This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2020.3046436, IEEE Transactions on Knowledge and Data Engineering

Answering Why-Questions for Subgraph Queries

Qi Song, Mohammad Hossein Namaki, Peng Lin, and Yinghui Wu

Abstract—Subgraph queries are routinely used to search for entities in richly attributed graphs *e.g.,* social networks and knowledge graphs. With little knowledge of underlying data, users often need to rewrite queries multiple times to reach desirable answers. Why-questions are studied to clarify missing or unexpected query results. This paper makes a first step to answer Why-questions for entity search in attributed graphs. We consider three common types of Why-questions: Why-not, Why, and Why-rank, which suggest query manipulations that are responsible for user-specified missing, unexpected, and undesirably ranked entities, respectively. (1) We approach a general query rewriting paradigm that suggests 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 rewrites is intractable, we develop feasible algorithms, from approximation to fast heuristics, and provide query rewrites with (near) optimality guarantees whenever possible, for Why, Why-not and Why-rank questions. We further show that our results remain intact for Why questions that (1) request a single query rewrite to clarify multiple types of entities, and (2) variants such as Why-empty and Why-so-many, by providing the matching algorithms. 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.

Index Terms—Data provenance, Query Processing, Data Exploration, Graph Data

1 Introduction

Subgraph queries have been applied to search for entities in richly-attributed graphs *e.g.,* knowledge bases [1] and social networks [2]. A *subgraph query* Q is a graph pattern with a designated "focus" entity u_o to specify entities of interests. Given a graph G, it returns (top k) entities $Q(G)$ in G (*w.r.t.* a ranking function) that match u_o with desirable attribute values and satisfy topological constraints.

Query processing has been extensively studied for subgraph queries [3]. Writing such queries is, nevertheless, a nontrivial task for end users. With little prior knowledge of data, users often have to revise the queries multiple times to find desirable answers. An explain functionality supported by *query rewriting* is thus desirable to help them tune their queries towards desirable answers. Specifically, one often want to ask the following *Why-questions*:

- **Why**: "*why there are some unexpected entities?*";
- **Why-not**: "*why certain entities are missing?*"; and
- **Why-rank**: "*why some entities are not ranked as desired?*"

Example 1: Fig 1 illustrates a fraction of a knowledge graph G about products of *e.g.,* online stores [4]. Each entity carries a type *e.g.,* Cellphone and attributes (*e.g.,* Price) with corresponding values (*e.g.,* "$250"). A user poses a query Q with a marked node "Cellphone?" to search for Samsung cellphones packed with color pink and carrier $AT\&T$, with price no more than $650. The answer $Q(G)$ contains three Samsung models $S5$, $A5$, and $S6$. To further explore desired cellphones, the user asks the questions below.

- *Q. Song, M. H. Namaki and P. Lin are with the School of Electrical Engineering and Computer Science, Washington State University, Pullman, WA 99163. E-mail: qi.song@wsu.edu, m.namaki@wsu.edu.*
- *Y. Wu is with the Department of Electrical Engineering and Computer Science, Case Western Reserve University, Cleveland, OH, 44106, and the Pacific Northwest National Laboratory, 902 Battelle Blvd, Richland, WA 99354. E-mail: yxw1650@case.edu.*

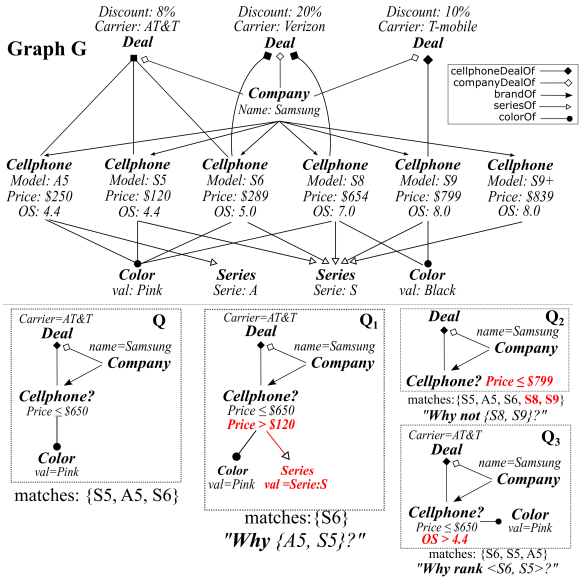


Fig. 1: Why, Why-not and Why-rank: product recommendation.

