resp. $h(u).A = h(u').A$) if $l = (u.A = a)$ (resp. $u.A = u'.A$).

**Example 4:** Fig. 1 illustrates the patterns of constraints $\varphi_1$, $\varphi_2$, $\varphi_3$ with conditions and matches on $G$ specified as follows.

| $\mathcal{P}$ | Graph pattern $P$ | Literals $\mathcal{L}$ | Matches |
|---|---|---|---|
| $\mathcal{P}_1$ | $P_1(u_1, u'_1)$ | $\{u_1.\mathsf{name} = u'_1.\mathsf{name}$, $u_1.\mathsf{DOB} = u'_1.\mathsf{DOB}$, $u_4.\mathsf{val} = u'_5.\mathsf{val}\}$ | $\emptyset$ |
| $\mathcal{P}_2$ | $P_2(u_2, u'_2)$ | $\emptyset$ | $(v_1, v_2)$ |
| $\mathcal{P}_3$ | $P_3(u_3, u'_3)$ | $\{u_3.\mathsf{field} = u_8.\mathsf{field}\}$ | $\emptyset$ |
| $\mathcal{P}_4$ | $P_4(u_4, u'_4)$ | $\{u_9.\mathsf{name} = u_{10}.\mathsf{name}$, $u_9.\mathsf{genre} = u_{10}.\mathsf{genre}$, $u_{11}.\mathsf{name} = u_{12}.\mathsf{name}\}$ | $(z_2, z_6)$ |

Fig. 2 illustrates another pattern $\mathcal{P}_4 = (P_4(u_4, u'_4), \mathcal{L}_4)$, with the conditions and matches in $G_4$ also illustrated above. The matches $\mathcal{P}_4(G_4) = \{(z_2, z_6)\}$ is induced by a matching $h$, where $h(u_{11})$=$h(u_{12})$=$z_7$, and $h(u_9)$=$h(u_{10})$=$z_3$. $\square$

*Semantics*. A $\mathsf{NE}$ (resp. $\mathsf{EG}$) $\varphi$ with pattern $\mathcal{P}$=$(P(u_o, u'_o), \mathcal{L})$ states that "*for any match $(v, v')$ of $(u_o, u'_o)$, $v$ and $v'$ are equivalent and should refer to a same entity*" (resp. "*has a missing edge $r(v, v')$*"). A match $(v, v')$ of a $\mathsf{NE}$ (resp. $\mathsf{EG}$) $\varphi$ is a *violation* of $\varphi$ if $v.\mathsf{id} \neq v'.\mathsf{id}$ (resp. $r(v, v') \notin E$).

A graph $G$ *satisfies* a $\mathsf{NE}$ (resp. $\mathsf{EG}$) $\varphi$, denoted as $G \models \varphi$, if there exists no violation of $\varphi$ in $G$. It satisfies a set of constraints $\Sigma$ ($G \models \Sigma$), if for every $\varphi \in \Sigma$, $G \models \varphi$. In the rest of the paper, we consider $\Sigma$ as a set of $\mathsf{NE}$s and $\mathsf{EG}$s.

**Example 5:** Fig. 1 illustrates three constraints: a $\mathsf{NE}$ $\varphi_1$: $\mathcal{P}_1 \rightarrow u_1.\mathsf{id} = u'_1.\mathsf{id}$, an $\mathsf{EG}$ $\varphi_2$: $\mathcal{P}_2 \rightarrow \exists\,\mathsf{bornIn}(u_2, u'_2)$, and an $\mathsf{EG}$ $\varphi_3$: $\mathcal{P}_3 \rightarrow \exists\,\mathsf{knownFor}(u_3, u'_3)$. As is, $G \models \{\varphi_1, \varphi_3\}$, and $G \not\models \varphi_2$, where $(v_1, v_2)$ is a violation of $\varphi_2$.

We illustrate three more data constraints on $G_4$ in Fig. 2 and Fig. 3: $\mathsf{NE}$ $\varphi_4$: $\mathcal{P}_4 \rightarrow u_4.\mathsf{id} = u'_4.\mathsf{id}$, $\mathsf{EG}$ $\varphi_5$: $\mathcal{P}_5 \rightarrow \exists\,\mathsf{ost}(u_5, u'_5)$ and $\mathsf{EG}$ $\varphi_6$: $\mathcal{P}_6 \rightarrow \exists\,\mathsf{collaborate}(u_6, u'_6)$. $\square$

## III. Explaining Missing Values with Constraints

### A. Constraint Enforcement and Sequences

We next characterize the enforcement of data constraints following Chase [28], [29]. We start with a notion of operators.

**Operators**. An operator "modifies" graph $G$ to remove a violation of a data constraint. We consider the following.

(1) A *merge* operator $\circ(v, v')$ for a $\mathsf{NE}$ $\varphi$ replaces $v$ and $v'$ with a new node $v''$ (an "equivalent class") as follows. (a) $L(v'') = L(v) = L(v')$, $v''.\mathsf{id} = \min\{v.\mathsf{id}, v'.\mathsf{id}\}$. (b) Set tuple $F(v'')$ by setting $v''.A = v.A \cup v'.A$ for each attribute $A$ in $F(v)$ or $F(v')$ ($v.A = \emptyset$, if $A$ is not in $F(v)$ or $v.A = \perp$; similarly for $v'.A$). (c) Redirects the edges of $v$ and $v'$ to $v''$.
(2) An *insertion* operator $\oplus((v, v'), r)$ enforced by an $\mathsf{EG}$ $\varphi$ over a violation $(v, v')$ inserts a new edge $r(v, v')$.

**Sequences**. Given a graph $G$ and constraints $\Sigma$, an *action* $s$ is a triple $((v, v'), o(v, v'), \varphi)$, where $(v, v')$ is a violation of $\varphi$ in $G$, and $o(v, v')$ is an operator ($\circ$ or $\oplus$) that removes the violation $(v, v')$ of $\varphi$ from $G$. The *result* of $s$ on $G$, denoted as $G^s$, refers to the graph obtained by applying $o(v, v')$ on $G$.

A *sequence* $\rho = \{s_1, \ldots, s_n\}$ is a nonempty sequence of actions from $G$ with result $G'$ (denoted as $G' = G^\rho$), where $G_i = G_{i-1}^{s_i}$ ($i \in [1, n]$, $G_0$=$G$, $G_n$=$G'$). Specifically, $\rho$ can be a single "identity" action $\{\epsilon\}$, which yields $G$ itself ($G^\epsilon$=$G$).

**Example 6:** Given graph data constraints $\Sigma = \{\varphi_1, \varphi_2, \varphi_3\}$, and a sequence of actions $\rho = \{s_1, s_2, s_3\}$ where $s_1 = ((v_1, v_2), \oplus((v_1, v_2), \mathsf{bornIn}), \varphi_2)$, $s_2$=$((v_1, v'_1), \circ(v_1, v'_1), \varphi_1)$, and $s_3 = ((v_1, v_5), \oplus((v_1, v_5), \mathsf{knownFor}), \varphi_3)$, we have $G_3 = G^\rho$ (as illustrated in Fig. 2). Note that $G_3 \models \Sigma$.

For $\Sigma = \{\varphi_4, \varphi_5, \varphi_6\}$ (Fig. 2, 3), $G_5 = G_4^{s_1}$ (resp. $G_6 = G_5^{s_2}$) with $s_4 = ((z_2, z_6), \circ(z_2, z_6), \varphi_4)$ (resp. $s_5 = ((z_1, z_9), \oplus((z_1, z_9), ost), \varphi_5)$). The node $z_8$ is obtained from $\circ(z_2, z_6)$, which takes their common attribute $\mathsf{age}$ ='48', resolves possible values for $\mathsf{name}$ as a set $C_{name_{z_8}}$ = {'Eminem', 'M.Mathers'}, and retains 'genre' from $z_6$. One can verify that $G_5 \models \varphi_4$, $G_5 \not\models \varphi_5$, and $G_6 \models \{\varphi_4, \varphi_5\}$. $\square$

The result below verifies that sequences preserve the information of $G$. In other words, they are "non-destructive".

