Answering Why-Questions for Subgraph Queries in Multi-Attributed Graphs

Qi Song*, Mohammad Hossein Namaki*, Yinghui Wu*†
*Washington State University, USA
†Pacific Northwest National Laboratory, USA
{qi.song, m.namaki, yinghui.wu}@wsu.edu

***Abstract*—Subgraph queries have been routinely used to search graphs *e.g.,* social networks and knowledge bases. With little knowledge of underlying data, users often need to rewrite queries multiple times to reach desirable answers. Why-questions are studied to explain missing (as "*Why-not*" questions) or unexpected answers (as "*Why*" questions). This paper makes a first step to answer why-questions for subgraph queries in attributed graphs. (1) We approach query rewriting and construct *query rewrites*, which modify original subgraph queries to identify desired entities that are specified by Why-questions. We introduce measures that characterize good query rewrites by incorporating both query editing cost and answer closeness. (2) While computing optimal query rewrite is intractable for Why-questions, we develop feasible algorithms, from exact algorithms to heuristics, and provide query rewrites with (near) optimality guarantees whenever possible, for both Why and Why-not questions. These algorithms dynamically select "picky" operators that ensure to change (estimated) answers closer to desired ones, and incur cost determined by the size of query results and questions only. We also show that these algorithms readily extend to other Why-questions such as Why-empty and Why-so-many. Using real-world graphs, we experimentally verify that our algorithms are effective and feasible for large graphs. Our case study also verifies their application in *e.g.,* knowledge exploration.**[1]

***Keywords*-Graph Queries; Why-question; Query Rewriting**

I. Introduction

Subgraph queries have been applied to access and understand complex networks such as knowledge bases [1] or social networks [2]. Given a graph G, a *subgraph query* Q is a (labeled) graph pattern, and the *answer* $Q(G)$ of Q in G refers to subgraphs of G that are relevant to Q. The answer $Q(G)$ can be defined by subgraph isomorphism [1], [3] or approximate pattern matching [4], [5].

Writing queries to search large and heterogeneous graphs is nevertheless a nontrivial task for end users. With little prior knowledge of data, users often need to revise the queries multiple times to find desirable answers. An explain functionality supported by query rewriting is thus desirable to help users understand the unexpected answers. Specifically, users may want to ask two classes of *Why-questions*:

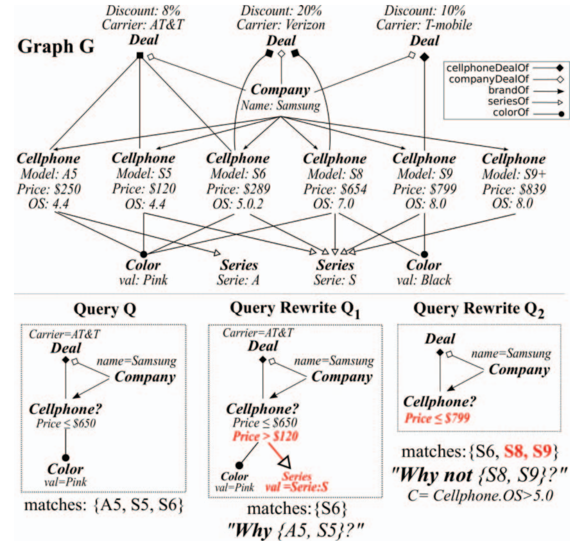


Figure 1: Why and Why-not questions for subgraph queries.

- ◦ (1) **Why question**: "*why some (unexpected) entities are in the query answer?*"; and
- ◦ (2) **Why-not question**: "*why certain entities are missing from the query result?*"

Answering these questions can help users to tune their queries towards desirable answers, which further supports effective graph exploration and query provenance [6].

Example 1: Fig 1 illustrates a fraction of a multi-attributed knowledge graph G about products of an online store. Each entity carries a type *e.g.,* Cellphone and a list of attributes (*e.g.,* Price) with corresponding values (*e.g.,* "$250"). A user wants to search for Samsung cellphones packed with color pink and carrier $AT\&T$, with price less than $650 (represented by a subgraph query Q , with a marked node "Cellphone?" designating the targeted entities). The query returns three entities that match "Cellphone" in Q, with Samsung models $A5$, $S5$, and $S6$, respectively. To further explore the product, the user may ask the following questions.

(1) She is surprised to see two unexpected entities, with older models $S5$ and $A5$ are returned. She may pose a follow-up *Why* question that specifies search focus "Cellphone" and the two entities and asks "*Why $S5$ and $A5$ are in the query result of Q?*" An

[1]This is the full version of paper "Answering Why-Questions for Subgraph Queries in Multi-Attributed Graphs" accepted by IEEE International Conference on Data Engineering (ICDE) 2019

2375-026X/19/$31.00 ©2019 IEEE
DOI 10.1109/ICDE.2019.00013

IEEE computer society

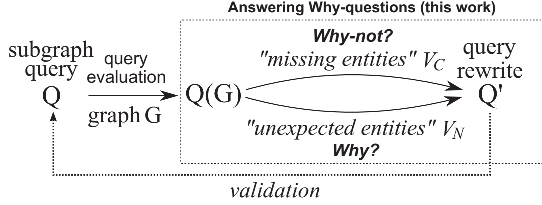


Figure 2: Enabling exploratory graph search with Why-questions.

answer to such a question can be a revised query Q_1 which properly "tighten" the constraints in Q to exclude $\{S5, A5\}$.

Observing the difference between Q_1 and Q, a possible explanation for the Why question reveals that the user may not be interested in Series A (due to a newly inserted edge that specifies "Serie S") and older (cheaper) versions of Series S (as a new lower bar of price $120 is added).

(2) She is also wondering why two more recent models, $S8$ and $S9$, are not included in the current results. She may pose a follow-up *Why-not* question, which asks "*Why model $S8$ or $S9$ are not in the query result?*" An answer to this question can be another query Q_2, which properly "relax" the conditions on the query , to include the two entities. The difference between Q_2 and Q reveals that 1) both recent models are more expensive than expected (as the price is relaxed to $790 in Q_2), 2) there is no pink $S9$ model, and no evidence shows that they are supported by $AT\&T$ (both constraints are removed in Q_2). □

These motivate the need for developing effective query rewriting techniques to answer *Why-* questions for *subgraph query*. Specifically, given a data graph G, a subgraph query Q and answer $Q(G)$, a set of *missing entities* V_C with possible additional value constraints (resp. a set of *unexpected entities* V_N in $Q(G)$), the problem is to modify Q to a "query rewrite" Q', such that $Q'(G)$ contains the entities in V_C (resp. excludes the entities in V_N) as much as possible.

The need of answering Why-questions is evident in exploratory graph search, as illustrated in Fig. 2. (1) The query rewrite Q' can be readily suggested to enable iterative query processing, upon receiving users' feedback on missing or unexpected entities. (2) The difference between the query rewrite Q' and its original counterpart Q blends visual querying and approximate search when G is large [7]. (3) Query rewrites also support graph exploration [6], [8] by suggesting entities relevant to missing and unexpected ones.

Contributions. We make a first step to answer Why-questions for subgraph queries in multi-attributed graphs.

(1) We formalize Why-questions for subgraph queries in terms of graph query rewrites (Section III-A). Given a subgraph query Q, graph G, and answers $Q(G)$, a *Why-not* question asks whether there exists a query rewrite Q' with answers that contains $Q(G)$ and a set of entities not in $Q(G)$, and satisfy certain value constraints C. A *Why* question, on the other hand, finds Q' that revises Q to have answers that exclude specified entities from $Q(G)$ (Section III-A).

(2) To characterize "good" query rewrites, we introduce two measures, *answer closeness* and *query editing cost*, to measure the closeness of query rewrite answers and desired ones, and query rewrites to original queries, respectively (Section III-C). The cost models are defined in terms of a set of *relaxation* and *refinement* operators, aiming to tune the search conditions of Q towards desired answers. Based on the cost model, we formalize the problem of answering Why-questions, which is to compute query rewrite Q' with optimized answer closeness given desired entities from Why-questions, under a budget B of query editing cost. We show that the problem is NP-hard for answering both Why and Why-not questions.

(3) Despite the hardness, we develop both exact algorithms that compute optimal query rewrites, and their fast alternatives to compute high-quality query rewrites.

(a) For **Why** questions (Section IV), we develop an exact algorithm to compute optimal query rewrite. The algorithm uses a partial enumeration scheme that requires only the verification of a class of maximal bounded operator set, instead of all possible query rewrites. In addition, we develop an approximation algorithm that achieves a near-optimality guarantee $\frac{1}{2}(1\text{-}\frac{1}{e})\text{-}f(\epsilon)$, where $f(\epsilon)$ is a function determined by an estimation error ϵ of the quality of query rewrites.

(b) For **Why-not** questions (Section V), we show a counterpart of the sufficient and necessary condition for Why questions also holds, and develop an exact algorithm. We also develop a fast heuristic that computes high-quality query rewrites without any subgraph isomorphism test.

All these algorithms incur a cost that is only determined by Q and its answer, size of user-specified entities in Why-questions and editing budget, which are small in practice.

(4) Using real-world graphs, we experimentally verify the effectiveness and efficiency of our algorithms (Section VI). These algorithms are feasible for large graphs. For example, our exact algorithms take up to 56 seconds to compute optimal query rewrite for queries with 10 edges and literals, and for 3 missing/unexpected entities. Better still, their approximation (resp. heuristic) counterpart is 9.7 times (resp. 15 times) faster, take up to 10 seconds (resp. 11), with up to 12% (resp. 16%) loss of quality. They return informative query rewrites for *e.g.,* exploring knowledge bases, as verified by our case study.

These results yield effective methods toward feasible data exploration in large graphs, fast query refinement and completion, and graph data provenance techniques. All the proofs and complexity analysis can be found in [9].

Related work. We categorize the related work as follows.