(1) **"Why?"** She observed two unexpected cellphones with older models $S5$ and $A5$, and may ask a follow-up *Why* question "*Why $S5$ and $A5$ are in $Q(G)$?*" . An answer to such a question can be a revised query Q_1 which properly "tighten" the constraints in Q to exclude $\{A5, S5\}$.

The difference between Q_1 and Q (marked as red) suggests to users a new preference other than Series A (replaced by an edge that specifies "Serie S") and older (cheaper) versions of Series S (by adding a flooring price $120).

(2) **"Why Not?"** She also wonders why two newer models, $S8$ and $S9$, are not included in $Q(G)$, and asks "*Why models $S8$ or $S9$ are not returned?*". A new query Q_2 that clarifies this question properly "compromise" the requirement in Q to include the two entities. The difference between Q_2 and Q reveals that 1) both desired models are more expensive than expected (as the price is relaxed to $790

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2020.3046436, IEEE Transactions on Knowledge and Data Engineering

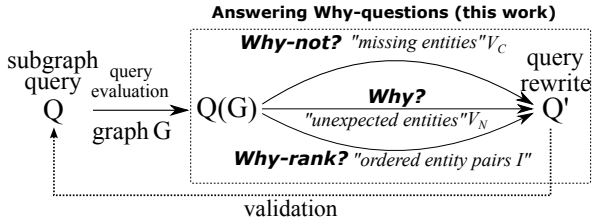


Fig. 2: Exploratory graph search by Why-questions.

in Q_2), and 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 to suggest the necessary trade-off).

(3) **"Why Rank?"** Assume the matches $Q(G)$ are ranked by a function that quantifies the closeness between a cellphone and the predicates on Cellphone in Q. The user also wonders why an older model $S5$ is ranked the highest in $Q(G)$, and poses a *Why-rank* question:"*Why model $S5$ is ranked higher than $S6$?*". A third query Q_3 can be suggested, which ensures an answer that *reorders* the cellphones to be $\{S6, S5, A5\}$. The difference between Q and Q_3 suggests that the attribute OS can be tuned to manipulate the ranking: model $S6$ with a more advanced OS version, thus is ranked higher.

Answering these Why-questions help users to clarify and explore their needs via *e.g.,* query autocompletion [5], and also facilitate providers to promote best sellers [6]. □

These motivate the need for developing effective query rewriting techniques to answer *Why-* questions for *subgraph query*. Given a graph G, a subgraph query Q and answer $Q(G)$, a set of *missing entities* V_C (resp. a set of *unexpected entities* V_N in $Q(G)$), the problem of *answering Why-not (resp. Why) questions* 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. Similarly, given a set I of ordered entity pairs, *answering Why-rank questions* is to compute Q' *w.r.t.* a ranking function, such that the top-k ranked list $Q'(G)$ satisfies the orders enforced by I.

Answering Why-questions also enables exploratory graph search (illustrated in Fig. 2). (1) The query rewrite Q' can be readily suggested to enable interactive search upon users' feedback on undesirable answers. (2) The difference between Q' and its original counterpart Q blends visual querying and approximate search for large G [7]. (3) Query rewrites support graph exploration [8] by suggesting entities relevant to missing and unexpected ones.

Contributions. This paper nontrivially extends [9] by introducing new optimization for Why/Why-Not, solutions for a new class of Why-rank questions, and their variants including Multi-Why, Why-empty and Why-so-many.

(1) We formalize Why-questions in terms of graph query rewrites (Section 3.1). Given a subgraph query Q, graph G, and answers $Q(G)$, (a) A **Why-not** question aims to find a query rewrite Q' with answers that cover missing but desirable entities not in $Q(G)$; (b) A **Why** question finds Q' with answers that exclude undesired entities from $Q(G)$; and (c) A *new class* of **Why-rank questions** compute Q' that can re-rank entities with desired order.