**Lemma 1:** *Given graph G and constraints $\Sigma$, for any sequence $\rho$, there is a graph homomorphism $h^\rho$ from G to $G^\rho$, and for any element $g$=$(v.A, a)$ (resp. $r(v, v')$) in G, $h^\rho(g) = (h^\rho(v).A, a)$ (resp. $r(h^\rho(v), h^\rho(v'))$) is an element in $G^\rho$.* $\square$

**Proof sketch:** Let $\rho = \{s_1, \ldots, s_n\}$. We construct $h^\rho$ as a composition function $h^{s_1} \ldots h^{s_n}$. Each $h^{s_i}$ ($i \in [1, n], G_0 = G, G_n = G^\rho$) is a function from the nodes of $G_i$ to those in $G_{i+1}$, where $h^{s_i}(v) = h^{s_i}(v') = v''$, if $s_i = \circ(v, v')$ and yields $v''$ in $G_{i+1}$, or $h^{s_i}(v) = v$ otherwise. We can verify that each $h^{s_i}$ ($i \in [1, n]$) (resp. $h^\rho$) is a graph homomorphism from $G_i$ to $G_{i+1}$ (resp. $G$ to $G^\rho$) that preserves the elements of $G$. $\square$

Given Lemma 1, we say an element $g = (v.A, a)$ (resp. $r(v, v')$) *occurs* in the instance $G^\rho$ of $G$, simply denoted as $g {\in} G^\rho$, if $h^\rho(v).A$=$a$ (resp. $r(h^\rho(v), h^\rho(v')$ is an edge) in $G^\rho$.

**Properties**. We justify the sequences by showing that they ensure the *informativeness* of the results towards a *unique* result. We start with a notion of *informativeness ordering*.

*Informativeness*. Given $G$ and $\Sigma$, we denote the set of all the results of a sequence from $G$ (including $G$) as $G_\Sigma$. Given two instances $G'$ and $G''$ in $G_\Sigma$, we say $G'$ is *not more informative* then $G''$, denoted as $G' \preceq G''$, if $G''_\Sigma \subseteq G'_\Sigma$.

Intuitively, $G''$ is "more informative" if it has *less* results $G''_\Sigma$. That is, the more possible instances of $G'$ that can be derived by a sequence that enforces the data constraints from $\Sigma$, the less informative $G'$ is. We have the following result.

**Theorem 2:** *Given a graph $G$ and constraints $\Sigma$, for any instance $G' \in G_\Sigma$ and any sequence $\rho$ from $G'$, $G' \preceq G'^{\rho}$.* □

We prove Theorem 2 by showing the following result.

**Lemma 3:** *The relation $\preceq$ is a partial order over $G_\Sigma$.* □

Intuitively, a sequence results valuated instances that become more "certain" on missing values. We provide the detailed proof of Lemma 3 and Theorem 2 in [30].

Uniqueness. Following Chase and its Church-Rosser property [28], [29], we next present a *uniqueness* guarantee, which states that any sequence will terminate at a "unique" result. We say a sequence $\rho$ *terminates* at an instance $G' \in G_\Sigma$ if $G'^{s} = G'$ for any possible action $s$. We show the following result.

**Theorem 4:** *Given $G$ and $\Sigma$, (1) any sequence from $G$ terminates with at most $|V| + |V|^2$ actions, and (2) for any two sequences $\rho$ and $\rho'$, the results $G^{\rho}$ and $G^{\rho'}$ are homomorphically equivalent,* i.e., *there exists a graph homomorphism from $G^{\rho}$ to $G^{\rho'}$, and vice versa.* □

**Proof sketch:** Theorem 4 (1) can be verified by observing that any action reduces either one node or insert one s edge from a finite node set $V$ of $G$. We show Theorem 4(2) by contradiction. Assume $G^{\rho}$ to $G^{\rho'}$ are not homomorphically equivalent, then there exists at least one violation in *w.l.o.g.* $G^{\rho}$ and an action $s$ such that $G^{\rho s} \neq G^{\rho}$, which contradicts that $\rho$ terminates at $G^{\rho}$. We present the details in [30]. □

## B. Explanations for Missing Values

We next characterize the explanations for specific element.

**$\Sigma$-explainable**. Given graph $G$ and constraints $\Sigma$, a missing element $g$ not in $G$ is $\Sigma$-*explainable*, if there is a sequence $\rho$ such that $g \in G^{\rho}$. We say $\rho$ is an *explanation* of $g$.

**Example 7:** Continue with Example 6. The two sequences $\rho_1 = \{s_1, s_2, s_3\}$ over $G$ (Fig. 1) and $\rho_2 = \{s_4, s_5\}$ over $G_4$ in Example 6 are explanations for missing elements KnownFor $(v_1, v_5)$ in $G$ and $\mathsf{ost}(z_1, z_9)$ in $G_4$, respectively. Note that $\rho_1$ terminates at $G_3$ given $\Sigma = \{\varphi_1, \varphi_2, \varphi_3\}$. □

Given Theorem 2 and Theorem 4, an explanation $\rho$ of a missing element $g$ is well-defined with guarantees on informative instances, ensures the occurrence of $g$, and eventually terminates at unique result up to graph homomorphism.

## C. Measures for Explanations

There can be multiple explanations for a missing element. We next introduce measurements for "good" explanations.

**Informativeness**. Given an explanation $\rho$ of element $g$ that starts from $G$, the *cumulative informativeness gain*, denoted as $\mathsf{cg}(\rho, G)$, is defined as

$$\mathsf{cg}(\rho, G) = \sum_{s \in \rho} \mathsf{supp}(s, G) \cdot \mathsf{cg}(s, G)$$

where $\mathsf{supp}(s, G)$ for $s$ that enforces $\varphi$ (with $\mathcal{P}$ and graph pattern $P$) is computed as the fraction of the matches that satisfy $\varphi$ to the total matches $P(G)$. $\mathsf{cg}(s, G)$ is the informativeness gain of an action $s \in \rho$, and is separately defined as follows:

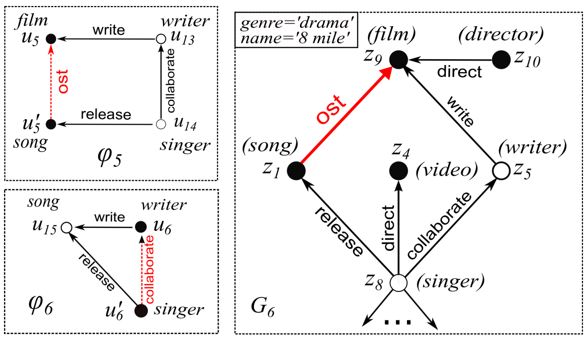


Fig. 3: Instances by enforcing NEs and EGs

$$\mathsf{cg}(s, G) = \begin{cases} \frac{1}{|r(L(v), L(v'))|}, & \text{if } s = \oplus((v, v'), r) \\ \mathsf{diff}(v, v'), & \text{if } s = \circ(v, v') \end{cases}$$

(1) For edge insertion $\oplus$, $r(L(v), L(v')) = \{(v_1, v_2) | (v_1, v_2) \in E, h(v_1) = h(v), h(v_2) = h(v')\}$. The measure $\mathsf{cg}(s, G)$ ($\mathsf{cg}(s, G) \in (0, 1]$) quantifies the additional gain under partial closed world assumption [2], [31]: the more edges similar to $r(v, v')$ are known, the less the gain is. $\mathsf{cg}(s, G) = 1$ if $r(L(v), L(v')) = \emptyset$ (by inserting "a first of its kind") [31].

(2) For node merge $\circ$, $\mathsf{cg}(s, G)$ favors to merge two equivalent nodes with tuples $F(v)$ and $F(v')$ that are more different. This can be quantified as the symmetric difference $\mathsf{diff}(v, v')$ of their attributes [32], which indicates more variables can be valuated, or more possible values can be identified. We provide more details of cg (supp and diff) in the full version [30].

**Conciseness**. We follow the principle of minimality for explanations. A *minimal* explanation for an element $g$ is a sequence $\rho$ such that $g \in G^{\rho}$, and $g \notin G^{\rho'}$ for any subsequence $\rho'$ of $\rho$. We prefer minimal explanations within a bounded length. Note that $|\rho| \leq |V|^2 + |V|$ for any sequence $\rho$ (Theorem 4).

**Example 8:** Assume the nodes $z_2$ and $z_6$ in Example 6 have another common attribute occupation (not shown) then $\mathsf{cg}(s_4, G_4) = \frac{2}{4}$. We can also verify that $\mathsf{cg}(s_5, G_4) = 1$. Given that $\mathsf{supp}(s_4, G_4) = 1$ and $\mathsf{supp}(s_5, G_4) = 1$, we have $\mathsf{cg}(\rho_2, G_4) = 1.5$. □

Properties of measures. We show the following properties of our measures. (1) $\mathsf{cg}(\rho) = 0$ if $\rho = \emptyset$ (Consistency:"no gain if not an explanation"). (2) $\mathsf{cg}(\rho) \geq \mathsf{cg}(\rho')$ if $G^{\rho} \preceq G^{\rho'}$ (Informativeness: "more informative result, more gain"). (3) $\mathsf{cg}(\rho\{s\}) \geq \mathsf{cg}(\rho\{s'\})$, if $s.\varphi \models s'.\varphi'$ (Generality: "prefer $\rho$ that enforces constraints which logically imply those enforced by others"). These properties justify our measures.

We present the detailed proofs of these properties in [30].