Why-questions for relational data. Why and Why-not queries have been studied for relational queries [10]–[15]. There are typically two methods: (1) Data provenance modifies data such that the missing (resp. unexpected) answers appear (resp. disappear) in the modified database; and (2)

Query manipulation identifies the relational operators that eliminate specific tuples [10] (for Why questions) or introduce new ones [11], and update queries accordingly [15]. Such techniques have been studied for (reversed) top-k [12], [14] and reverse skyline queries [13]. These work cannot be directly applied for subgraph queries over general multi-attributed graphs. We approach query rewriting to answer Why-questions for subgraph queries.

Why-questions for graphs. Why-not questions have been studied to support exploratory graph search, for both structured queries, such as SPARQL [16] and unstructured queries, such as keyword queries [17] and pattern matching [18]–[21]. (1) Similar to relational queries, SPARQL queries are decomposed into basic operators and triple patterns [16]. Operators that lead to empty answers are identified, and relaxed to include more entities of interests. (2) To explain empty answers for keyword queries in XML data, terms with no content nodes included in the answers are identified [17]. These terms are then replaced by their counterparts with content nodes with similar semantics. (3) To enrich the query results with more (diversified) matching graphs, the query graph is reformulated into supergraphs with maximized diversity [19]. Maximal common subgraphs between queries and data graphs are computed to explain "too many" or "too few" matched graphs [20], [21].

Closer to our work is answering Why-not queries in graph databases [18]. Given a graph database, a query graph and a set of missing graphs, the "Why-not" question aims to find a subgraph by adding/deleting edges, such that the missing graphs that contain it can be included in the answer set. Our work differs from the prior work as follows. (1) We focus on entity search with subgraph queries in single attributed graphs. This is a different task with searching graph databases that return all isomorphic (sub) graphs. (2) Our methods support a rich set of structural and semantic editing operators, not limited to edge insertion and deletion only [17], [18]. Moreover, we consider practical value constraints and cost models that yield more intuitive explanations for entity search in graphs. (3) We develop algorithms with provable performance guarantees for Why-questions. These are not addressed by prior work.

Exploratory search. Exploratory search is commonly used to help users to explore unfamiliar data [22], [23]. (1) Query completion constructs queries progressively by suggesting top-k possible query fragments that can be added into the query, where the fragments are discovered as frequent subgraphs [22]. Instead, we discover "picky" structures and operators to construct new queries for users. Indeed, frequent subgraphs are not necessarily helpful to acquire specific entities. (2) Query generation by example learns how to generate queries from a set of positive and negative examples [23], which may be "missing" and "unexpected" entities. The approach in [23] is developed for relational queries only, and does not apply for general subgraph queries. (3) Querying by example [24] finding similar answers to specified examples rather than computing queries that identify them. Why-questions cannot be directly answered by these methods.

II. Graphs and Subgraph Queries

We start with the notions of graphs and subgraph queries.

Graphs. We consider a directed graph $G = (V, E, L, F_A)$, where V is a finite set of nodes and $E \subseteq V \times V$ is a set of edges. Each node $v \in V$ (resp. edge $e \in E$) carries a label $L(v)$ (resp. $L(e)$). $F_A(v)$ is a tuple $<(A_1, a_1),\ldots,(A_n, a_n)>$, where A_i is a node attribute, and constant $a_i \in D(A_i)$ is the value of attribute $v.A_i$. Here $D(A_i)$ is a finite *active domain* of attribute A_i, which records all the values of $v.A_i$ appeared in G with node v ranges over V. In practice, the node and edge label may represent type and relation (predicates), respectively, and $F_A(v)$ may encode the node content such as properties in *e.g.,* social networks and knowledge bases.

We introduce a class of subgraph queries with *output nodes* for practical entity search in graphs.

Subgraph queries. A subgraph query Q is a graph $(V_Q, E_Q, L_Q, F_Q, u_o)$, where (1) V_Q (resp. $E_Q \subseteq V_Q \times V_Q$) is a set of query nodes (resp. query edges); (2) For each node $u \in V_Q$ with a label $L_Q(u)$, $F_Q(u)$ is a predicate that contains a set of literals. Each literal is in the form of $u.A$ op c, where op is a comparison operator from the set $\{>, \geq, =, \leq, <\}$ defined on the domain of attribute A, and c is a constant. Specifically, there is a designated *output node* $u_o \in V_Q$. The output node indicates the "search focus", for which the matched entities are returned as answers of Q.

For simplicity, we refer to subgraph query as "query". We refer to the *size* of query Q, denoted as $|Q|$, as the sum of the number of literals and edges of Q.

Query answer. Consider a query node u in a query Q with label $L_Q(u)$ and predicate $F_Q(u)$. A node v in G is a *candidate* of u if (1) $L(v)$=$L_Q(u)$, and (2) for each literal $l \in F_Q(u)$ in the form of $u.A_i$ op c, $(v.A_i, a_i) \in F_A(v)$ and $a_i \mathsf{op}$ c. For example, given a literal $l='u.A_i \leq c'$, then $(A_i, c) \in F_A(v)$ and $c \leq c'$, for any candidate v of u.

Given query $Q = (V_Q, E_Q, L_Q, F_Q, u_o)$ and a graph G, a *match* of Q in G is a subgraph $G_s = (V_s, E_s, L, F_A)$ of G, such that there exists a bijective matching function $h \subseteq V_Q \times V_s$, where (1) for each node $u \in V_Q$, $h(u)$ is a candidate of u, and (2) $e = (u, u')$ is an edge in Q if and only if $e' = (h(u), h(u'))$ is an edge in G_s and $L(e) = L(e')$. Given a query node $u \in V_Q$, the *matches* of u, denoted as $Q(u, G)$, refers to the set of all the nodes in G that can match node u via a matching function $h(u)$ from Q to G. We define the *answer* of Q in G as $Q(u_o, G)$, *i.e.,* the matches of the output node u_o of Q in G ($Q(u_o, G) \subseteq V$).

Remarks. Queries with output nodes are commonly used to identify entities in *e.g.,* social networks [25] and knowledge

Notation	Description
G=(V, E, L, F_A)	attributed graph G
Q=$(V_Q, E_Q, L_Q, F_Q, u_o)$	subgraph query Q, with output node u_o
$Q(u_o, G)$	query answer of Q in G
(u_o, V_{C_u}, C)	Why-not (w. missing nodes V_{C_u}, constraints C)
(u_o, V_{N_u})	Why question (w. unexpected nodes V_{N_u})
Q'=$Q \oplus O$	A query rewrite induced by operator set O
$c(O)$	Editing cost of operator set O
$\mathsf{cl}(O)$	Closeness for Why-questions
$N_d(V_{N_u})$	d-hop neighbors of the nodes in V_{N_u}

Table I: Notations

graphs [26]. To simplify the discussion, we focus on a single output node. Nonetheless, our results extend to multiple ones (Section V). We invite the readers to consult [9] for details.

Example 2: The query Q illustrated in Fig. 1 has an output node u_o with label Cellphone, which indicates that the user searches for cell phones. Given the graph G, one can verify that the answer $Q(u_o, G)$ = $\{A5,\ S5,\ S6\}$. We abuse the model value of a Cellphone as its identifier for simplicity; similarly for the answer of queries Q_1 and Q_2. □

III. Why-Questions for Subgraph Queries

A. Categorization of Why-Questions

We study two classes of Why-questions, "*Why-not*" and "*Why*", for subgraph queries. These questions have their counterparts in relational query provenance [10]–[15], and are specialized for graph search.

Why-not. A Why-not question aims to clarify missing answers in $Q(u_o, G)$. Given a graph G = (V, E, L, F_A), a query Q = (V_Q, E_Q, L_Q, u_o), answer $Q(u_o, G)$, a Why-not question is a triple (u_o, V_{C_u}, C), where (1) u_o is the output node of Q, (2) $V_{C_u} \subseteq V \setminus Q(u_o, G)$ is a set of "missing matches" of u_o, and (3) C is a (possibly empty) selection condition to further describe the expected entities of interests (discussed later). In practice, V_{C_u} can be directly specified, or identified by established keyword search in graphs [8].

Beyond encoding user feedback as missing nodes V_{C_u}, a user-defined selection condition C can be posed on V_{C_u} and $Q(u_o, G)$ to further describe expected entities in a similar way to enforce predicates in Q. Formally, C = $\bigwedge l$ is a conjunction of literals, where each literal l is in the form of either $x.A$ op c or $x.A$ op $y.A$ (op $\in \{<, \leq, =, >, \geq\}$), and x and y are variables that can be matched by nodes from $V_{C_u} \cup Q(u_o, G)$). As such, C can be used to refine V_{C_u} (as constraints), or retrieve lineage information (as a query) [15].

A general Why-not question asks "*Why the nodes in V_{C_u}, with attribute values that satisfy the value constraints in C (if not empty), are not matches of u of Q?*"

Why. A *Why question* (u_o, V_{N_u}) is similarly defined on a targeted output node u_o of Q, yet on a set of unexpected answers $V_{N_u} \subseteq Q(u_o, G)$, for a given query Q. It asks "*Why the nodes in V_{N_u} are included as matches for u in G?*"

Example 3: A Why question (Cellphone, $\{A5, S5\}$) (illustrated in Fig. 1) asks "why $Q(u_o, G)$ includes cell phones with model $A5$ and $S5$?" Similarly, a Why-not question (Cellphone, $\{$S8, S9$\}$, $\{$Cellphone.OS$\geq 5.0\}$) asks "*Why are the cell phones with models $S8$ and $S9$ not in $Q(u_o, G)$?*" Here the constraint C further specifies that only the cell phones with newer OS are of interests. □

B. Answers for Why-Questions

We approach query refinement [15] to answer Why-questions. We start with a set of query editing operators. We then introduce query rewrites based on these operators.

Query rewrites. We use six classes of primitive query editing operators. These operators either relax or refine search constraints of a given query Q, for *any* graph.