(2) To characterize "good" query rewrites, we generalize *answer closeness* and *query editing cost* in [9] to quantify the closeness between new answers of query rewrites Q' and desired ones, and Q' to original queries, respectively (Section 3.3). The problem is to compute a query rewrite Q' that maximizes answer closeness with bounded query editing cost. We show that the problem remains to be NP-hard for Why, Why-not and Why-rank questions.

(3) Despite the hardness, we present both approximation algorithms that compute query rewrites with quality guarantees, and their faster alternatives for online data exploration. The idea is to quickly identify and verify a set of "picky" operators that can distinguish undesired matches and desired counterparts.

(a) For **Why** questions, we show there is an approximation algorithm (Section 4.1) that can achieve a 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 4.2), we introduce a fast heuristic that computes high-quality query rewrites *without* expensive subgraph isomorphism verification.

We further optimize these algorithms with a new strategy, which incrementally estimates the quality of query rewrites and prunes unpromising operators without enumeration. This improves the efficiency of answering Why questions (resp. Why-not questions) in [9] by 23.2% (resp. 12%) as verified in the experiment study.

(c) In addition, we study three variants of Why-questions including: Multi-Why (with multiple search "focus"), as well as Why-so-many ("Why there are so many entities returned?") and Why-empty ("Why no answer is returned?"), in Sections 5.1 and 5.2, respectively. We show that our approaches and pruning strategy readily extend to these cases with intact performance guarantees, by presenting detailed specifications. These are not discussed in [9].

(d) For **Why-rank** questions, we further develop a fast heuristic. The algorithm dynamically tracks a mapping between applicable query manipulators and the impact of entity ranking, and performs online pruning accordingly to reduce verification cost.

All these algorithms incur a cost that is only determined by Q and its answer, size of desired entities (if specified) and editing budget, which are all small in practice.

(5) Using real-world graphs, we experimentally verify the effectiveness and efficiency of our algorithms and new pruning strategies (Section 7). These algorithms are feasible for large graphs. For example, the approximation (resp. heuristic) algorithm is 12.7 times (resp. 19.5 times) faster compared with the exact algorithm, with up to 12% (resp. 16%) loss of quality. These algorithms also suggest provenance information to "explain" query rewrites and the new answers, as verified by our case study.

2 Graphs and Subgraph Queries

We start with the notions of graphs and subgraph queries.

Graphs. We consider a directed graph $G = (V, E, L, F_A)$ with a finite set of nodes V, 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 A_i with a range $[\min D(A_i), \max D(A_i)]$. It contains all the values of $v.A_i$ 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 properties as in *e.g.,* social networks and knowledge bases.

We next introduce *subgraph queries* with *output nodes* to express practical entity search in graphs.

Queries. A subgraph query Q is a directed, connected 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 $\{>, \geq, =, \leq, <\}$ and c is a constant. (3) Specifically, there is a designated *output node* $u_o \in V_Q$.

In practice, the output node in Q indicates the "search focus", for which the matched entities are returned as answers of Q. Such queries are commonly used to search entities in *e.g.,* social networks [10] and knowledge graphs [11]. For simplicity, we refer to subgraph query as "query".

We use the following conventions. (1) For any literal $u.A$ op c in Q, $c \in [\min D(A), \max D(A)]$. Indeed, literals that enforce "out of active domain" constants either trivially fail (for *e.g.,*'=', $> \max D(A)$, $< \min D(A)$), or are equivalent to their counterparts with c replaced by $\min D(A)$ (for '$>$','$\geq$') or $\max D(A)$ (for '$<$',' $\leq$'). (2) The *size* of query Q (denoted as $|Q|$) refers to the total number of nodes, literals and edges of Q. We focus on queries with a single u_o. Our techniques apply to queries with multiple output nodes (Section 5.1).