**Problem Statement**. Based on these measures, we study the following optimization problem, denoted as $\Sigma$-*explanation*.

- **Input**: Graph $G$, a missing element $g \notin G$ (can be a *wildcard* '_'), graph data constraints $\Sigma$, and a bound $b$;

- **Output**: a minimal explanation $\rho$ for $g$ s.t.

$$\rho = \arg\max_{|\rho'| \leq b} \mathsf{cg}(\rho', G)$$

The targeted element $g$ can be a wildcard ("don't care"). This is useful when no preference or ground truth is specified, and one wants to find what can be inferred (see Section V-A).

**Complexity**. We relativise the hardness of $\Sigma$-explanation with the *validation* problem, which is to decide whether $G \models \Sigma$ (Section II). We make a case for $\Sigma$ as a set of graph keys and graph association rules, and show the following result.

**Theorem 5:** *$\Sigma$-explanation is in $\Delta_2^P$ for $\Sigma$ defined as a set of graph keys and graph association rules.* □

**Proof sketch:** We prove Theorem 5 as follows.

(1) Consider the following special case of $\Sigma$-explanation: given $G$, $\Sigma$ and $g$='_', it is to decide whether there exists an explanation $\rho$ with size 1 (a single action). We show that this problem is a complement of the validation of $\Sigma$ over $G$: there exists an explanation if and only if $G \not\models \Sigma$.

(2) We next show that $\Sigma$-explanation is equivalent to deciding whether there is a path with length bounded by $b$ from $G$ to an instance with $g$, in the lattice $(G_\Sigma, \preceq)$ induced by the partial order $\preceq$ (Lemma 3). We provide a PTIME algorithm to solve $\Sigma$-explanation that invokes an oracle for the validation of $\Sigma$.

As the validation problem of graph keys and association rules is coNP-complete [33], Theorem 5 follows. □

We present the detailed proof in [30].

**Remarks**. We compare explanations with constraint-based data repairing. Unlike Chase sequences that resolve inconsistencies with a single preferred value (cf. [29]), we use merge operators to union possible values in order to identify possible explanations. Subset repairs and prioritized repairs with Pareto optimality [21] aim to resolve all the inconsistencies with minimal repairs. In contrast, an explanation infers small amount of new data that leads to designated missing values.

## IV. Computing Optimal Explanations

We next introduce algorithms for $\Sigma$-explanation. We assume an "oracle" (denoted as DetVio) is available to detect the violations of $\Sigma$ in $G$. It can be efficiently implemented by *e.g.,* incremental and parallel pattern matching [33].

*A naive solution*. Following Theorem 5, one may simply invokes DetVio to construct a lattice $\mathcal{E} = (G_\Sigma, \preceq)$, and computes the explanation as a shortest path (by reversing gains to distances). Nevertheless, this can be expensive due to the excessive number of instances and violation detection.

We next develop an efficient algorithm, denoted as BiExp, to compute explanations without constructing $\mathcal{E}$.

### *A. Bi-directional Exploration*

The algorithm BiExp initializes and explores a partially observed lattice $\mathcal{E} = (G_\Sigma, \preceq)$ (not known a-priori) with a *bi-directional* exploration strategy as follows.

```
Algorithm BiExp
Input: Graph $G$, element $g$, data constraints $\Sigma$, size bound $b$.
Output: a minimal explanation $\rho$ for $g$.
1.  initialize queue $\mathcal{Q}_f$:={$s_r$}; $\mathcal{Q}_b$:={$s_g$}; tree $T_f$={$s_r$}; $T_b$={$s_g$}
2.  while $\mathcal{Q}_f \neq \emptyset$ and $\mathcal{Q}_b \neq \emptyset$ and $h(T_f) + h(T_b) \leq b$ do
3.      if $\mathcal{Q}_f \neq \emptyset$ then $s_f$ := $\mathcal{Q}_f$.Dequeue(); /*Forward search*/
4.          if $s_f$=$s_g$ or $s_f \in \mathcal{Q}_b$ then
5.              return $\rho$ := ConstrExp($T_f, T_b, s_f$);
6.          for each $s'_f \in$ Forward($T_f, s_f$) do
7.              $\mathcal{Q}_f$.Enqueue($s'_f$);
8.      if $\mathcal{Q}_b \neq \emptyset$ then $s_b$ := $\mathcal{Q}_b$.Dequeue(); /*Backward search*/
9.          if $s_b$=$s_r$ or $s_b \in \mathcal{Q}_f$ then
10.             return $\rho$:=ConstrExp($T_f, T_b, s_b$);
11.         for each $s'_b \in$ Backward($T_b, s_b$) do
12.             $\mathcal{Q}_b$.Enqueue($s'_b$);
13. return $\emptyset$;
```

Fig. 4: Algorithm BiExp

- <u>*Forward search*</u>: starts from $G$ and invokes DetVio to explore admissible actions (called *forward frontier*).
- <u>*Backward search*</u> simultaneously starts with a "virtual" instance $G_g$ that contains $g$, and "reverse engineers" $\Sigma$ enforcement to explore a set of *enabling* actions (maintained in its *backward frontier*) that may result $G_g$.

The algorithm reconstructs explanations upon frontier intersection over explored fraction of $\mathcal{E}$.

The bidirectional search reduce unnecessary exploration by refining forward frontier given current backward frontier, and vice versa. As the unknown $G_\Sigma$ is monotonically decreasing (ensured by Theorem 2), BiExp ensures better explanations by making the graph $G$ more "certain" in the exploration.

We start with the auxiliary structures of BiExp.

**Auxiliary structures**. Algorithm BiExp maintains the following. (1) A set of Boolean variables (called *element variable*), where a variable $x(g, G)$ is 1 if $g \in G$, and 0 otherwise. The values can be easily maintained by tracking the valuation of $G$ (the values of the variables $X$). (2) A *forward tree $T_f$* and a *backward tree $T_b$*, specified as follows.

<u>*Forward tree*</u>. The forward tree $T_f$ has a root $s_r$. Each node in $T_f$ is an action $s = ((v, v'), o, \varphi)$ ($s_r = (\emptyset, \epsilon, \emptyset)$). There is an instance $G_s$ associated with node $s$, where $G_s = G^\rho$, and $\rho$ is the sequence (path) from $s_r$ to $s$ in $T_f$. The *forward frontier $o_f$* of $T_f$ is a set of leaves (initialized as $s_r$) that are selected to be applied to generate new instances.

<u>*Backward tree*</u>. The backward tree $T_b$ maintains a set of *enabling* actions that are required (depending on forward frontier). $T_b$ is initialized as a root node $s_g$ as follows.

- If $g = r(v, v')$, $s_g = ((v, v'), \oplus((v, v'), r), \varphi))$, where EG $\varphi \in \Sigma$ asserts a missing edge $r(u_o, u'_o)$ such that $v$ and $u_o$ (resp. $v'$ and $u'_o$) have the same label.
- If $g = (v.A, a)$, $s_g = (\emptyset, \epsilon, \emptyset))$.

Sequences are constructed in a backward direction in $T_b$. The *backward frontier $o_b$* of $T_b$ contains a set of (backward) leaves, which are selected to explore preceding actions that *may* lead to $g$. For each node $s = ((v, v'), o(v, v'), \varphi)$ in $o_b$, the following are dynamically maintained:

```
Procedure Backward
Input: backward tree $T_b$, node $s$.
Output: backward frontier $o_b$.
1.  initializes enabler set $S_e := \{s\}$;
2.  induce graph $G(s) :=$ InduceSG$(G_s, d)$;
3.  while $s.con \neq$ False and $S_e \neq \emptyset$ do
    /* generate enablers for action $s$ */
4.      $S_e :=$ GenAction$(G(s), s)$;
5.      for each $s_e \in S_e$ and $\varphi \in \Sigma$ do
6.          if $(s_e.v, s_e.v')$ may violate $\varphi$ then
7.              $S := S \cup \{((s_e.v, s_e.v'), s_e.o, \varphi)\}$;
8.      $o_b = o_b \cup$ PruneBwd$(S)$; update $T_b$ and $s$.con;
9.  return $o_b$;
```

Fig. 5: Procedure Backward

- "virtual" instance $G_s$ associated to $s$,
- a set of *enablers*, where each enabler $s'$ of $s$ is an action that makes $(v, v')$ a violation of $\varphi$ in $G_{s'}$, and
- a Boolean condition $s$.con on the element variables.

The element variables are shared by $T_f$ and $T_b$. The condition $s$.con, incrementally evaluated by the shared element variables, tracks whether the required elements occurs to enable the application of $s$ towards interested element $g$. ($s$.con = true means $s$ becomes applicable; see "Optimization").