Relaxation operators. These include:

- RxL ($u.A$ op c, $u.A$ op' c'): relax the literal ($u.A$ op c)$\in F_Q(u)$ to $u.A$ op' c';
- RmL (u, l): remove a literal l from $F_Q(u)$; and
- RmE (u, u'): remove an edge e=(u, u') in Q.

Refinement operators. These include:

- RfL ($u.A$ op c, $u.A$ op' c'): refine a literal ($u.A$ op c) $\in F_Q(u)$ to $u.A$ op' c', such that u has fewer candidates;
- AddL ($u.A$ op c): add literal ($u.A$ op c) to $F_Q(u)$; and
- AddE (u, u'): add a new edge e=(u, u') . More specifically, (a) if nodes u and u' are both in Q, AddE (u, u') adds a new edge with edge label; (b) assume *w.l.o.g.* the node u' is not in Q, then AddE (u, u') creates u' with specific labels/literals (see Sections IV-V).

We shall refer to RxL, RfL, RmL, and AddL (resp. RmE and AddE) as *node operators* (resp. *edge operators*), as they involve a single query node (resp. query edge).

A *query rewrite* Q' of Q is a query obtained by applying an operator set O to Q (denoted as Q'=$Q \oplus O$). Note that the query rewrites preserve the output node of Q, due to unlikely change of user's search focus in follow-up Why-questions.

Answering Why-questions. Given query Q, $Q(u_o, G)$ and graph G, a query rewrite Q' of Q is an *answer* of a Why-not question (u_o, V_{C_u}, C), if (1) $Q'(u_o, G) \cap V_{C_u} \neq \emptyset$, and (2) $Q'(u_o, G)$ satisfies all constraints of C. That is, $Q'(u_o, G)$ contains at least a "missing" match that satisfies C. Similarly, it answers a Why-question (u_o, V_{N_u}) if there exists at least a node $v \in V_{N_u}$ such that $v \notin Q'(u_o, G)$. That is, Q' excludes unexpected match v.

Query rewrites provide a natural way to explain answers. The editing operators that modify Q to Q' suggest query manipulations that are responsible to "missing" or "unexpected" answers, as verified by the result below.

Lemma 1: *Given a query Q with output node u_o, for any query rewrite Q'=$Q \oplus O$ and any graph G, $Q(u_o, G) \subseteq Q'(u_o, G)$ (resp. $Q'(u_o, G) \subseteq Q(u_o, G)$), if O contains relaxation (resp. refinement) operators only.* □

As a first step towards answering Why-questions for subgraph queries, we shall use relaxation (resp. refinement)

operators to answer Why-not (resp. Why) questions. We defer the case that uses arbitrary operators in future work.

C. Measures of Subgraph Query Rewrites

We next introduce cost models for operator set that modifies query Q to Q', to characterize "good" query rewrites.

Query editing cost. A query rewrite Q' should be similar to the original query Q, thus more likely to preserve $Q(u_o, G)$ and incur less recognition effort. Intuitively, operators O that modify more "important" fraction (closer to u_o) of Q should be more "expensive". For example, changing "price" on output node Cellphone of Q (Fig. 1), should be more expensive than altering its neighbor Color as adjectives.

Output centrality. Given Q with output node u_o, denote the diameter of Q as d_Q, the *output centrality* of a query node u' in Q, denoted as $\mathsf{oc}(u')$, is computed as

$$\mathsf{oc}(u', u_o) = \frac{d_Q}{d(u', u_o) + 1}$$

where $d(u', u_o)$ is the distance from u' to the output node u_o. The output centrality is a normalized closeness centrality in terms of u_o. Intuitively, the "closer" u' to u_o, the more "important" u' is in Q (as observed in concept closeness [27]).

Given query rewrite Q'=$Q \oplus O$, the cost of O, simply denoted by $c(O)$ (for given Q), is defined as

$$c(O) = \sum_{o \in O} c(o),$$

where the cost $c(o)$ for a single operator $o \in O$ is defined as: (1) if o is a node operator posed on node u, $c(o)$ = $\mathsf{oc}(u, u_o)$; (2) if o is an edge operator on edge e=(u, u'), $c(o)$ = $\min(\mathsf{oc}(u, u_o), \mathsf{oc}(u', u_o))$.

Remarks. Our output centrality model extends to a case that incorporates the number of changes to literals with comparable constant values. For example, relaxing "price≤ 500" to "price≤ 1000" should be more expensive than changing it to "price≤ 600". For operator o as either RxL or RfL, it can be treated as "removing a literal" (with a unit cost) and "add a new one" (that accounts for value difference). Thus $c(o)$ = $w(o) \cdot \mathsf{oc}(u, u_o)$ and $w(o)$ can be defined as $1 + \frac{|c'-c|}{\mathsf{range}(D(A))}$, where $\mathsf{range}(D(A))$ is the range of the active domain $D(A)$ (Section II) of attribute A in G.

The editing cost penalizes operators that are "closer" to the output node (and introduce larger difference to value constraints when possible). The smaller $c(O)$ is, the better. Our general cost model and techniques apply to disconnected query rewrites (due to RmE; see Section IV and V).

Answer closeness. The query rewrites should also ensure answers that are as close as the desired ones. We take a pragmatic dichotomous measure to distinguish and quantify the impact of (1) entities that "should be" excluded or included suggested by Why-questions, and (2) entities "unnecessarily" removed or introduced by a guard condition. $c(O)$ measures the difference between Q and Q' while closeness measures the difference between their answers.

Closeness for Why-questions. Given $Q(u_o, G)$ and a Why-question, the *answer closeness* $\mathsf{cl}(\cdot)$ of a query rewrite Q' = $Q \oplus O$ is defined accordingly as follows.

(1) For a Why question (u_o, V_{N_u}),

$$\mathsf{cl}(O, V_{N_u}) = \frac{|(Q(u_o, G) \setminus Q'(u_o, G)) \cap V_{N_u}|}{|V_{N_u}|}$$

which measures the fraction of V_{N_u} that are excluded from $Q'(u_o, G)$; the more, the better.

(2) For a Why-not question (u_o, V_{C_u}, C),

$$\mathsf{cl}(O, V_{C_u}) = \frac{|Q'(u_o, G) \cap V_{C_u}|}{|V_{C_u}|}$$

which measures the fraction of new matches in V_{C_u} that are introduced in $Q'(u_o, G)$; the more, the better. We denote the answer closeness as $\mathsf{cl}(O)$.

Guard conditions. To further penalize the entities "unnecessarily" added or removed from original answer $Q(u_o, G)$, we set a *guard condition* for both Why and Why-not questions.

(1) To penalize the nodes unnecessarily excluded from $Q(u_o, G)$ for Why questions, we pose a guard condition as

$$|(Q(u_o, G) \setminus Q'(u_o, G)) \cap (Q(u_o, G) \setminus V_{N_u})| \leq m$$

That is, we only consider query rewrites that exclude no more than m nodes from the "desired" fraction of $Q(u_o, G)$, where $m \leq |Q(u_o, G)|$ can be set by users.

(2) Similarly, for Why-not questions, a *guard condition* is posed for any operator set O to ensure

$$|Q'(u_o, G) \setminus (Q(u_o, G) \cup V_{C_u})| \leq m$$

That is, only query rewrites that introduce no more than m "undesired" matches are considered to be valid.

Example 4: The query Q_1 (Fig. 1) answers the Why question in with operators $O_1 = \{\mathsf{AddL}(\mathsf{Cellphone.Price} > \$120), \mathsf{AddE}(\mathsf{Cellphone}, \mathsf{Series}), \mathsf{AddL}(\mathsf{Series.Serie} = \mathsf{S})\}$ with answer closeness $\mathsf{cl}(O_1, \{A5, S5\})$=1. Given d_Q=2 and output node Cellphone, the total cost $c(O_1)$ is 4. Similarly, we can verify that the query Q_2 answers the Why-not question with operators $O_2 = \{\mathsf{RmE}(\mathsf{Cellphone}, \mathsf{Color}), \mathsf{RxL}(\mathsf{Cellphone.price}, \$799), \mathsf{RmL}(\mathsf{Deal}, \mathsf{carrier})\}$, cost $c(O_2)$=4.2, and answer closeness $\mathsf{cl}(O_2, \{S8, S9\})$ = 1. □

We formulate the problem of answering Why-questions.

Problem statement. Given a query Q, answer $Q(u_o, G)$, graph G, a Why-question W, and an editing budget B, the problem of *answering why-question* is to compute a query rewrite Q^*=$Q \oplus O^*$, such that

$$O^* = \underset{O: c(O) \leq B}{\arg\max}\ \mathsf{cl}(O, V_u)$$

where set V_u refers to V_{C_u} (resp. V_{N_u}) for a Why-not question (resp. Why question) W with guard conditions. That is, it is to compute a query rewrite Q' that ensures an answer that is closest to the desired one specified by W, and incurs bounded editing cost.

These problems are, nevertheless, nontrivial.

Theorem 2: *The problem of answering Why questions and Why-not questions are both* NP*-hard.* □

Algorithm ExactWhy
Input: graph G, query Q, cost bound $B > 0$,
Why question W = (u_o, V_{N_u}), integer m (guard condition);
Output: the optimal query rewrite Q' that answers W.
1. Q' := $\emptyset$; O' := $\emptyset$; $\mathcal{O}$:= GenMBS(Q, G, V_{N_u}, B);
2. **for each** set $O \in \mathcal{O}$ **do**
3. **if** $\mathsf{cl}(O', V_{N_u}) < \mathsf{cl}(O)$ **and** $\mathsf{guard}(O) \leq m$ **then**
4. Q'=$Q \oplus O'$; O'=O;
5. **if** $\mathsf{cl}(O', V_{N_u})$ = 1 **then break** ;
6. **return** Q';

Figure 3: Algorithm ExactWhy

One may show the hardness by constructing a reduction from subgraph isomorphism, for Why-not questions, and k-clique, for Why questions (see detailed proofs in [9]).

IV. Answering Why Questions

We start with algorithms that answer Why question.