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 op c. For example, given a literal $u.A_i \leq c'$, 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 *mapping* from Q to G is an injective function $h \subseteq V_Q \times V$ such that (1) for each node $u \in V_Q$, $h(u)$ is a candidate of u, and (2) for each edge $e = (u, u')$ in Q, $e' = (h(u), h(u'))$ is an edge in G, and $L(e) = L(e')$. The *matches* of u, denoted as $Q(u, G)$, refers to the set of nodes $h(u)$ in G with h ranges over all the mappings from Q to G. The *answer* of Q in G, denoted as $Q(u_o, G)$, refers to the matches of the output node u_o of Q in G ($Q(u_o, G) \subseteq V$).

Top-k matches. We next consider a total order of $Q(u_o, G)$ induced by a *ranking function* $R(Q, v)$. Following top-k selection queries [12], the function $R(Q, v)$ maps each match $v \in Q(u_o, G)$ to a ranking score, by aggregating scores of tuple $F_A(v)$ *w.r.t.* predicates $F_Q(u_o)$:

$$R(Q, v) = \frac{1}{|F_Q(u_o)|} \sum_{l \in F_Q(u_o)} r(l, v.A)$$

where $r(l, v.A)$ is a polynomial-time computable function that computes a "closeness" score between predicate l = $u_o.A$ *op* c and attribute value $v.A$. For example, $r(l, v.A)$ can be defined in terms of normalized Euclidean distance between constant c and attribute value $v.A$ for numerical attributes, or text similarity for string attributes [13].

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
$Q_k(u_o, G)$	top-k query answer of Q in G
(u_o, V_{C_u})	Why-not (w. missing nodes V_{C_u})
(u_o, V_{N_u})	Why question (w. unexpected nodes V_{N_u})
(u_o, I)	Why-rank question (w. ordered node pair list I)
Q'=$Q \oplus O$	A query rewrite induced by operator set O
$c(O)$	Editing cost of operator set O
cl(O)	Answer closeness for Why-questions
$N_d(V_{N_u})$	d-hop neighbors of the nodes in V_{N_u}

TABLE 1: Notations

Given a ranking function R, the *top-k matches* of a query Q in G, denoted as $Q_k(u_o, G)$, refers to the top-k matches of u_o in G ranked by R ($Q_k(u_o, G) \subseteq Q(u_o, G)$).

Example 2: The query Q illustrated in Fig. 1 designates an output node u_o with label Cellphone to searches for cell phones. Assume the closeness function $r(l, v.A)$ is defined as the difference between constant c in l and $v.A$ normalized by the range of active domain of A in G. One can verify that the top 3 matches $Q_3(u_o, G)$ = {$S5$, $A5$, $S6$}, where $S5$ has a ranking score $\frac{650-120}{839-120} = 0.737$; similarly, $A5$ and $S6$ have ranking scores 0.556 and 0.5, respectively. □

3 Why-Questions for Subgraph Queries

3.1 Categorization of Why-Questions

Given a graph G = (V, E, L, F_A), a query Q = (V_Q, E_Q, L_Q, F_Q, u_o), and answer $Q(u_o, G)$, we study three classes of Why-questions, namely, "*Why*", "*Why-not*", and "*Why-rank*". These questions have counterparts for relational queries [14]–[19], and are specialized for graph search.

Why-not. A Why-not question is a pair (u_o, V_{C_u}), where (1) u_o is a designated output node of Q, and (2) $V_{C_u} \subseteq V \setminus Q(u_o, G)$ is a set of "missing matches" of u_o. It clarifies missing answers in $Q(u_o, G)$ by asking "*Why the nodes in V_{C_u}, are not matches of u in Q?*". In practice, V_{C_u} can be designated as examples [20] or by *e.g.,* keyword search [8].

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

Why-rank. A *Why-rank question* is defined as a pair (u_o, I), where I is a set of ordered pairs. For each pair $\langle v_i, v_j \rangle \in I$, v_i and v_j are in $Q_k(u_o, G)$, and $R(Q, v_i) < R(Q, v_j)$. The pair $\langle v_i, v_j \rangle$ suggests that user expects v_i to be ranked at least as high as v_j, which does not hold in the current ranked top-k answer $Q_k(u_o, G)$. The question asks "*Why the node pairs in I are not ranked as expected in $Q_k(u_o, G)$?*".