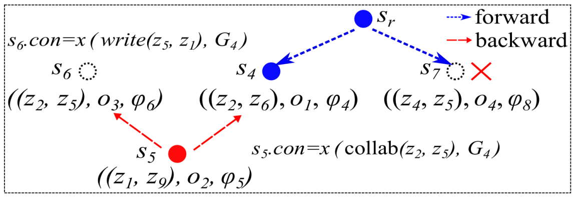
**Example 9:** Consider the computation of the explanation for element $g = x^{ost}_{z_1,z_9}$ in Figure 3. Following Example 6, $s_4$ is an enabler of $s_5$ as $(z_1, z_9)$ becomes a a violation of $\varphi_5$, encoded as an edge $(s_4, s_5)$ in backward search $T_b$. The enabling condition for $s_5$ is initialized as $s_2.con = x(collab(z_2, z_5), G_4)$. This reduces the problem to computing explanations for collab$(z_2, z_5)$. □

**Algorithm**. Our algorithm BiExp (as illustrated in Fig. 4) initializes the forward tree $T_f$ (resp. backward tree $T_b$ ) as a single root node $s_r$ (resp. $s_g$) (line 1). It also maintains two queues, $\mathcal{Q}_f$ and $\mathcal{Q}_b$ to store the forward frontier $o_f$ and backward frontier $o_b$, respectively (line 1).

BiExp next performs a bidirectional Breadth-First search (BFS) (up to size bound $b$; line 2), by invoking a procedure Forward (resp. Backward) to grow $T_f$ (resp. $T_b$ backwardly) and refine $o_f$ (resp. $o_b$), until a common action $s \in o_f \cap o_b$ is identified (lines 4-5, 9-10). This indicates the action $s$ is both verified by forward search and meanwhile "needed" as an enabler in backward search to include $g$, resulting a sequence $\rho$ from $s_r$ to $s_g$ passing $s$ with a result that includes $g$.

Upon the frontier intersection, BiExp invokes a procedure ConstrExp (not shown) to construct the optimal explanation (lines 5, 10). There may be multiple actions in $o_f \cap o_b$. For each $s \in o_f \cap o_b$, an explanation is constructed as the sequence from $s_r$ to $s_g$ passing $s$. The best one that maximizes accumulated gain cg is then returned ($\emptyset$ if $o_f \cap o_b$=$\emptyset$).

**Procedure** Forward. Given the current forward frontier $o_f$, procedure Forward identifies a set of applicable actions. For each action node $s = ((v, v'), o, \varphi) \in o_f$ with instance $G_s$, Forward invokes procedure DetVio to compute a set of violations Vio$(G_s, \Sigma) = \bigcup_{\varphi \in \Sigma}$ Vio$(G_s, \varphi)$, where Vio$(G_s, \varphi)$ refers to the violations of $\varphi$. It also invokes a procedure PruneFwd,



$s_r$
forward
backward
$s_6.con$=x(write($z_5$, $z_1$), $G_4$)
$s_6$
$s_4$
$s_7$
(($z_2$, $z_5$), $o_3$, $\varphi_6$)
(($z_2$, $z_6$), $o_1$, $\varphi_4$)
(($z_4$, $z_5$), $o_4$, $\varphi_8$)
$s_5$
$s_5.con$=x(collab($z_2$, $z_5$), $G_4$)
(($z_1$, $z_9$), $o_2$, $\varphi_5$)

Fig. 6: Bidirectional construction for element $g = x^{ost}_{z_1,z_9}$

to refine $o_f$ according to desired actions in $o_b$ (see "Optimization"). The new actions are added as children of $s$ in $T_f$.

**Procedure** Backward. Similarly as Forward but more involved, Backward (illustrated in Fig. 5) determines a set of enablers for the current $o_b$, and refines $o_b$ with enablers.

<u>*Computing enablers*</u>. For each action $s = ((v, v'), o, \varphi) \in o_b$, Backward computes a set of enablers as follows.

(1) If $(v, v')$ is not already a match of $\varphi$ with pattern $\mathcal{P}$=$(P, \mathcal{L})$) in $G$, Backward first computes a set of *enabling elements* $S_e$ for $s$. A missing element $g'$ is an enabling element for $s$, if adding it to $G_s$ make $(v, v')$ a match of $\varphi$ in $G_s$. Specifically, Backward performs the following.

- Induce a subgraph $G(s)$ of $G_s$ (procedure InduceSG, line 2) with the $d$-hop neighbors $N_d(s) = N_d(v) \cup N_d(v')$, where $d$ is the diameter of the graph pattern $P(u_o, u'_o)$ of $\mathcal{P}$ (*i.e.,* the length of the longest shortest paths between any two nodes in $P$). It suffices to consider $G(s)$ due to the data locality of graph homomorphism.
- Construct a set of missing edges $r(v_1, v_2)$ and missing attribute values $(v_1.A, a)$ from $G(s)$ , where $r(v_1, v_2)$ and $v_1.A$ are needed to form a graph homomorphism from $P(u_o, u'_o)$ to $G(s)$ that also satisfy $\mathcal{L}$.

(2) For each enabling element $g'$ and a corresponding $\varphi \in \Sigma$, an enabler is constructed if enforcing $\varphi$ may introduce $g'$ (Procedure GenAction, lines 4-7). The enablers are further refined by procedure PruneBwd (see "Optimization") and are added to backward frontier $o_b$ (line 8). .

**Example 10:** Continue with Example 9, Backward first induces a subgraph (with $d = 2$) from $z_1$ and $z_9$. It then generates two enablers (illustrated in Figure 6): $s_4$ which merges node $z_2$ and $z_6$, and $s_6$ which adds an edge between $z_2$ and $z_5$. BiExp continues from forward search and finds $s_4 \in o_f \cap o_b$. This generates a best explanation $\rho = \{s_4, s_5\}$ for the missing element $g = x^{ost}_{z_1,z_9}$ (Figure 3). □

## *B. Optimization*

BiExp uses two strategies below to further reduce the cost.

**Early termination with conditions**. BiExp maintains a condition $s$.con = $\bigvee \bigwedge(x)$ for each $s \in o_b$. Each clause $\bigwedge(x)$ involves element variables for a set of elements in a same matching $h$ to make $s.(v, v')$ a match of $s.\varphi$ in $G_s$.

BiExp incrementally updates $s$.con with partial evaluation [34]. It dynamically induces residual conditions upon the assignment of an element variable $x$ (true or false), without waiting for all the element variables to be evaluated. This enables early pruning of branches (when all disjunct clauses become false, *i.e.,* $s$.con=false), or early detection of frontier intersection (when a disjunct clause is true, *i.e.,* $s$.con=true).

**Bidirectional pruning**. The forward and backward search interactively refine each other by "pre-matching" the enablers and actions without verification. Given $\Sigma$ and constraints $\varphi, \varphi' \in \Sigma$, we say $\varphi$ triggers $\varphi'$ ($\varphi \vdash \varphi'$), if one can find small model $(G, G_1, G_2)$ such that $G_1, G_2 \in G_\Sigma$, $G_1 = G^s$, $G_2 = G^{s,s'}$, and $s$ enforces $\varphi$, $s'$ enforces $\varphi'$.

BiExp estimates a *forward closure* $\varphi^{+\downarrow}$ and a *backward closure* $\varphi^{+\uparrow}$ of a data constraint $\varphi \in \Sigma$ as follows. We observe $\varphi \not\vdash \varphi'$ if (a) a NE $\varphi$ enforces equivalence on nodes with label $L(u_o) = L(u'_o)$, and no nodes in the pattern of $\varphi'$ has the same label; (b) an EG $\varphi$ enforces edge $r(v, v')$ and no edge in the pattern of $\varphi'$ has matching node and edge label. This can be checked in PTIME. Denote all such data constraints $\varphi'$ (resp. $\varphi$) for $\varphi$ (resp. $\varphi'$) as $\varphi{\downarrow}^-$ (resp. $\varphi'{\uparrow}^-$).

(1) $\varphi^{+\downarrow}$ is define as: (a) $\varphi \in \varphi^{+\downarrow}$, and (b) $\varphi' \in \varphi^{+\downarrow}$ if there exists a data constraint $\varphi'' \in \varphi^{+\downarrow}$ such that $\varphi' \notin \varphi''^{\downarrow -}$.

(2) $\varphi^{+\uparrow}$ is defined as: (a) $\varphi \in \varphi^{+\uparrow}$, and (b) $\varphi' \in \varphi^{+\uparrow}$ if there exists a data constraint $\varphi'' \in \varphi^{+\uparrow}$ such that $\varphi' \notin \varphi''{\uparrow}^-$.

The forward closures of a set $\Sigma^{+\downarrow}$ is defined as $\bigcup_{\varphi \in \Sigma} \varphi^{+\downarrow}$. $\Sigma^{+\uparrow}$ is defined similarly. Denote the constraints involved in the enablers and actions in $o_f$ and $o_b$ as $\Sigma_o$ and $\Sigma_b$, respectively. PruneBwd and PruneFwd refine $o_f$ and $o_b$ as follows.