A. Computing Optimal Query Rewrites

An optimal query rewrite is induced by an operator set O with $c(O) \leq B$ that maximized $\mathsf{cl}(O)$ and satisfies the guard condition. It is clearly not practical to verify all query rewrites for a Why question. Fortunately, it suffices to verify those induced by a class of *maximal bounded set* (MBS). Given Q, B and guard condition, an operator set O is an MBS, if O satisfies guard condition, $c(O) \leq B$, and there exists no other set O' such that $c(O') \leq B$ and $O \subset O'$.

We have the following sufficient and necessary condition.

Lemma 3: *Given Q and budget B, a query rewrite Q' is optimal if and only if there exists a* MBS *O, such that $Q' = Q \oplus O$, and O has a maximized* $\mathsf{cl}(O)$ *among all* MBS. □

This result can be easily verified by Lemma 1 and the definition of answer closeness for Why questions (see [9]).

Algorithm. Our first algorithm, denoted as ExactWhy and illustrated in Figure 3, makes use of Lemma 3. It takes a partial enumeration scheme to only verify MBS.

(1) It first invokes a procedure GenMBS to generate a set of all MBS $\mathcal{O}$ (line 1).

(2) It then verifies the query rewrites induced by MBS, following the ascending order of their cost. For each operator set $O \in \mathcal{O}$, it verifies the closeness of Q' with a procedure Match, which adapts subgraph isomorphism verification to V_N. It chooses the set O that (a) maximizes $\mathsf{cl}(O)$, and (b) removes no more than m nodes from $Q(u_o, G) \setminus V_{N_u}$ ($\mathsf{guard}(O) \leq m$), and returns Q' by applying O to Q.

We next introduce procedures GenMBS and Match.

Procedure GenMBS**.** Procedure GenMBS "reverse engineers" node pruning in standard subgraph isomorphism. Instead of finding nodes that are non-matches, the goal is to find operators O, that if applied to Q, can exclude V_{N_u}. GenMBS uses a two-step construction: first generates a set of all the "picky" operators $\mathcal{O}_s$, each is promising to prune at least a node in V_{N_u}, and then constructs MBS with $\mathcal{O}_s$. While the query rewrites can be disconnected, the correctness of the algorithm remains intact (see "Correctness").

The procedure GenMBS uses the following notions. (1) Let $N_d(V_{N_u})$ (resp. $N_d(\overline{V_{N_u}})$) denote the d-hop neighbors of the nodes in V_{N_u} (resp. $Q(u_o, G) \setminus V_{N_u}$) in G, *i.e.,* nodes having distance d to some nodes in V_{N_u}. Given a node u' in Q with distance $d(u', u_o)$ to the output node u_o, we define a set $N(V_{N_u}, u')$ (resp. $N(\overline{V_{N_u}}, u')$) as the nodes in $N_{d(u',u_o)}(V_{N_u})$ (resp. $N_{d(u',u_o)}(\overline{V_{N_u}})$) having the same label of u'. (2) Given a node u' and a literal l = $u'.A$ op $c \in F_Q(u')$, attribute $u'.A$ is *common* (resp. *differential*) if it is in a literal of a node $v \in N(\overline{V_{N_u}}, u')$, *and* is in a literal of a (not necessarily the same) node v'' (resp. not seen in any literal of the nodes) from $N(V_{N_u}, u')$. (3) The *active domain* of attribute A *w.r.t.* node set V, denoted as $\mathsf{dom}(A, V)$, is a set of all the distinct values of $v.A$ with v ranges over V.

Generating picky operators. Given V_{N_u}, a single refinement operator o is *picky*, if $Q \oplus \{o\}$ *may* exclude a node in V_{N_u} from $Q(u_o, G)$, by reducing candidates. To ensure the completeness, GenMBS generates AddE first, followed by AddL and RfL. It applies the following rules.

Generating AddE. GenMBS first adds operator AddE(e) to insert a new query edge e = (u_1, u_2), if and only if there is an edge e'= (v_1, v_2), and $v_1 \in N(Q(u_o, G), u_1)$, and $v_2 \in N(Q(u_o, G), u_2)$. Specifically, (1) When both u_1 and u_2 are in Q, it inserts e and sets $L(e)$=$L(e')$; (2) Assume *w.l.o.g.* u_2 is not in Q, it extends AddE (e) to a "composite operator": for each attribute $v_2.A$, it also adds AddL(l) to insert a *template* literal l = $(u_2.A$, '$\perp$', '_'), where op = '$\perp$' is a placeholder ("*don't care*"), and wildcard '_' means "any value", both to be resolved. As either u_1 or u_2 is in Q with diameter d_Q, any query rewrite of Q induced by this operator has diameter at most d_Q+1.

Generating AddL. These rules insert new literals to Q that involve both common and differential attributes to reduce candidates. For each node u' in Q and literal l = $u'.A$ op c, there are two cases. (1) *Pairing constraints*. if $u'.A$ is a common attribute and op $\in \{>, \geq\}$ (resp. op $\in \{<, \leq\}$) specifying lower (resp. upper) bar, but no "pairing" constraints is found at u', it adds a template literal AddL$(u'.A, \leq$, '_') (resp. AddL$(u'.A, \geq$, '_')) to $\mathcal{O}_s$, where '_' is a wildcard to be resolved. (2) *Differential attributes*. if $u'.A$ is differential, it adds a template literal AddL$(u'.A$, '$\perp$', '_'), to be refined.

Generating RfL. These rules refine existing literals in Q to reduce candidates. Given $\mathsf{dom}(A, N(V_{N_u}, u'))$ and each literal l = $u'.A$ op $c \in F_Q(u')$,

- if op $\in \{<, \leq\}$, for each $a \in \mathsf{dom}(A, N(V_{N_u}, u'))$, and $c \geq a$, add RfL$(l$, $u'.A < a)$;
- if op $\in \{>, \geq\}$, for each $a \in \mathsf{dom}(A, N(V_{N_u}, u'))$ and $c \leq a$, add RfL$(l, u'.A > a)$;
- if op is '=', for each $a \in \mathsf{dom}(A, N(Q(u_o, G), u')$ such that $a \neq c$, add RfL$(l, u'.A = a)$.

Intuitively, each of these operators is picky in that there always exist at least a candidate $v' \in N(V_{N_u}, u')$ that fails the literal it enforces, thus may in turn removes matches in V_{N_u}. Note u' can be the targeted output node u itself.

Resolving templates. The last step of GenMBS resolves those AddL(l) with template literals l in $\mathcal{O}_s$, by replacing l with constant literals. (1) For l = ($u'.A$, '$\perp$', '_'), it replaces '$\perp$' to any of $\{<, \leq, =, \geq, >\}$, and reduces AddL(l) to its counterparts with literals carrying '_' only. (2) It then resolves each template with '_' by a case analysis for op, following the rules used in "Generating RfL" to derive "picky" literals. This replaces all AddL(l) with template literals to a set of picky operators applicable to Q.

Example 5: Recall the Why question in Example 1 with V_{N_u} = $\{A5, S5\}$, and let budget B as 4. A fraction of a picky set, which includes o_1 = AddE (Cellphone, Series), o_2 = AddL (Series.val = Series:S)}, and o_3 = AddL (Cellphone.Price>$120, is generated by GenMBS as follows. (1) It first follows rules for adding AddE, and add o_1 ={AddE(Cellphone, Series). As a part of composite operator, it also adds o_4 = AddL(Series.val, '$\perp$', '_')} with a template literal. (2) Next, adds a pairing constraints o_5 = AddL (Cellphone.Price>'_'), with a template literal. (3) It generates RfL operators with common attributes (omitted).

GenMBS next resolves the template literals. (a) It first replaces o_4 with operators with specific op, including o_6 = AddL(Series.val = '_')}. (b) It then resolves the template literals in o_6 and o_5, respectively. For o_6, as dom($Series.val, N(V_{N_u}$, Series)) = $\{Series : S, Series : A\}$, it replaces o_6 with o_2 = AddL(Series.val = $Series : S$)} to exclude $A5$, and o_7 AddL(Series.val = $Series : A$)} to exclude $S5$, both are picky operators. For o_5, it finds that dom($Cellphone.Price, N(V_{N_u}$, Cellphone)) = $\{\$250, \$120\}$, thus replaces o_5 with picky operators o_3 = AddL (Cellphone.Price> $120) to exclude $S5$, and o_8 = AddL (Cellphone.Price> $250) which excludes $S5$ and $A5$. The picky set contains o_1- o_3, o_7, o_8, among others. □

Computing MBS. Once the picky set $\mathcal{O}_s$ is generated, it continues to compute bounded maximal sets $\mathcal{O}$ by partially enumerating $\mathcal{P}(\mathcal{O}_s)$ as the power set of $\mathcal{O}_s$, up to those with cost bound no larger than B. That is, an operator set O is added to $\mathcal{O}$ whenever adding an operator makes its cost larger than B. The set $\mathcal{O}$ is then returned for verification.

The result below verifies that it suffices to only use the picky set computed by GenMBS to answer Why questions.

Lemma 4: ***[completeness of picky set]*** *For any optimal query rewrite Q'=$Q \oplus O$, $O \in \mathcal{P}(\mathcal{O}_s)$ (power set of $\mathcal{O}_s$).* □

Proof sketch: It suffices to show that any single operator $o \notin \mathcal{O}_s$, and their combinations, do not change cl($\cdot$) (not "picky"). This can be verified by a case analysis below. (1) Empty candidates. For example, no refinement can be triggered for literals $l = u'.A$ op c, op $\in <, \leq$, and for common attribute A, $c < a$ for $a \in \min \mathsf{dom}(A, N(V_{N_u}))$. (2) Applying o does not change the match set of any node in Q. (3) o reduces the matches but does not exclude any node in V_{N_u} (see detailed proofs in [9]). □

Procedure Match. Given an operator set $O \in \mathcal{O}$, Match verifies query rewrite Q'=$Q \oplus O$, by *incrementally* checking whether each node $v \in Q(u_o, G)$ remains to be a match of Q', without computing entire $Q'(u_o, G)$ from scratch. It early terminates whenever a subgraph isomorphism is found for v, and updates cl(O) accordingly. If Q' is disconnected, Match simply retains the connected component Q'_{u_o} that contains u_o and performs the evaluation. Indeed, it is easy to verify that $Q'_{u_o}(u_o, G)$ = $Q'(u_o, G)$ for any graph G.