Remarks. We clarify a practical assumption that the nodes in I are also in $Q(u_o, G)$. Indeed, for any non-matches in I, a Why-not question can be first processed to obtain Q' with answers that include the nodes in I as much as possible. One can then refine Q and I for a follow-up Why-rank question.

Example 3: The three questions in Example 1 can be expressed by a Why question (Cellphone, $\{A5, S5\}$) (Fig. 1), a Why-not question (Cellphone, {S8, S9}), and a Why-rank question (Cellphone, $\{\langle S6, S5 \rangle\}$), respectively. □

3.2 Answers for Why-Questions

We approach *query rewriting* to answer Why-questions for graph search. The technique has been applied to explain unexpected answers in relational query processing [19].

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.

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 operators 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 Q, $Q(u_o, G)$ and G, a query rewrite $Q' = Q \oplus O$ is an *answer*

- for a Why-not question (u_o, V_{C_u}), if $Q'(u_o, G) \cap V_{C_u} \neq \emptyset$, *i.e.,* at least a missing match is introduced by Q',
- for a Why question (u_o, V_{N_u}), if $V_{N_u} \setminus Q'(u_o, G) \neq \emptyset$, *i.e.,* at least an undesired match is removed from the match list, and
- for a Why-rank question (u_o, I), there is a pair $\langle v_i, v_j \rangle \in I$, such that $R(Q', v_i) \geq R(Q', v_j)$, *i.e.,* the order of v_i and v_j is "reversed" as requested.

Query rewrites provide a natural way to explain answers. The editing operators that modify Q to Q' suggest query manipulations that are responsible for "missing", "unexpected", or "undesirably ranked" matches.

The capacity of query operators and their impact to the matches has been clarified by the following property.

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.* □

We shall use relaxation (resp. refinement) operators to answer Why-not (resp. Why) questions. For Why-rank questions, we use both relaxation and refinement operators.

3.3 Quality Measures of Query Rewrites

A "good" query rewrite Q' should identify desirable answers as required by the Why-questions with a small difference compared with the original query Q. We extend the closeness measures in [9] for top-k queries.

Query editing cost. We follow the intuition that the modification to more "important" fraction (closer to u_o) of Q should be more "expensive". Given Q with output node u_o, denote the diameter of Q as d_Q, the *output centrality* [9] of a 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 [21]).

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

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

The unit cost $c(o)$ for a single operator $o \in O$ is defined as follows. (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 $e=(u, u')$, $c(o) = \min(\mathsf{oc}(u, u_o), \mathsf{oc}(u', u_o))$. (3) 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 $w(o) = 1 + \frac{|c'-c|}{\max D(A) - \min D(A)}$, normalized by the range of $D(A)$.

The editing cost penalizes operators that are "closer" to the output node or introduce larger value differences on literal constants. The smaller $c(O)$ is, the better. Our general cost model and techniques also apply to disconnected query rewrites (due to RmE; see Section 4).

Answer closeness. We next extend answer closeness [9] to top-k queries. We take a similar treatment for Why and Why-not questions, and introduce a new closeness measure for Why-rank questions. Given $Q(u_o, G)$ and a Why-question, the *answer closeness* of a query rewrite $Q' = Q \oplus O$, denoted simply as $\mathsf{cl}(O)$, 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 larger, the better.

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

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

which measures the fraction of new matches in V_{C_u} that are introduced in $Q'(u_o, G)$; the more, the better.

Quantifying "Overdo". A query may be "overly" refined for Why questions, thus unnecessarily excluding many matches from $Q(u_o, G)$ that are not among V_{N_u}. Similarly, it can be overly relaxed to introduce many matches to $Q(u_o, G)$ that are not among V_{C_u}. To characterize the tolerance of such overdo, we define a *guard ratio* guard for a set of updates O. Specifically, for a Why question (u_o, V_{N_u}),

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

Similarly, for a Why-not question (u_o, V_{C_u}),

$$\mathsf{guard}(O) = \frac{|Q'(u_o, G) \setminus (Q(u_o, G) \cup V_{C_u})|}{|Q(u_o, G)|}$$

The guard ratio quantifies the fraction of the entities "unnecessarily" added or removed from original answer $Q(u_o, G)$, the smaller, the better. We shall use a *guard condition* to identify query rewrites Q'=$Q \oplus O$ with $\mathsf{guard}(O) \leq \delta$, for a user-specified guard threshold δ.