**Lemma 6:** *For any action $s$ that enforces $\varphi \in \Sigma$, and any explanation $\rho$ of $g$ such that $G^\rho \in G_\Sigma$, $s \notin \rho$ if (1) $s \in o_b$, and $\varphi^{\uparrow +} \cap \Sigma_o^{+\downarrow} = \emptyset$; or (2) $s \in o_f$, and $\varphi^{\downarrow +} \cap \Sigma_b^{+\uparrow} = \emptyset$.* □

Lemma 6 allows the forward and backward search to iteratively refine each other (see detailed proof in [30]).

**Example 11:** Following Example 10, $s_6$ requires a single enabler to introduce collab$(z_2, z_5)$. As $\{\varphi_6\}^{\uparrow +} = \emptyset$ (no constraint in $\Sigma$ can trigger $\varphi_6$), PruneBwd removes $s_6$ from $o_b$. □

These optimization is quite effective. For a real graph with 4.5 million edges, the pruning reduces the time cost by 52% without losing the quality of explanations (see Section VI).

**Analysis**. BiExp simulates a bidirectional breadth first search of the lattice $(G_\Sigma, \preceq)$, with at most $\min\{|V|^2|\Sigma|, b\}$ forward or backward spawning. Each spawning invokes DetVio with a time $T$. Each spawning triggers at most $|V|^2$ violations, thus generate up to $|V|^2|\Sigma|$ actions. The bidirectional search is thus in $O(T \cdot (|V|^2|\Sigma|)^{\frac{b}{2}})$, given that $b \ll |V|$. (2) For correctness, BiExp terminates with a minimal explanation $\rho$ ($\emptyset$ if $g$ is not $\Sigma$-explainable). It computes $\rho$ with the maximized gain as a shortest path from $s_r$ to $s_g$ in the lattice $(G_\Sigma, \preceq)$ (Theorem 5).

## V. Extensions

In this section, we extend BiExp to support cost-effective graph refinement, and to clarify missing answers.

### A. Budgeted Graph Refinement

Refining graphs by inferring new elements with $\Sigma$ can still be expensive when no targeted element is provided ($g$='_') and for large $G$ (*e.g.,* social networks). It is often desirable to obtain some result first, and "gradually" improve the results towards overall solution [18], [35]. Such need can be addressed by solving the following budgeted $\Sigma$-explanation problem.

- **Input**: Graph $G$, a wildcard element, data constraints $\Sigma$, and a cost bound $B$;
- **Output**: a sequence $\rho^* = \arg\max_{|c(\rho)| \leq B}$ cg$(\rho, G)$.

where the cost $c(\rho) = \sum_{s \in \rho} c(s)$. Moreover, each action $s$ has a cost $c(s)$ that is not known a priori. In practice, the costs can be quantified by graph editing cost [35] or the dissimilarity of the nodes [1]. For simplicity, we choose unit cost. Our techniques can be readily extended to other cost functions.

```
Algorithm ApxExp
Input: an incomplete graph $G$, data constraints $\Sigma$, resource bound $B$.
Output: a sequence $\rho$.
1.  Queue $\mathcal{Q}$:=$\emptyset$; Sequence $\rho$:=$\emptyset$; Instance $G' = G$;
2.  initialize action $s_r$:= $(\emptyset, \epsilon, \emptyset)$;
3.  List $\mathcal{S}$ := NextBatch $(s_r, G', \Sigma)$; $\mathcal{Q}$.Enqueue($\mathcal{S}$);
4.  while $\mathcal{Q} \neq \emptyset$ and $c(\rho) \leq B$ do
5.      action $s$ := $\mathcal{Q}$.Dequeue();
6.      if $\frac{\mathsf{cg}(s,G')}{c(s)} \geq \Psi(\frac{c(\rho)}{B})$ then
7.          $\rho := \rho \cup \{s\}$; $G' := G'^s$;
8.          $\mathcal{S}$ := NextBatch$(s, G', \Sigma)$; $\mathcal{Q}$.Enqueue($\mathcal{S}$);
9.  return $\rho$;
```

Fig. 7: Algorithm ApxExp

The hardness of $\Sigma$-explanation remains intact for the above problem. As the cost of an action can only be determined upon observed, we aim to incrementally maintain a sequence $\rho^*$ that has the largest cg$(\rho)$ among the sequences over all the currently observed actions. We show the following result.

**Theorem 7:** *There is a one-pass algorithm for budgeted $\Sigma$-explanation with wildcard, with a competitive ratio $6 \cdot ln|V| + ln(\frac{c_u}{c_l}) + 1$ at any time, where $c_u$ and $c_l$ is the maximum and minimum cost of an action, respectively.* □

That is, the algorithm maintains a sequence $\rho$ at any time, such that cg$(\rho^*) \leq (6 \cdot ln(|V|) + ln(\frac{c_u}{c_l}) + 1)$ cg$(\rho)$. As a proof of Theorem 7, we outline such an algorithm.

**Algorithm.** The one-pass algorithm ApxExp is shown in Figure 7. It uses a queue $\mathcal{Q}$ to store candidate actions, and maintains an instance to be updated (initialized as $G$). Its main driver progressively extends a sequence $\rho$ as follows. It first invokes a procedure NextBatch to populate a batch of promising actions $\mathcal{S}$ (line 3). For each action $s \in \mathcal{S}$, it verifies whether its gain-to-cost ratio $\frac{\mathsf{cg}(s,G')}{c(s)}$ is no less than threshold $\Psi(\frac{c(\rho)}{B})$, and extends $\rho$ with $s$ if so (lines 6-7). The threshold function $\Psi(\cdot)$ for input $z$ is defined as

$$\begin{cases} \frac{1}{|V|^4 \cdot c_u} & \text{if } z \leq \frac{1}{1+6\cdot ln(|V|)+ln(\frac{c_u}{c_l})} \\ (\frac{|V|^6 \cdot c_u \cdot e}{c_l})^z (\frac{1}{e|V|^4 \cdot c_u}) & \text{otherwise} \end{cases}$$

where $\Psi(0)$ provides a lower bound of the gain for any single action. It then invokes NextBatch to spawn a new batch of actions given selected $s$ to be verified (line 8).

**Procedure** NextBatch**.** The procedure NextBatch performs forward search as Forward in BiExp, and refines the actions by pruning those that can no longer introduce gain beyond the threshold $\Psi(\cdot)$. It also sorts $\mathcal{S}$ to early terminate its sequential processing in ApxExp against a non-decreasing threshold. We present the details of NextBatch in [30].

**Analysis**. Algorithm ApxExp approximately maintains the optimal sequence by solving a budgeted secretary problem [36], [37], which is to choose a set of items from a sequence to maximize the total value under a fixed budget. (1) Let the lower and upper bound of the gain-to-cost ratio of an action to be $L$ and $U$ respectively. The selection of actions against gain-to-cost threshold ensures a competitive ratio [36], [37] as $ln(U/L)+1 = 6\cdot ln(|V|) + ln(\frac{c_u}{c_l})+1$. (2) For time cost, NextBatch takes $O(T|\Sigma||V|^2)$ time ($T$ is the cost of DetVio) to expand an action in forward search. As at most $\frac{B}{c_l}$ actions are inspected, the total cost is in $O(\frac{B}{c_l}\cdot|\Sigma||V|^2T)$.

Theorem 7 follows from the above analysis. When shorter inferences are preferred ($c(\rho) = |\rho|$), we show that ApxExp achieves better approximation of $6\cdot ln(|V|)+1$ (see [30]).

## *B. Explaining Missing Answers*

Our second extension has a practical premise that it is useful to provide users with operators to change $G$ such that a query $Q$ returns desired yet missing answer (cf. [25]).

**Queries**. A query $Q$ maps an instance $G$ to an answer $Q(G)$. The answer $Q(G)$ is a set of elements (attribute value pairs or edges) that only contain constant values. An element $g$ is a missing answer if $g \notin Q(G)$. We make case for subgraph queries $Q$, which returns a set of nodes induced by graph homomorphisms (Section II) from $Q$ to $G$, *e.g.,* SPARQL.

We study the following variant of $\Sigma$-explanation.

- **Input**: instance $G$, a query $Q$ and query answer $Q(G)$, an element $g \notin Q(G)$; constraints $\Sigma$, and a size bound $b$;
- **Output**: a minimal sequence $\rho$ s.t. $g \in Q(G^\rho)$ and $|\rho| \leq b$.

We next outline a variant of BiExp, denoted as BiExpQ to compute explanations for a missing answer $g \notin Q(G)$.