Example 6: Continue with Example 5, given budget B = 4, and the picky set $\mathcal{O}_s$ that contains o_1, o_2, o_7, o_3 and o_8, ExactWhy finds a MBS O' $\{o_1, o_2, o_3\}$ with maximized answer closeness 1. Indeed, (1) adding any operator leads to cost beyond 4; (2) removing any operator (*e.g.,* o_3) leads to smaller closeness (*e.g.,* 0.5 as only $A5$ can be removed). It thus returns $Q_2 = Q \oplus O'$ as in Fig. 1. □

Post processing. Algorithm ExactWhy can be further extended to compute an optimal set O' that also minimizes editing cost (bounded by B). To this end, for each MBS O having maximized cl(O), it performs a post processing to compute a set of minimal MBS that preserves the closeness cl(O), by iteratively removing operators from O whenever possible. Once all such minimal MBS are generated, it chooses the one with the smallest cost (see details in [9]).

Correctness & Complexity. ExactWhy guarantees two invariants below: (1) GenMBS correctly generates all MBSs with picky operators (Lemma 4); and (2) ExactWhy correctly computes the MBSs with largest cl($\cdot$). By Lemma 3, the correctness of ExactWhy follows.

We next analyze the time cost of ExactWhy. Define the size $|N_{d_Q+1}(Q(u_o, G))|$ of $N_{d_Q+1}(Q(u_o, G))$ as the number of total literals and edges, similarly as the size $|Q|$. (1) GenMBS generates a constant number of operators for each literal and edge in $N_{d_Q+1}(Q(u_o, G))$. It thus takes $O(|Q||N_{d_Q+1}(Q(u_o, G))|)$ time to generate the picky set $\mathcal{O}_s$, with size $|\mathcal{O}_s|$ in $O(|N_{d_Q+1}(Q(u_o, G))|)$. (2) As any operator has a cost at least $\frac{d_Q}{d_Q+2}$, any MBS has size no larger than $\frac{B(d_Q+2)}{d_Q}$, further bounded by $3B$ for nontrivial Q beyond a single output node ($d_Q \geq 1$). Thus at most $|\mathcal{O}_s|^{3B}$ MBS are verified, each takes at most $|N_{d_Q+1}(V_{N_u})|^{|Q|}$ time. The total time cost is thus in $O(|Q||N_{d_Q+1}(Q(u_o, G))| + |\mathcal{O}_s|^{3B}|N_{d_Q+1}(V_{N_u})|^{|Q|})$.

Under a practical assumption that B, d_Q and $|V_{N_u}|$ are small constants, ExactWhy is feasible in large G. Indeed, 96% of real-life SPARQL queries (a common subgraph query language for entity search) contain up to 7 edges, and 63% can be exactly verified in polynomial-time [28].

B. Approximating Optimal Query Rewrites

Algorithm ExactWhy needs to find and verify every cost-bounded sets and may be expensive. We can further reduce the enumeration and verification cost by developing feasible approximation algorithms. Our major result is as follows.

Theorem 5: *There is an algorithm for answering Why questions which satisfies the following (O^* is the optimal set):*
- *it computes an operator set O within cost B, such that* $\mathsf{cl}(O, V_{N_u}) \geq \frac{1}{2} \cdot (1 - \frac{1}{e}) \cdot \mathsf{cl}(O^*, V_{N_u})$ - $6B\epsilon$, *and*
- *it incurs a cost determined by Q, V_{N_u} and B only.*

where ϵ is the absolute error it makes in estimating $\mathsf{cl}(\cdot)$. □

That is, the algorithm guarantees a *relative* approximation ratio, in terms of an error ϵ between an estimated, polynomial-time computable closeness and its exact counterpart, for any query rewrites. If ϵ = 0,it is a constant factor approximation. Moreover, as Q, $|V_{N_u}|$ and B are often small, the algorithm is feasible for big G. To prove Theorem 5, we next show that answering Why questions can be reduced to a budgeted submodular maximization problem [29].

Submodularity. Given refinement operator set O, we define the *marginal gain* of a refinement operator o to O as $\mathsf{mg}(O, o)$ = $\mathsf{cl}(O \oplus \{o\})$ - $\mathsf{cl}(O)$. We first show the following result, which can be easily verified by Lemma 1 (see [9]).

Lemma 6: *Function* $\mathsf{cl}(\cdot)$ *is submodular over picky set $\mathcal{O}_s$,* i.e., *for any sets O_1 and O_2, such that $O_1 \subset O_2 \subset \mathcal{O}_s$, and an operator $o \in \mathcal{O}_s$ ($o \notin O_2$),* $\mathsf{mg}(O_1, o) \geq \mathsf{mg}(O_2, o)$. □

The above property holds for disconnected query rewrites. Given Lemma 6, we can verify that finding optimal query rewrites is to solve a *budgeted submodular maximization* [29] problem. Given picky set $\mathcal{O}_s$ and set V_{N_u}, it computes a set $O \subseteq \mathcal{O}$, such that $c(O) \leq B$, and $\mathsf{cl}(O, V_{N_u})$ is maximized. It can be verified that a greedy algorithm ensures a $1-\frac{1}{e}$ approximation by greedily selecting operators with maximum mg [30], yet requires $O(|\mathcal{O}_s|^2)$ subgraph isomorphism test. We can do better: our algorithm only verifies each picky operator once, and efficiently estimates $\mathsf{cl}(\cdot)$ for all the rest sets, with guarantees in Theorem 5.

Algorithm. The algorithm, denoted as ApproxWhy and illustrated in Figure 4, has the following steps.

(1) It invokes a procedure GenPicky, which simulates the first phase of GenMBS to only generate picky set $\mathcal{O}_s$ (line 1). It then initializes sets $\mathcal{O}_1$ and $\mathcal{O}_2$ (lines 2-3), where $\mathcal{O}_1$ is set as the single operator with best $\mathsf{cl}(\cdot)$, verified by Match.

(2) It then iteratively selects a refinement operator o^* from $\mathcal{O}$ that maximizes the ratio of cost $c(o^*)$ to an estimated marginal gain $\hat{\mathsf{mg}}(o^*)$ (lines 5-9). The latter refers to an estimated number of the matches in V_{N_u} that are removed due to o^*, estimated by a procedure EstMatch The process repeats until all the picky operators are processed.

Algorithm ApproxWhy
Input: graph G, query Q, cost bound $B > 0$,
  a Why question $W = (u_o, V_{N_u})$, integer m (guard condition);
Output: a query rewrite Q' that answers W.
1. $\mathcal{O}_s := \mathsf{GenPicky}(Q, G, V_c, B)$;
2. set $\mathcal{O}_1 := \arg\max\{\mathsf{cl}(\{o\})\} : o \in \mathcal{O}_s, c(o) \leq B$, $\mathsf{guard}(\{o\}) \leq m$ };
3. set $\mathcal{O}_2 := \varnothing$;
4. $\mathcal{O}'_s = \mathcal{O}_s$; V'_{N_u}:=V_{N_u};
  /*greedy selection of refinement operators*/
5. **while** $\mathcal{O}'_s \neq \varnothing$ **do**
6. **for** $o \in \mathcal{O}_s$ **do** $\hat{\mathsf{mg}}(o) := \hat{\mathsf{cl}}(\mathcal{O}_2 \cup \{o\}) - \hat{\mathsf{cl}}(\mathcal{O}_2)$;
7. $o^* := \arg\max\{\frac{\hat{\mathsf{mg}}(o)}{c(o)} : o \in \mathcal{O}_s\}$;
8. **if** $c(\mathcal{O}_2)$+$c(o^*)$$\leq$$B$ **and** $\mathsf{guard}(\mathcal{O}_2 \cup \{o^*\}) \leq m$ **then**
9. $\mathcal{O}_2 := \mathcal{O}_2 \cup \{o^*\}$; $\mathcal{O}'_s := \mathcal{O}'_s \setminus \{o^*\}$;
10. $\mathcal{O}' = \arg\max_{\mathcal{O}_i \in \{\mathcal{O}_1, \mathcal{O}_2\}} \hat{\mathsf{cl}}(O_i)$;
11. construct query rewrite Q' = $Q \oplus \mathcal{O}'$;
12. **return** Q';

Figure 4: Algorithm ApproxWhy

(3) It then set $\mathcal{O}$' as the set of $\mathcal{O}_1$ or $\mathcal{O}_2$, whichever has larger estimated answer closeness. The query rewrite Q' is constructed with $\mathcal{O}'$ and returned (lines 11-12).

Procedure EstMatch. Given operators O and operator o, EstMatch estimates $\mathsf{cl}(O)$ and $\mathsf{cl}(O \cup \{o\})$ as $\hat{\mathsf{cl}}(O)$ and $\hat{\mathsf{cl}}(O \cup \{o\})$, and computes $\hat{\mathsf{mg}}(o)$ as $\hat{\mathsf{cl}}(O \cup \{o\})$ - $\hat{\mathsf{cl}}(O)$.
(1) For each picky operator $o \in O$, it first finds, *once for all*, a set of affected nodes Aff (o) that are no longer matches of Q due to o. This is doable as soon as $\mathsf{cl}(o)$ is computed.
(2) It then estimates $\mathsf{Aff}(V_{N_u})$, a fraction of V_{N_u} that becomes non-matches due to O. (a) Given O, it first sets $\mathsf{Aff}(V_{N_u})$ = $\bigcup_{o \in O}(\mathsf{Aff}(o) \cap V_{N_u})$, *i.e.,* all the non-matches of u already identified by $\mathsf{Aff}(o)$. (b) It then extends $\overline{\mathsf{Aff}}(V_{N_u})$ and updates $\hat{\mathsf{cl}}(O)$, by checking if the nodes in $\overline{\mathsf{Aff}}(V_{N_u})$ = $V_{N_u} \setminus \mathsf{Aff}(V_{N_u})$ becomes a non-match. This is done by consulting a path index that samples a bounded number of paths from Q and verifies path matches (see [9]). The step continues until all the nodes in $\overline{\mathsf{Aff}}(V_{N_u})$ are processed.