Example 4: The query Q_1 (Fig. 1) answers the Why question 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 = {RmE(Cellphone, Color), RxL(Cellphone.price, \$799), RmL(Deal, carrier)}, cost $c(O_2)$ = 4.2, and answer closeness $\mathsf{cl}(O_2, \{S8, S9\})$ = 1. □

(3) For Why-rank, the query rewrite should ensure a ranking that is consistent with the ordered pairs enforced by I. The closeness can be naturally defined as cl (O,I) = $\frac{|L'|}{|L|}$, where $L' = \{\langle v_i, v_j\rangle \in L \mid R(Q', v_i) \geq R(Q', v_j)\}$. Intuitively, it quantifies the fraction of the ordered pairs in L which are consistently ranked among the matches identified by Q'.

Example 5: The query Q_3 (Fig. 1) answers the Why-rank question with operators $O_3 = \{\mathsf{AddL}(\mathsf{Cellphone.OS} > 4.4)\}$. We can verify that $c(O_3)$=2 and $\mathsf{cl}(O_3, \{\langle S6, S5\rangle\})$=1. □

3.4 Answering Why-questions

Given a query Q, answer $Q(u_o, G)$ or $Q_k(u_o, G)$, graph G, a Why-question W, and an editing budget B, the problem of *answering why-question* is to compute an operator set O^*:

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

where set Δ refers to the missing matches V_{C_u}, the undesired matches V_{N_u}, or the ranked pairs I to be enforced by W as a Why-not, a Why, or a Why-rank question, respectively. 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 already nontrivial even when only edge operators are considered.

Theorem 2: *Answering Why, Why-not and Why-rank questions are already* NP*-hard when only edge removing (*RmE*) or edge insertion (*AddE*) are involved.* □

Proof: Given cost bound B and closeness threshold θ, the decision problem is to decide whether there exists an operator set O^* such that $Q' = Q \oplus O^*$ has editing cost within B and answer closeness at least θ.

(1) We show a tighter lower bound for Why-not compared with [9]: it is already NP-hard when only edge removing is involved and queries have *tree patterns*. We construct a reduction from subgraph isomorphism between a tree and a general graph (known to be NP-hard).

Given a tree Q_0 and a general graph $G_0 = (V_0, E_0, L)$, we construct a new tree query Q that contains (a) two subtrees Q_1 and Q_2 that are isomorphic to Q_0, and (b) a bridge (u_1, u_2) that connects the root of Q_1 to the root of Q_2, with a unique label r not in G, where u_1 in Q_1 is set to be the output node u_o of Q. One can verify that Q has a tree pattern. We construct a graph G' with two disconnected components G_0 and G_1, where G_1 is isomorphic to G_0. Clearly, $Q(u_o, G) = \emptyset$. We next set constrained operators that remove edges at unit cost, budget B = 1, C=$\emptyset$, V_{C_u} = V_0 and $\delta = \frac{1}{|V_{C_u}|}$. To answer Why-not defined on Q, G and V_{C_u}, one has to apply an RmE that removes the edge (u_1, u_2) from Q. We can verify that Q' obtained by removing edge (u_1, u_2) answers W with closeness at least θ, if and only if there exists at least a subgraph isomorphism from Q_0 to G_0.

(2) To see the hardness of answering Why questions, we construct a reduction from k-clique problem. The following proof is for a general case where there is no *guard condition*.