**Algorithm**. The algorithm BiExpQ uses the bidirectional search as in BiExp. The differences are as follows.

(1) Besides the instance graph $G_s$ at each node $s$, BiExpQ also tracks the query answer $Q(G_s)$, in both forward and backward search. $Q(G_s)$ can be incrementally maintained [38].

(2) Given an action $s$ in $o_b$, BiExpQ extends Backward to generate refined enabling elements. These elements not only make $s$ applicable, but are also the missing elements in the potential matches of $Q$ in order to make $g$ a part of the answer.

(3) Unlike BiExp, when an element $s$ is identified in $o_b \cap o_f$, it extracts $\rho$ that contains $s$, and verifies whether $g \in Q(G^\rho)$.

**Analysis**. The correctness of BiExpQ follows from the variant that any returned sequence $\rho$, if not $\emptyset$, is an explanation of $g$, and the verification further ensures $g \in Q(G^\rho)$. For time cost, BiExpQ performs one additional step for query processing for each action, and remains to be in $O(T\cdot(|V|^2|\Sigma|)^{\frac{b}{2}})$ time.

## VI. Experiment

Using real-world graphs and query benchmark, we evaluate (1) the effectiveness and efficiency of BiExp and BiExpQ on clarifying missing elements; (2) the effectiveness of ApxExp for budgeted inference, and (3) case analysis for applications.

**Experimental setting.** We used the following setting.

*Datasets.* We use three real graphs. Each graph $G$ contains nodes that are curated from two real knowledge bases, with a set of "ground truth" node pairs $\Gamma$. Each pair in $\Gamma$ is either equivalent or bears a missing edge.

(1) DBYa[1]: [10] with $592K$ nodes, $4.5M$ edges, and $50K$ equivalent pairs (covering 10 types of entities) with aligned attributes curated from knowledge bases DBPedia and YAGO.

(2) DBIM[2]: [39] contains $33K$ nodes, $200K$ edges and $33.4K$ entities covering 10 types, totaling $9.5K$ equivalent pairs across DBPedia and IMDb (a movie knowledge base).

(3) OAG[3]: an open academic graph which unifies Microsoft Academic Graph and Aminer. We sample graphs that contain papers with selected topics (*e.g.,* "database", "machine learning") and related information (authors, citation). The graph contains $2.5M$ nodes, $5.2M$ edges and $106K$ equivalent pairs.

(4) SynDBYa: synthetic graphs with ground truth, initialized by DBYa and its ground truth. We enhance $\Gamma$ of DBYa by duplicating the equivalent node pairs and missing edges, and sample missing elements from the enhanced ground truth. We produced SynDBYa with size up to $2M$ nodes and $25M$ edges.

We induce missing elements of interests from the corresponding ground-truth pairs in $\Gamma$, which refer to "one-sided" values that occur in only one source (thus missing in another).

*Constraint generation.* We calibrated the mining of data constraints from $\Gamma$ of all the graphs to ensure the following high quality data constraints. (1) Graph keys as NEs. We use the algorithm in [9] to discover keys that cover ground-truth in $\Gamma$. We set support and confidence threshold as 0.8 and 0.9, respectively, and extracted 250, 20 and 12 graph keys, which cover $40K$ (80%), $7.6K$ (80%), and $90K$ (85%) equivalent node pairs in DBYa, DBIM, and OAG, respectively. (2) Graph association rules [12], [13] as EGs. We detect EGs with confidence threshold 0.8 and a smaller support threshold 0.1 to cover missing edges with various types. We extracted in total 750, 25 and 10 EGs for DBYa, DBIM and OAG, respectively. (3) We also generated a set of GFDs [7] as NEs for rule-based entity matching. GFDs subsume graph keys but cannot directly infer missing links.

*Query generation.* We generate queries based on DBpedia SPARQL Benchmark [40] for DBYa. We choose queries to have some answers in the ground truth but not seen in the incomplete graphs, and sample missing elements for tests.

*Algorithms.* We implemented BiExp (Section IV), ApxExp (Section V-A), BiExpQ (Section V-B), and the following.

[1]https://github.com/lgalarra/vickey

[2]https://www.csd.uoc.gr/~vefthym/minoanER/datasets.html

[3]https://www.openacademic.ai/oag/

(1) We compare BiExp with (a) BiExp_N, a variant of BiExp without pruning techniques (PruneFwd and PruneBwd); (b) BiExp_Fwd and BiExp_Bwd, variants of BiExp_N that performs only forward and backward search, respectively; (c) BiExp_NE and BiExp_EG, which access $\Sigma$ that contains only NEs and EGs respectively. Similarly, we compare BiExpQ with BiExpQ_N, BiExpQ_NE and BiExpQ_EG.

(2) We compare ApxExp with (a) ApxExp_N, a variant without forward pruning PruneFwd; (b) ApxExp_NE (resp. ApxExp_EG) which access $\Sigma$ that only contains NEs (resp. EGs). We consider actions with unit cost for these algorithms.

(3) Batch enforcement: NE + EG, which stacks the batch enforcement of NEs followed by EGs, and a reversed counterpart EG + NE that enforces EGs first. These methods simulate entity matching and link prediction over the entire $G$.

Beyond data constraints, we also implemented the following models. (a) TransD + NE first uses TransD [41], a class of embedding based predictive models to infer all missing links, and then enforces NEs. We follow [42] to train TransD via supervised learning. (b) Rule-based link prediction AMIE+ [5] and entity matching Vickey [10]. We apply AMIE+ and Vickey in a batch mode similar to EG + NE and NE + EG.

We adopt VF2 [43] to detect violations (DetVio), and incremental pattern matching [38] to maintain query answers in BiExpQ. It takes on average 8 seconds to detect all the violations for graph patterns with 7 nodes and edges.

*Metrics.* For BiExp (resp. BiExpQ), we report *coverage*, the ratio of the number of missing elements (resp. missing answers) explained by BiExp to the total $N$ requests (resp. queries). For ApxExp, we define the *normalized informativeness gain* as $\frac{\mathsf{cg}(\rho)}{\mathsf{cg}(\rho^*)}$ to measure the closeness between computed explanations and optimal counterpart (obtained by enumeration). To evaluate the applications in graph refinement, we also report standard precision $\frac{|\mathcal{U}\cap\Gamma|}{|\mathcal{U}|}$ and recall $\frac{|\mathcal{U}\cap\Gamma|}{|\Gamma|}$, where $\mathcal{U}$ refers to the elements inferred by the algorithms.

By default, we set $N$=20, a size bound $b$ = 4 for BiExp, a budget $B$=800 as the total number of actions for ApxExp, and $\Sigma$ contains 20 NEs and 20 EGs per test.

*Environment*. All the algorithms are implemented in Java[4]. We ran all our experiments on a Linux machine powered by an Intel 2.4 GHz CPU with 128 GB of memory. We ran each experiment 10 times and reported the averaged results.

**Experimental results.** We next report our findings.

**Exp-1: Coverage of** BiExp. We sampled missing elements to be clarified from ground truth and report the coverage of requests in Figure 8(a). BiExp_N, BiExp_Fwd and BiExp_Bwd (omitted) produce the same results as BiExp. (1) For all cases, BiExp covers on average 91% of the 20 requests with explanations that contain at most 4 actions. BiExp outperforms BiExp_EG (resp. BiExp_NE) by 62% (resp. 65%). We found that 34% of the explanations generated by BiExp have at least 2 actions. These actions can not be derived by a batch enforcement of NEs or EGs. (2) BiExp further improves the coverage by 17% (resp. 27%) compared with NE + EG (resp. EG + NE). It improves the coverage by 33% compared with TransD + NE. These results verify that BiExp effectively explains targeted missing elements with only the necessary amount of inference rather than refining the entire graph.

*The impact of* $|\Sigma|$. Varying $|\Sigma|$ from 10 to 40, we report the coverage of the algorithms in Figure 8(b). The coverage of all algorithms increases as more data constraints are available. This leads to richer semantics of the incomplete graphs, thus is more likely to recover missing elements. On the other hand, the coverage of BiExp increases from 46% to 92% as $|\Sigma|$ varies from 10 to 40, while BiExp_NE and BiExp_EG explain up to 33% and 23% of the missing elements. This shows that BiExp can effectively exploit more data constraints.

*The impact of size bound b.* Using the same setting with 20 requests, we varied $b$ from 1 to 4. Figure 8(c) shows that all algorithms cover more requests over larger $b$. BiExp "degrades" to EG +NE and NE +EG when $b$=1, but effectively exploits useful actions to cover more requests for larger $b$. For $N$ requests with queries and missing elements, the coverage of BiExpQ is close to that of BiExp (thus not shown).