Example 7: Consider the picky set $\mathcal{O}_s$ in Example 5. Let $\mathcal{O}_2$ contains o_3 = AddL(Cellphone.Price $>$ \$120). ApproxWhy next greedily chooses o_2 = AddL (Series.val=Serie:S), which has maximized $\frac{\hat{\mathsf{mg}}(o_2)}{c(o_2)}$ = 0.5. Other picky operators (*e.g.,* o_8 = AddL(Cellphone.Price $>$ \$250)) are not as good (*e.g.,* $\hat{\mathsf{mg}}(o_8)$=0) given o_3, thus are not selected. □

Analysis. To see the approximation ratio, we construct a reduction from answering Why question to the budgeted submodular maximization with *estimated* marginal gain [29], following the construction remarked earlier. Given estimated marginal gain with absolute error ϵ', $\mathsf{cl}(\cdot)$ can be maximized with guarantee $\frac{1}{2} \cdot (1 - \frac{1}{e}) \cdot \mathsf{cl}(O^*, V_{N_u})$-$(\frac{B(d_Q+2)}{d_Q})\epsilon'$ by a greedy selection, following [29]. ApproxWhy simulates the greedy selection with an error $2{\cdot}\epsilon$ for estimating mg, thus ensures closeness at least $\frac{1}{2}{\cdot}(1{-}\frac{1}{e}){\cdot}\mathsf{cl}(O^*, V_{N_u}){-}2(\frac{B(d_Q+2)}{d_Q})\epsilon$, which is bounded by $\frac{1}{2}{\cdot}$(1-$\frac{1}{e}$)${\cdot}\mathsf{cl}(O^*, V_{N_u})$- $6B\epsilon$ (as $d_Q \geq 1$).

It takes $O(|Q||N_{d_Q+1}(Q(u_o,G))|+|\mathcal{O}_s||N_{d_Q+1}(V_{N_u})|^{|Q|})$ time for initialization, following the analysis of ExactWhy. It takes $O(|N_{(d_Q+1)}(V_{N_u})|)$ time to estimate $\hat{\text{mg}}(o)$ for each o. ApproxWhy is thus in total $O(|Q||N_{d_Q+1}(Q(u_o,G))| + |\mathcal{O}_s||N_{d_Q+1}(V_{N_u})|^{|Q|} + |\mathcal{O}_s|^2|N_{(d_Q+1)}(V_{N_u})|)$ time.

The above analysis completes the proof of Theorem 5.

V. Answering Why-Not Questions

We next study answering Why-not questions.

A. Computing Optimal Query Rewrites for Why-not

The first good news is that the sufficient and necessary condition (Lemma 3) for Why questions has a counterpart for Why-not questions. The only difference is that the maximal bounded sets consist of relaxation operators only.

Lemma 7: *Given query Q and cost bound B, a query rewrite Q'=$Q \oplus O$ is optimal for a Why-not question, if and only if O is a maximal bounded set with relaxation operators, and has a maximum* cl(O) *among all maximal bounded sets.* □

Our first algorithm, denoted as ExactWhyNot, takes a partial enumeration and verification strategy, similarly as its counterpart ExactWhy. The difference is that (1) it uses a revised procedure GenMBS to generate a picky set of relaxation operators and maximal bounded sets; and (2) it invokes a revised Match to verify if the nodes in V_{C_u} become new matches and the query rewrite satisfies the guard condition, and enforce additional constraints in C.

We present the details of the revised procedure GenMBS.

Procedure GenMBS. Similar to its counterpart in ExactWhy, procedure GenMBS follows two-step process that first generates a picky set, and then verifies maximal bounded sets, but consists of picky relaxation operators that can include new matches in V_{C_u}. It uses the following similar notions. (1) Given a query node u' in Q, we define a set $N(V_{C_u}, u')$ as the nodes in $N_{d(u',u)}(V_{C_u})$ having the same label of u'. (2) The *common attributes* of a node u' in Q *w.r.t.* V_{C_u} refers to all the node attributes in a literal of Q that is also seen in a literal of a node from $N(V_{C_u}, u')$.

Generating picky set. Procedure GenMBS inspects the candidates in $N(V_{C_u}, u')$ for nodes u' in Q to identify picky set that enlarges candidates of u'. It uses the following rules.

Generating RxL. For each literal l= $(u'.A$ op $c) \in F_Q(u')$ of Q with common attribute $u'.A$, and each constant $a \in$ dom$(A, N(V_{C_u}, u'))$,

- if op $\in \{<, \leq, =\}$ and $c \leq a$, add RxL(l, $u'.A \leq a$);
- if op $\in \{>, \geq, =\}$ and $c \geq a$, add RxL(l, $u'.A \geq a$);

Generating RmL *and* RmE. GenMBS simply adds RmE(e) (resp. RmL(l)) to $\mathcal{O}$ for each edge e and literal l in Q.

Once the set $\mathcal{O}_s$ with a single operator is generated, it continues to compute the maximal operator sets $\mathcal{O}$ with cost bounded by B, until no new operator sets can be inserted. The completeness of the picky set remains intact (see [9]).

Example 8: Given query Q and the Why-not question that specifies V_{C_u} as $\{S8, S9\}$ (Figure 1), GenMBS generates $\mathcal{O}_s$ that include o_1 = RxL(l, Cellphone.price $\$654$), o_2 = RxL(l, Cellphone.price, $\$799$), o_3 = RmE(Cellphone, Color), and o_4 = RmL(Deal, carrier=$AT\&T$), among others, where l = (Cellphone.price $\leq \$650$) is a literal from Q. For example, as the active domain of $price$ in V_{C_u} contains $\{\$654, \$799\}$, it relaxes l with o_1 and o_2, which tries to include $S8$, and both,respectively. Given B=4.2, a MBS $\{o_2, o_3, o_4\}$ with the best closeness induces optimal query rewrite Q_2. □

Correctness & Complexity. Following a similar analysis of ExactWhy, the correctness of algorithm ExactWhyNot follows from that GenMBS correctly computes complete bounded maximal operator sets, and by Lemma 7 (see [9]).

For the complexity, it takes in total $|Q||N_{d_Q}(V_{C_u})|$ time to generate $\mathcal{O}_s$, with size bounded by $O(|Q||N_{d_Q}(V_{C_u})|)$. The largest maximal set has size bounded by $\frac{B(d_Q+1)}{d_Q}$, thus there are at most $|\mathcal{O}_s|^{\frac{B(d_Q+1)}{d_Q}}$ bounded maximal sets. The total time is in $O((|Q||\mathcal{O}_s|)^{2B}|N_{d_Q}(V_{C_u})|^{|Q|})$ (when $d_Q \geq 1$).

B. A Faster Heuristic

One may also consider approximation algorithms for Why-not questions. Nevertheless, unlike Why questions, the answer closeness under relaxation is no longer submodular, and is hard to approximate even with a value oracle that reports cl$(\cdot)$ [30]. We thus resort to fast heuristic algorithms.

Algorithm. The general idea is to compute query rewrites by solving a *budgeted maximum cover* problem. Given picky set $\mathcal{O}_s$ and V_{C_u}, it computes an operator set $\mathcal{O} \subseteq \mathcal{O}_s$ with estimated matches that maximally "cover" V_{C_u}.

The algorithm, denoted as FastWhyNot (see [9]), performs the following. (1) It invokes a procedure GenPicky to generate picky operators $\mathcal{O}_s$ as in GenMBS, and initializes a working set $\mathcal{O}'$. (2) It then iteratively selects an operator o^* from $\mathcal{O}_s$ that (1) satisfies the guard condition, and (2) maximizes the ratio of $c(o^*)$ to estimated marginal gain $\hat{\text{mg}}(o^*)$ (estimated by a revised EstMatch) similarly as in ExactWhy, and constructs Q' with greedily selected $\mathcal{O}'$.

Revised EstMatch. Given relaxation operator O, EstMatch estimates cl(O) by estimating new matches from V_{C_u} introduced by O. It traces back to GenPicky and finds nodes Aff(o) that are potential new matches for each o. It then performs similar sampled path tests for each node $v \in V_{C_u}$, by consulting the path index at run-time. The difference is that it considers v to be an estimated match, only when it passes all tests, and has matched paths that contain only matches in $Q(u_o, G)$ and Aff(o). It treats all such node as the new matches, and update $\hat{\text{cl}}(\cdot)$ accordingly (see [9]).

Time cost. FastWhyNot takes in total $O(|Q||N_{d_Q}(V_{C_u})| + (|\mathcal{O}_s|)^2|N_{d_Q}(V_{C_u})|)$ time to compute picky set $\mathcal{O}_s$ and performs greedy selection. We found that it is much (15 times) faster than ExactWhyNot, without losing much quality of optimal query rewrites (at least 80% as good).

Extensions. Our algorithms can be readily extended to support (1) answering multiple Why-questions that involve multiple output nodes in Q, with provable performance guarantees; (2) *Why-empty* (resp. *Why-so-many*) [21] questions, which are special cases of Why-not (resp. Why) questions without specifying V_{C_u} (resp. V_{N_u}), but compute Q' that returns at least a match (reduces certain number of matches); and (3) Subgraph queries defined by approximate pattern matching [4], [5]. We present the details in [9].

VI. EXPERIMENT

Using real-world graphs, we next experimentally verify the effectiveness and efficiency of our algorithms for answering why-questions (see [9] for more results).