Given a graph $G = (V, E, L)$, the k-clique problem is to decide whether there exists a clique that contains k nodes. (a) We construct a query Q_0 that contains a single independent output node u_o with a unique label l_o not in G, and a sub-query Q_c as a k-clique with all query nodes associated with a label '_' that can match any node in G. (b) We construct a graph G' that contains a node v_o with a unique label l_o not in G, graph G, and an edge from v_o to every node in G. (c) We set B=1, $V_{N_u} = \{v_o\}$, δ = 1, and a constrained operator that inserts edge only. For Q_0, it has to be a single edge insertion from u_o to any node in Q_c, given B=1. We can verify that there exists a query rewrite Q' that answers Why question $(u_o, \{v_o\})$ if and only if G *does not* contain a k-clique. As k-clique problem is known to be NP-hard and APX-hard, answering Why question is also NP-hard and APX-hard.

(3) We observe that Why-not questions are a special cases of Why-rank where I enforces a node in V_{C_u} to be ranked higher than at least a node from $Q(G)$. Specifically, given a Why-not question $W_1 = (u_o, V_{C_u})$ defined on answer $Q(G)$ with a closeness threshold θ, we construct a Why-rank question $W_2 = (u_o, I)$ with threshold δ', initial query Q_r and answers $Q_r(G_r)$ as follows. (a) $k = |Q(G)| + |V_{C_u}|$, $Q_r = Q$, and $G_r = G$. (b) We set a ranking function R defined on an auxiliary attribute of the nodes $Q(G) \cup V_{C_u}$ such that no node in $Q(G)$ is ranked higher than any node in V_{C_u}. (c) $\theta' = \frac{\theta}{|Q(G)|}$, and $I = \{(v, v') \mid v \in V_{C_u}, v' \in Q(G)\}$.

Assume there exists a query rewrite Q'_r for Why-rank with closeness at least θ'=$\frac{\theta}{|Q(G)|}$. As $|I| = |Q(G)||V_{C_u}|$, at least $\theta'|I| = \theta|V_{C_u}|$ pairs are re-ranked to follow the order posed by I. Given the definition of I, at least $\theta|V_{C_u}|$ nodes in V_{C_u} are ranked higher than at least a node in $Q(G)$ in the top k=$|Q(G)|+|V_{C_u}|$ matches. Set $Q'_r = Q'$. It follows that Q' is a solution for W_1 that ensures a closeness at least $\frac{\theta|V_{C_u}|}{|V_{C_u}|} = \theta$. Similarly, a solution for W_1 also reduces to a counterpart for W_2. The hardness of Why-rank thus follows. □

4 Answering Why and Why-not Questions

4.1 Approximating Answers for Why

We present our main result for Why questions below.

Theorem 3: *There is an algorithm for answering Why questions (O^* refers to the optimal set) which:*

- *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*
- *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 $\epsilon \in [0,1]$ 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 show Theorem 3, we next show the connection between answering Why questions and the budgeted submodular maximization problem [22].

Submodularity. Given refinement operators 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. We say an operator is *picky* if it is *likely* to prune at least a node in V_{N_u}.

Lemma 4: *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* $o \in \mathcal{O}_s$ $(o \notin O_2)$, $\mathsf{mg}(O_1,o) \geq \mathsf{mg}(O_2,o)$. □

Proof: A set function f is said to be **submodular** on a set $\mathcal{O}$ if for $O_1 \subseteq O_2 \subseteq \mathcal{O}$ and $o \in \mathcal{O}$

$$\mathsf{cl}(O_1 \cup \{o\}) - \mathsf{cl}(O_1) \geq \mathsf{cl}(O_2 \cup \{o\}) - \mathsf{cl}(O_2)$$

Here, $\mathsf{cl}(O) = \frac{|Q'(u,G) \cap V_{N_u}|}{|V_{N_u}|}$ represents the number of nodes in V_{N_u} that has been removed from $Q(u_o,G)$ given operator set O. $\mathsf{cl}(O \cup \{o\}) - \mathsf{cl}(O)$ refers to the normalized number of nodes in V_{N_u} that has been removed from the match set given operator o. To show the submodularity, it suffices to show that if a node in V_{N_u} is removed from $Q_{O_2}(u_o,G)$, it is also removed from $Q_{O_1}(u_o,G)$ given same operator o. Here $Q_{O_i} = Q \oplus O_i$, and Q_{O_i} could be a disconnected query.