**Exp-2: Efficiency of** BiExp. Using the same setting in Exp-1, we report the efficiency of explanation generation. Figure 8(d) verifies that it is feasible to explain missing elements for large real-world graphs. (1) On average, BiExp outperforms BiExp_N, BiExp_Fwd and BiExp_Bwd by 2.2, 3.5 and 8.9 times respectively due to the pruning strategy. For example, it takes on average 5 seconds per request on OAG with $2.5M$ nodes and $5.2M$ edges. (2) BiExp_N improves the efficiency of BiExp_Fwd and BiExp_Fwd by 1.6 and 4 times with the bidirectional strategy alone. BiExp_Bwd takes the most time, as it explores an excessive number of enablers. (3) BiExp_EG and BiExp_NE take less time compared with BiExp due to less actions triggered with only NEs or EGs.

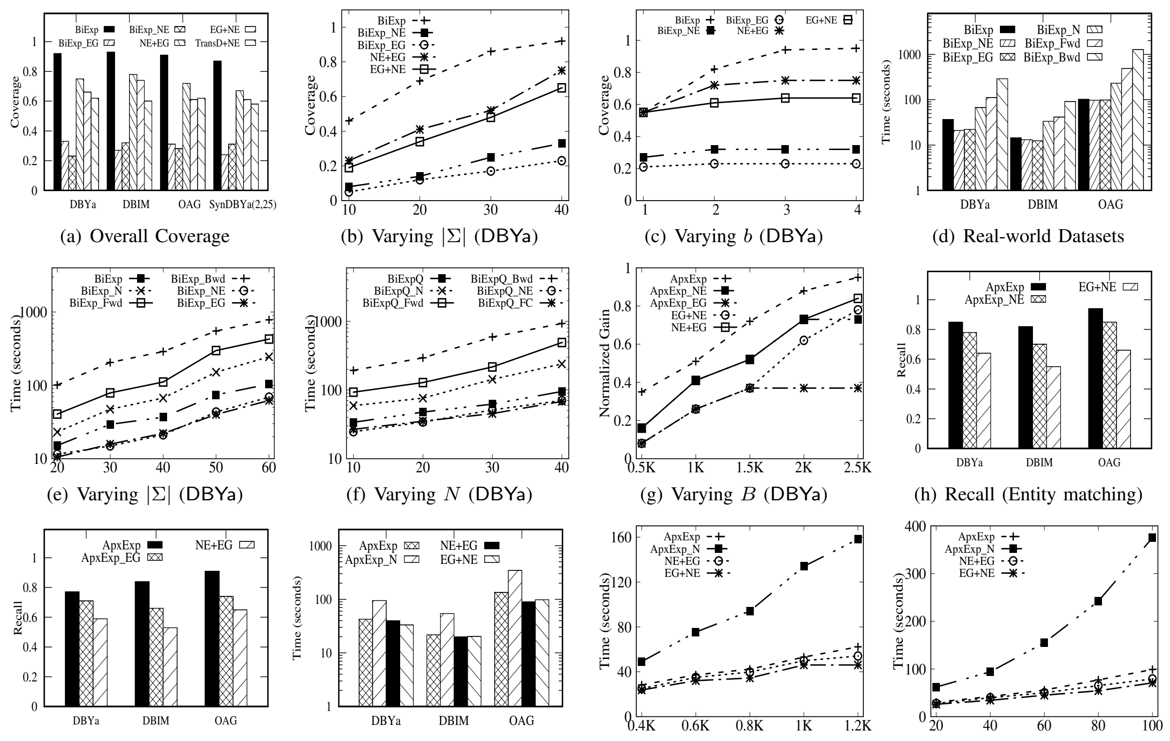
We also evaluate our algorithms over a large synthetic graph SynDBYa. While all the methods scale well, BiExp is the least sensitive among all variants. It takes on average 5.9 per request over graphs of size $(2M, 25M)$. We report the details in [30].

*Varying* $|\Sigma|$. Fixing $N$ = 20 and varying $|\Sigma|$ from 20 to 60, we report the performance over DBYa in Figure 8(e). All the algorithms take more time to explore more actions and data constraints. BiExp takes up to 1.9 seconds to explore 40 data constraints per request.

*Varying N (*BiExpQ*).* Varying $N$ from 10 to 40, we report the performance of BiExpQ in Figure 8(f). As other methods are not designed for explaining missing answers, we report their time for clarifying the missing element. While all algorithms scale well over $N$, BiExpQ takes up to 90 seconds to explain 40 missing query answers (less than 2.3 seconds per query).

**Exp-3: Effectiveness of** ApxExp. To evaluate ApxExp, We set budget $B$ as the total number of actions allowed to be explored (same as the number of calls for oracle; Section V-A). ApxExp_N (omitted) generates the same result.

[4]source code available at: https://github.com/wsu-db/GRIP/


(a) Overall Coverage (b) Varying $|\Sigma|$ (DBYa) (c) Varying $b$ (DBYa) (d) Real-world Datasets

(e) Varying $|\Sigma|$ (DBYa) (f) Varying $N$ (DBYa) (g) Varying $B$ (DBYa) (h) Recall (Entity matching)

(i) Recall (Link prediction) (j) Real-world Datasets (k) Varying $B$ (DBYa) (l) Varying $|\Sigma|$ (DBYa)

Fig. 8: Performance of BiExp and BiExpQ

*The impact of resource bound $B$.* As shown in Figure 8(g) over DBYa, all the algorithms achieve higher informativeness gain when more actions are allowed to be explored. We observe the following. (1) ApxExp achieves the highest gain among all the methods. It converges earlier at the highest gain compared with other methods. This verifies that ApxExp can make "better" decisions early by prioritizing the informative actions. (2) ApxExp_NE (resp. ApxExp_EG) terminates after $B$ =$2K$ (resp. $B < 1.5K$) actions, as it reaches relative completeness instances using NEs (resp. EGs) alone. (3) EG + NE continues to improve the gain and outperforms ApxExp_EG when $B > 1.5K$. We found more node merges are still applicable due to new edges inserted by enforcing EGs.

*Accuracy*. Fixing $B = 2K$, we report the recall (precision not shown) of ApxExp_EG using NEs (resp. EGs) alone for entity matching (resp. link prediction) over relevant ground truth in Figure 8(h) (resp. 8(i)). For entity matching, we omit ApxExp_EG and NE + EG, since EGs do not trigger merging. Similarly, we omit ApxExp_NE and EG + NE. (1) ApxExp improves the recall and precision (not shown) for both entity matching and link prediction, as more missing elements are inferred by interacting NEs and EGs. (2) NE + EG (resp. EG + NE) improves precision of ApxExp_NE (resp. ApxExp_EG) but consumes most of resource for entity matching (resp. link prediction) alone, thus has lower recall.

**Exp-4: Efficiency of** ApxExp. Using the same setting as in Exp-3, Figure 8(j) verifies that ApxExp is feasible to refine large graphs (*e.g.,* 130 seconds for OAG with 800 actions). It outperforms ApxExp_N by 2.36 times on average due to the online prioritization and pruning strategy. The pruning effectively reduces 76% of graph homomorphism (isomorphism) verifications. While ApxExp performs more verifications than NE + EG and EG + NE, it incurs a comparable time cost.

*The impact of budget $B$.* Figure 8(k) reports the the time cost of the algorithms up to the exploration of $B$ actions. All the algorithms take longer time with larger $B$. ApxExp is quite feasible in budgeted scenario. It incurs a delay time on average 0.25 (resp. 0.19) second per batch, and 42 (resp. 22) seconds to explore 800 actions over DBYa (resp. DBIM). EG + NE terminates when $b > 1K$ and exhausts applicable actions. In contrast, ApxExp is able to exploit more and useful actions.

*Varying $|\Sigma|$ and $|G|$*. Figure 8(l) verifies that all algorithms scale well over $|\Sigma|$. Specifically, ApxExp takes 98.9 seconds with 100 constraints and is 3.8 times faster than ApxExp_N. ApxExp also scales well with $|G|$, and outperforms ApxExp_N by 3.9 times when $|G|$ is varied to $(2M, 25M)$ (see [30]).

**Comparison with other models.** Using the same setting as in Exp-1, we run AMIE+ and Vickey in a batch mode to infer missing elements over DBYa. We found that BiExp improves the coverage of AMIE+ + Vickey (resp. Vickey + AMIE+) by 22% (resp. 13%), is 3.6 (resp. 3.8) times faster, and triggers smaller number of operators. For example, it takes on average 265 operators with 40 constraints, while AMIE+ + Vickey requires 874 operators using 20 AMIE+ rules and 20 Vickey keys. We also compare BiExp with GFD + EG over DBYa, using 20 GFDs and 20 EGs. We find that BiExp outperforms GFD + EG by 35% on coverage with comparable time cost.