Experiment Setting. We used the following setting.

Datasets. We use five real-life graphs: (1) *DBpedia*[2] consists of 4.86M entities, 15M edges, 676 labels (*e.g.,*Person, Building), and on average 9 attributes per node, (2) *Yago*[3], with 1.54M nodes and 2.37M edges (sparser compared to *DBpedia*), but with more diversified (324343) labels and on average 5 attributes per node, (3) *Freebase* (version 14-04-14)[4], with 40.32M entities and on average 8 attributes per node, 63.2M relationships, and 9630 labels; (4) *Pokec*[5], a social network with 1.6M users, 30.6M edges and 60 attributes per node; and (5) *IMDb*[6], with 1.7M nodes (*e.g.,*movies), 5.2M edges,and on average 6 attributes per node.

We also use *BSBM*[7] e-commerce benchmark to generate synthetic knowledge graphs over products with different number of nodes (up to $50M$) and edges (up to $126M$), and labels drawn from an alphabet of 3080 labels.

Query & Question generation. We developed a query generator, which generates queries controlled by query size $|Q|$ and topologies (trees, acyclic, cyclic), as follows: (1) it randomly picks a set of keywords as types and generate summaries [5] as query templates; (2) for each template, it randomly selects a query node u as the output node; (3) from the isomorphic subgraphs of the query template, we assign a set of attributes for each query node to ensure non-empty $Q(u_o, G)$. To generate Why questions, we randomly select a set of nodes in $Q(u_o, G)$ as V_{N_u}. For Why-not questions, we select V_{C_u} with the same type as u_o and randomly associate each question with a constraint C that has up to 2 literals.

By default, we set Q to contain 4 edges, 2 literals per node, both $|V_{C_u}|$ and $|V_{N_u}|$ to 3, editing budget $B = 4$, and $m = 2$ for guard conditions, unless otherwise specified.

Algorithms. We implemented the following in Java. (1) For Why questions, we compare the approximation algorithm ApproxWhy with its exact counterpart ExactWhy, and its variant IsoWhy, which uses subgraph isomorphism Match instead of EstMatch, thus ensures ϵ=0. (2) For Why-not questions, we compare FastWhyNot with the exact algorithm ExactWhyNot, and its variant IsoWhyNot, which applies exact Match to compute the marginal gain.

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, each batch with 50 Why-questions, and report the averaged results.

Experimental results. We next present our findings.

Exp-1: Effectiveness (Why Questions). We first evaluate the effectiveness of our algorithms, including ExactWhy, ApproxWhy and IsoWhy, for answering Why questions. We measure their effectiveness by the absolute answer closeness over all real-world datasets. We also ensure known optimal cases with closeness ≈ 1, as a reference for ExactWhy.

Figure 5(a) tells us the following. (1) The exact algorithm ExactWhy always reports the optimal query rewrite. (2) With estimated closeness (EstMatch), ApproxWhy computes good query rewrites that have closeness at least 85% to their optimal counterpart, in all cases. (3) Replacing EstMatch with Match improves the quality of query rewrites (up to 0.08). These verify the quality guarantees of ExactWhy and ApproxWhy. Moreover, EstMatch is quite accurate ($\epsilon \leq 0.02$ on average).

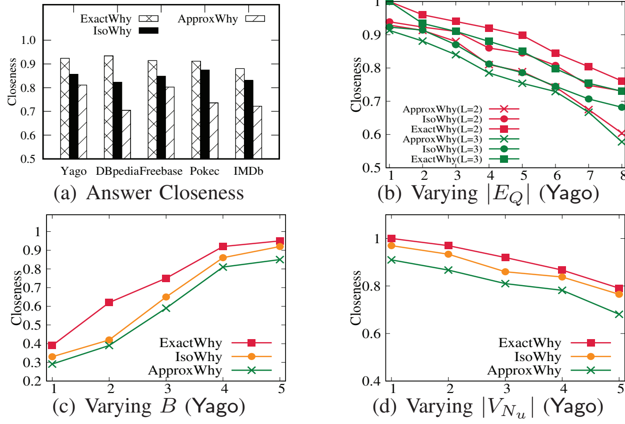
Varying query size. Varying number of edges $|E_Q|$ from 1 to 8, and number of literals L per node from 2 to 3, we report the effectiveness of our algorithms over Yago in Figure 5(b). (1) Under a fixed budget ($B = 4$), the closeness decreases given more edges and literals in Q, for all the algorithms. We found that it often requires more budget for larger $|Q|$, due to increased query complexity from topology and literals. (2) ApproxWhy preserves 85% of the optimal closeness in all cases. It approximates ExactWhy better for small queries, as EstMatch is more accurate over small queries.

Varying cost budget B. As shown in Figure 5(c) (over Yago), the answer closeness of all three algorithms increase when given larger cost budget B. The closeness converges at small B ($B = 4$), which indicates that it often requires a small modification to Q to answer why questions in practice.

Varying $|V_{N_u}|$. We evaluate the impact of $|V_{N_u}|$ under a fixed cost budget ($B = 4$). We simulate interactive sessions for asking Why questions, which enlarges V_{N_u} by adding entities to it only, and varies $|V_{N_u}|$ from 1 to 5. As shown in Figure 5(d), under a fixed budget, it is "harder" to exclude more nodes as V_{N_u} grows, as more operators are needed; On the other hand, all our algorithms still provide reasonable query rewrite with closeness at least 0.68.

Exp-2: Efficiency (Why Questions). Using the same setting in Exp-1, we report the efficiency of our algorithms for answering why questions. Figure 6(a) verifies that it is feasible to answer why questions for large graphs. On

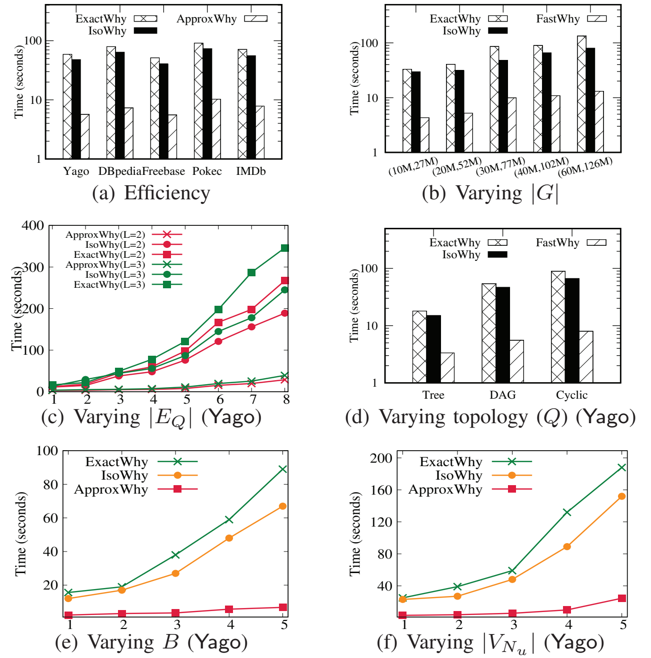
[2]http://dbpedia.org
[3]http://www.mpi-inf.mpg.de/yago
[4]http://freebase-easy.cs.uni-freiburg.de/dump/
[5]https://snap.stanford.edu/data/soc-pokec.html
[6]https://www.imdb.com/interfaces/
[7]http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/



(a) Answer Closeness (b) Varying $|E_Q|$ (Yago)

(c) Varying B (Yago) (d) Varying $|V_{N_u}|$ (Yago)

Figure 5: Answering Why questions: Effectiveness

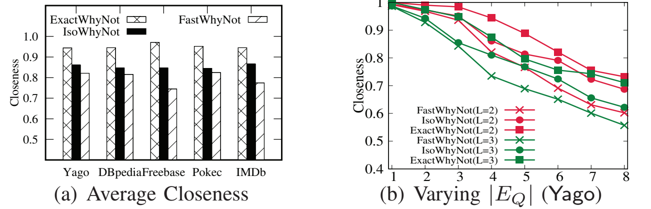


(a) Efficiency (b) Varying $|G|$

(c) Varying $|E_Q|$ (Yago) (d) Varying topology (Q) (Yago)

(e) Varying B (Yago) (f) Varying $|V_{N_u}|$ (Yago)

Figure 6: Answering why questions: Efficiency



(a) Average Closeness (b) Varying $|E_Q|$ (Yago)

Figure 7: Answering Why-not questions: Effectiveness

average, ApproxWhy outperforms ExactWhy and IsoWhy, by 9.7 times and 7.7 times, respectively. It takes on average 7 seconds to achieve near optimal results ExactWhy takes on average 71 seconds to ensure optimal query rewrites.

Scalability. Figure 6(b) reports the performance on synthetic graphs using the same set of Why-questions. The result shows that ApproxWhy scales well with $|G|$, and improves ExactWhy and IsoWhy better when $|G|$ becomes larger. Figure 6(c) evaluates the impact of query size, by varying $|E_Q|$ and literals per node (L) of Q. (1) While all algorithms take longer for larger queries as more operators and nodes need to be inspected, they are quite feasible: it takes up to 3.3 minutes for queries with 6 edges and 12 literals. (2) ApproxWhy is less sensitive to $|E_Q|$ and L. In all cases, it takes up to 39 seconds, and is on average 8.7 and 6.8 times faster than ExactWhy and IsoWhy.

Varying query topologies. Figure 6(d) tells us that answering Why questions posed on tree queries are in general more efficient than those on acyclic or cyclic queries, as simpler topologies reduce the cost of recursive verification.

Varying cost budget B. Varying budget B from 1 to 5, Figure 6(e) shows that all the algorithms take longer time to consume larger budget. ApproxWhy is the least sensitive one, due to its reduced verification cost with estimated closeness. Note that the costs of ExactWhy and IsoWhy do not necessarily grows exponentially with B, due to that for larger B, they both have higher chance to early terminate.

Varying $|V_{N_u}|$. As shown in Figure 6(f), under the same B, larger $|V_{N_u}|$ induces larger picky sets and more matches to be verified (with Match or EstMatch) and removed from the result, consistent with our complexity analysis.

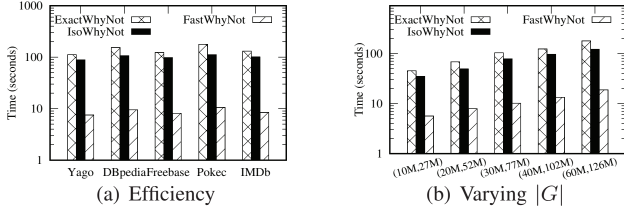
Exp-3: Effectiveness (Why-Not Questions). Figure 7(a) reports the closeness of query rewrites generated from ExactWhyNot, FastWhyNot and IsoWhyNot. We find the following. (1) Under small budget B = 4, ExactWhyNot can already cover almost all V_{C_u} (average closeness > 0.95). (2) While FastWhyNot reports answer closeness not as good, it achieves a quite good result, with answer closeness at least 84% of the optimal counterparts, for all the cases.

Varying size $|Q|$. We evaluate the impact of $|Q|$ to the effectiveness in Figure 7(b), varying $|E_Q|$ from 1 to 8 and average literal per node L from 2 to 3. The closeness of all computed query rewrites decreases as Q becomes larger, which is consistent with its counterpart for Why questions (Fig. 5(b)). Indeed, it is more likely to cover matches for small queries given a fixed budget, and EstMatch is also more accurate. In general, it incurs an error $\epsilon \leq 0.04$.

Varying B and $|V_{C_u}|$. The result is consistent with the results for Why questions. We present more details in [9].

Exp-4: Efficiency (Why-not Questions). Figure 8(a) shows that it takes on average 2 minutes, 100 seconds and 8.8 seconds for ExactWhyNot, IsoWhyNot and FastWhyNot to answer a Why-not question with 3 missing matches under cost budget $B = 4$. FastWhyNot is 15.7 times and 11 times faster than ExactWhyNot and IsoWhyNot, respectively, and computes query rewrites with good answer closeness.

Figure 8(b) verifies that FastWhyNot scales well with $|G|$ and $|E_Q|$. In all cases, it takes less than 30 seconds, and improves ExactWhyNot and IsoWhyNot better for larger $|G|$. We report more efficiency results in [9].



(a) Efficiency (b) Varying $|G|$

Figure 8: Answering Why-not questions: Efficiency

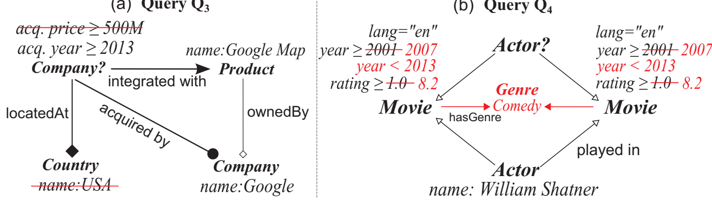


Figure 9: Real-world Why-questions

Exp-5: Case Analysis. We also verify the application of our algorithms in knowledge exploration, with two cases below. *Knowledge exploration*. A query Q_3 (Fig. 9 (a), in black), is posed on DBpedia, which aims to find U.S companies acquired by Google since 2013, at a price more than $\$500M$ and integrated with Google Map. While a single company $\{$Skybox Imaging$\}$ is returned, a user wonders why Urban Engines and Waze, are not among the results.

(1) Given "*Why-not Urban Engines*?", FastWhyNot rewrites Q_3 to Q'_3, which successfully returns Urban Engines, by removing the price constraint (crossed by red line). Interestingly, it turns out that no price was reported for this entity in DBpedia, indicating a data quality issue (missing facts).

(2) Given "*Why-not Waze?*", FastWhyNot further removes country name USA and returns Waze. A closer look at the facts reveals that it was originally founded in Israel, providing a new fact to the user for her future investigation.

"Why-so-many?". A second query Q_4 (Fig. 9 (b)) on IMDB searches for actors who co-played with William Shatner in at least two movies with reasonable ratings no earlier than 2001. This query returns, surprisingly, more than $6,200$ actors. The user asks a follow-up "Why-so-many" question, with a hope to retain at most 100 actors. In response, ApproxWhy revises Q_4 by narrowing down the movies release dates, increasing the rating, and introducing genre Comedy. These operators refine the result to 27 actors. It turns out that many actors considered as co-players only co-attended the talk-shows, which are (not very accurately) labeled as "movies", if no genre is specified by ApproxWhy.

VII. Conclusions

We have formalized the problem of answering Why-questions for subgraph queries. We have shown that these problems are in general intractable. We have developed feasible algorithms, from exact and approximation to fast heuristics, with properties such as *relative approximation ratio*, and early termination. As verified by our experimental study, our methods and their extensions are efficient and report useful explanations. One future topic is to enable trade off between query cost and closeness for exploratory graph search. Another topic is to explore the applications of our methods for graph sensemaking and visualized querying.

Acknowledgments. This work is supported in part by NSF IIS-1633629, NSFC 61876144, USDA/NIFA 2018-67007-28797, Siemens and Huawei HIRP.

References

[1] G. Kasneci, F. M. Suchanek, G. Ifrim, M. Ramanath, and G. Weikum, "Naga: Searching and ranking knowledge," in *ICDE*, 2008.
[2] M. H. Namaki, Y. Wu, Q. Song, P. Lin, and T. Ge, "Discovering graph temporal association rules," in *CIKM*, 2017.
[3] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, "A (sub) graph isomorphism algorithm for matching large graphs," *TPAMI*, 2004.
[4] W. Fan, J. Li, S. Ma, N. Tang, Y. Wu, and Y. Wu, "Graph pattern matching: From intractable to polynomial time," *PVLDB*, 2010.
[5] Q. Song, Y. Wu, and X. L. Dong, "Mining summaries for knowledge graph search," in *ICDM*. IEEE, 2016, pp. 1215–1220.
[6] R. Pienta, J. Abello, M. Kahng, and D. H. Chau, "Scalable graph exploration and visualization: Sensemaking challenges and opportunities," in *BigComp*, 2015.
[7] Y. Song, H. E. Chua, S. S. Bhowmick, B. Choi, and S. Zhou, "Boomer: Blending visual formulation and processing of p-homomorphic queries on large networks," in *SIGMOD*, 2018.
[8] X. Z. Mohammad Hossein Namaki, Yinghui Wu, "Gexp: Cost-aware graph exploration with keywords," *SIGMOD*, 2018.
[9] "Full version," http://eecs.wsu.edu/∼qsong/Files/paper/ICDE19f.pdf.
[10] P. Buneman, S. Khanna, and T. Wang-Chiew, "Why and where: A characterization of data provenance," in *ICDT*, 2001, pp. 316–330.
[11] A. Chapman and H. Jagadish, "Why not?" in *SIGMOD*, 2009.
[12] Z. He and E. Lo, "Answering why-not questions on top-k queries," *TKDE*, vol. 26, no. 6, 2014.
[13] M. S. Islam, R. Zhou, and C. Liu, "On answering why-not questions in reverse skyline queries," in *ICDE*, 2013.
[14] Q. Liu, Y. Gao, G. Chen, B. Zheng, and L. Zhou, "Answering why-not and why questions on reverse top-k queries," *VLDBJ*, 2016.
[15] Q. T. Tran and C.-Y. Chan, "How to conquer why-not questions," in *SIGMOD*, 2010.
[16] M. Wang, J. Liu, B. Wei, S. Yao, H. Zeng, and L. Shi, "Answering why-not questions on sparql queries," *KAIS*, 2018.
[17] Z. Bao, Y. Zeng, H. Jagadish, and T. W. Ling, "Exploratory keyword search with interactive input," in *SIGMOD*, 2015, pp. 871–876.
[18] M. S. Islam, C. Liu, and J. Li, "Efficient answering of why-not questions in similar graph matching," *TKDE*, vol. 27, no. 10, 2015.
[19] D. Mottin, F. Bonchi, and F. Gullo, "Graph query reformulation with diversity," in *KDD*, 2015.
[20] E. Vasilyeva, M. Thiele, A. Mocan, and W. Lehner, "Relaxation of subgraph queries delivering empty results," in *SSDBM*, 2015.
[21] E. Vasilyeva, M. Thiele, C. Bornhövd, and W. Lehner, "Answering "why empty?" and "why so many?" queries in graph databases," *JCSS*, vol. 82, no. 1, 2016.
[22] P. Yi, B. Choi, S. S. Bhowmick, and J. Xu, "Autog: a visual query autocompletion framework for graph databases," *VLDBJ*, 2017.
[23] Y. Y. Weiss and S. Cohen, "Reverse engineering spj-queries from examples," in *PODS*, 2017, pp. 151–166.
[24] N. Jayaram, A. Khan, C. Li, X. Yan, and R. Elmasri, "Querying knowledge graphs by example entity tuples," *TKDE*, 2015.
[25] W. Fan, X. Wang, and Y. Wu, "Diversified top-k graph pattern matching," *VLDB*, 2013.
[26] L. Zou, J. Mo, L. Chen, M. T. Özsu, and D. Zhao, "gstore: answering sparql queries via subgraph matching," *VLDB*, 2011.
[27] G. Zhu and C. A. Iglesias, "Computing semantic similarity of concepts in knowledge graphs," *TKDE*, vol. 29, no. 1, 2017.
[28] X. Han, Z. Feng, X. Zhang, X. Wang, G. Rao, and S. Jiang, "On the statistical analysis of practical sparql queries," in *WebDB*, 2016, p. 2.
[29] A. Krause and C. Guestrin, "A note on the budgeted maximization of submodular functions," 2005.
[30] Z. Svitkina and L. Fleischer, "Submodular approximation: Sampling-based algorithms and lower bounds," *SICOMP*, vol. 40, no. 6, 2011.