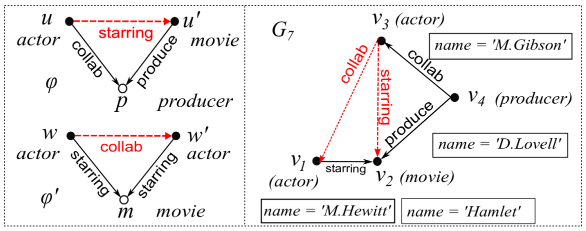


Fig. 9: Clarifying inaccurate elements: case study.

**Exp-5: Case Study.** We show that BiExp can also be naturally used to clarify the occurrence of erroneous elements that are inferred by the constraints. Fig. 9 illustrates a fragment of DBIM ($G_7$) with two discovered graph association rules:

($\varphi$):*"an **actor** ($u$) **stars** in a **movie** ($u'$) if a **producer** ($p$) he collaborates also produces the same movie"*

($\varphi'$): *"an **actor** ($w$) **collaborates** with an **actor** ($w'$) if they both starred a movie ($m$)."*

The insertion of an edge collab $(v_1, v_3)$ is annotated as "inaccurate" (*'M.Gibson' and 'M.Hewitt' collaborated in a movie*). BiExp generates an explanation, which states that an edge insertion $\oplus((v_3, v_2), \text{starring})$ (by enforcing $\varphi$) leads to the inaccurate element collab $(v_1, v_3)$ (by enforcing $\varphi'$). A closer inspection suggests that $\varphi$ can be an "overkill", given the exception of $(v_3, v_2)$ ("an actor may not always be starring a movie produced by a producer he collaborated with").

## VII. Conclusion

We have introduced a constraint-based approach to clarify missing elements with established graph data constraints. We have formulated the measurements and explanation problems. We have developed bidirectional algorithms with quality guarantees. We have verified that constraint-based methods can effectively and efficiently clarify targeted missing values with necessary inference. One future topic is to study parallel algorithms for constraint-based methods. Another topic is to extend our work to soft constraints (*e.g.,* probabilistic soft logic) to clarify erroneous attribute values in graph data.

**Acknowledgments.** This work is supported by NSF under CNS-1932574, OIA-1937143, ECCS-1933279, CNS-2028748, DoE under DE-IA0000025, USDA under 2018-67007-28797, and PNNL Data-Model Convergence initiative.

## References

[1] H. Paulheim, "Knowledge graph refinement: A survey of approaches and evaluation methods," *Semantic web*, vol. 8, no. 3, pp. 489–508, 2017.
[2] X. L. Dong, E. Gabrilovich, G. Heitz, W. Horn, K. Murphy, S. Sun, and W. Zhang, "From data fusion to knowledge fusion," *PVLDB*, 2014.
[3] A. Arioua and A. Bonifati, "User-guided repairing of inconsistent knowledge bases," in *EDBT: Extending Database Technology*, 2018.
[4] M. H. Namaki, Q. Song, Y. Wu, and S. Yang, "Answering why-questions by exemplars in attributed graphs," in *SIGMOD*, 2019.
[5] L. A. Galárraga, C. Teflioudi, K. Hose, and F. Suchanek, "Amie: association rule mining under incomplete evidence in ontological knowledge bases," in *WWW*, 2013.
[6] W. Fan, Z. Fan, C. Tian, and X. L. Dong, "Keys for graphs," *PVLDB*, vol. 8, no. 12, 2015.
[7] W. Fan, Y. Wu, and J. Xu, "Functional dependencies for graphs," in *SIGMOD*, 2016.
[8] L. Caruccio, V. Deufemia, and G. Polese, "Relaxed functional dependencies—a survey of approaches," *TKDE*, vol. 28, no. 1, 2016.
[9] A. Morteza and C. Fei, "Keyminer: Discovering keys for graphs," in *VLDB workshop*, 2018.
[10] D. Symeonidou, L. Galárraga, N. Pernelle, F. Saïs, and F. Suchanek, "Vickey: mining conditional keys on knowledge bases," in *ISWC*, 2017.
[11] H. Ma, M. Alipourlangouri, Y. Wu, F. Chiang, and J. Pi, "Ontology-based entity matching in attributed graphs," *PVLDB*, 2019.
[12] W. Fan, X. Wang, Y. Wu, and J. Xu, "Association rules with graph patterns," *PVLDB*, 2015.
[13] P. Lin, Q. Song, J. Shen, and Y. Wu, "Discovering graph patterns for fact checking in knowledge graphs," in *DASFAA*, 2018.
[14] G. Fan, W. Fan, Y. Li, P. Lu, C. Tian, and J. Zhou, "Extending graph patterns with conditions," in *SIGMOD*, 2020.
[15] J. Xu, W. Zhang, A. Alawini, and V. Tannen, "Provenance analysis for missing answers and integrity repairs." *IEEE Data Eng. Bull.*, vol. 41, no. 1, pp. 39–50, 2018.
[16] I. Bhattacharya and L. Getoor, "Entity resolution in graphs," *Mining graph data*, 2006.
[17] M. Pershina, M. Yakout, and K. Chakrabarti, "Holistic entity matching across knowledge graphs," in *Big Data*, 2015.
[18] D. Firmani, B. Saha, and D. Srivastava, "Online entity resolution using an oracle," *PVLDB*, 2016.
[19] J. Pujara, H. Miao, L. Getoor, and W. Cohen, "Knowledge graph identification," in *ISWC*, 2013.
[20] W. Fan, P. Lu, C. Tian, and J. Zhou, "Deducing certain fixes to graphs," *PVLDB*, 2019.
[21] F. N. Afrati and P. G. Kolaitis, "Repair checking in inconsistent databases: algorithms and complexity," in *ICDT*, 2009.
[22] Y. Cheng, L. Chen, Y. Yuan, and G. Wang, "Rule-based graph repairing: Semantic and efficient repairing methods," in *ICDE*, 2018.
[23] B. Kimelfeld, E. Livshits, and L. Peterfreund, "Detecting ambiguity in prioritized database repairing," in *ICDT*, 2017.
[24] P. Buneman, S. Khanna, and T. Wang-Chiew, "Why and where: A characterization of data provenance," in *ICDT*, 2001.
[25] A. Meliou, W. Gatterbauer, K. F. Moore, and D. Suciu, "Why so? or why no? functional causality for explaining query answers," *arXiv*, 2009.
[26] L. Libkin, "Certain answers meet zero-one laws," in *PODS*, 2018.
[27] B. Glavic, S. Köhler, S. Riddle, and B. Ludäscher, "Towards constraint-based explanations for answers and non-answers," in *TAPP*, 2015.
[28] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of databases: the logical level*, 1995.
[29] W. Fan and P. Lu, "Dependencies for graphs," in *PODS*, 2017.
[30] Full version. *https://songqi1990.github.io/Files/paper/icde2021full.pdf.*
[31] L. Galárraga, S. Razniewski, A. Amarilli, and F. M. Suchanek, "Predicting completeness in knowledge bases," in *WSDM*, 2017.
[32] S. Guo, X. L. Dong, D. Srivastava, and R. Zajac, "Record linkage with uniqueness constraints and erroneous values," *PVLDB*, 2010.
[33] W. Fan, "Dependencies for graphs: Challenges and opportunities," *JDIQ*, vol. 11, no. 2, pp. 1–12, 2019.
[34] N. D. Jones, "An introduction to partial evaluation," *ACM Comput. Surv.*, vol. 28, no. 3, p. 480–503, 1996.
[35] S. E. Whang, D. Marmaros, and H. Garcia-Molina, "Pay-as-you-go entity resolution," *TKDE*, vol. 25, no. 5, pp. 1111–1124, 2012.
[36] M. Babaioff, N. Immorlica, D. Kempe, and R. Kleinberg, "A knapsack secretary problem with applications," in *Approximation, randomization, and combinatorial optimization*, 2007, pp. 16–28.
[37] Y. Zhou, D. Chakrabarty, and R. Lukose, "Budget constrained bidding in keyword auctions and online knapsack problems," in *WINE*, 2008.
[38] W. Fan, X. Wang, and Y. Wu, "Incremental graph pattern matching," *TODS*, vol. 38, no. 3, pp. 1–47, 2013.
[39] V. Efthymiou, K. Stefanidis, and V. Christophides, "Benchmarking blocking algorithms for web entities," *IEEE Trans. on Big Data*, 2016.
[40] M. Morsey, J. Lehmann, S. Auer, and A.-C. N. Ngomo, "Dbpedia sparql benchmark–performance assessment with real queries on real data," in *ISWC*, 2011.
[41] G. Ji, S. He, L. Xu, K. Liu, and J. Zhao, "Knowledge graph embedding via dynamic mapping matrix," in *ACL*, 2015.
[42] P. Lin, Q. Song, Y. Wu, and J. Pi, "Discovering patterns for fact checking in knowledge graphs," *JDIQ*, 2019.
[43] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, "A (sub) graph isomorphism algorithm for matching large graphs," *TPAMI*, 2004.