We next prove the submodularity by contradiction. If a node v has been removed from $Q_R(u_o,G)$ given operator o, which means $v \in Q_{O_2}(u_o,G)$ but $v \notin Q_{O_2 \cup \{o\}}(u_o,G)$. Assume v has not been removed from $Q_{O_1}(u_o,G)$ with o. Then we have $v \in Q_{O_1}(u_o,G)$ and $v \in Q_{O_1 \cup \{o\}}(u_o,G)$. Given Lemma 1, it contradicts that if $v \notin Q_{O_2 \cup \{o\}}(u_o,G)$ then $v \notin Q_{O_1 \cup \{o\}}(u_o,G)$. Based on Lemma 1, the above discussion holds for both connected and disconnected queries. Given these, Lemma 4 follows. □

The submodularity suggests that we may approximate the optimal query rewrites by solving an instance of *budgeted submodular maximization* problem [9], [22]. 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. A greedy algorithm ensures a $1 - \frac{1}{e}$ approximation by greedily selecting operators with maximum mg [23]. This nevertheless requires $O(|\mathcal{O}_s|^2)$ subgraph isomorphism tests, and is infeasible for large G.

We can do better: our algorithm reduces unnecessary verification of operators, by (1) estimating $\mathsf{cl}(\cdot)$ of the currently selected and verified picky operators, and (2) dynamically deciding and pruning unpromising operators.

Algorithm. The algorithm, denoted as ApproxWhy and illustrated in Figure 3, has the following steps.

(1) It invokes a procedure GenPicky to generate a set of refinement picky operators $\mathcal{O}_s$ (line 1). It then initializes sets $\mathcal{O}_1$ and $\mathcal{O}_2$ (lines 2-3), where $\mathcal{O}_1$ contains a single operator with the maximum $\mathsf{cl}(\cdot)$ verified by Match.

Algorithm ApproxWhy
Input: graph G, query Q, cost bound $B > 0$,
a Why question W=(u_o, V_{N_u}), guard threshold δ;
Output: the optimal operator set $\mathcal{O}'$ that answers W.
1. $\mathcal{O}_s$:= GenPicky(Q, G, V_{N_u}, B);
2. set $\mathcal{O}_1 := \arg\max\{\mathsf{cl}(\{o\})\} : o \in \mathcal{O}_s, c(o) \leq B,$ $\mathsf{guard}(\{o\}) \leq \delta$ };
3. set $\mathcal{O}_2 := \varnothing$; set $\mathcal{O}'_s = \mathcal{O}_s$;
/*greedy selection of refinement operators*/
4. **while** $\mathcal{O}'_s \neq \varnothing$ **do**
5. **for** $o \in \mathcal{O}_s$ **do** $\hat{\mathsf{mg}}(o) := \hat{\mathsf{cl}}(\mathcal{O}_2 \cup \{o\}) - \hat{\mathsf{cl}}(\mathcal{O}_2)$;
6. $o^* := \arg\max\{\frac{\hat{\mathsf{mg}}(o)}{c(o)} : o \in \mathcal{O}_s\}$;
7. **if** $c(\mathcal{O}_2)$+$c(o^*) \leq B$ **and** $\mathsf{guard}(\mathcal{O}_2 \cup \{o^*\} \leq \delta$ **then**
8. $\mathcal{O}_2 := \mathcal{O}_2 \cup \{o^*\}$; $\mathcal{O}'_s := \mathcal{O}'_s \setminus \{o^*\}$;
/*prune conflicting operators for selected o^**/
9. $\mathcal{O}_s := \mathcal{O}_s \setminus \mathsf{Confl}(o^*))$; $\mathcal{O}'_s := \mathcal{O}'_s \setminus \mathsf{Confl}(o^*))$;
/*prune more constrained counterparts of discarded o^**/
10. **if** $\mathsf{guard}(\mathcal{O}_2 \cup \{o^*\}) > \delta$ **then**
11. $\mathcal{O}_s := \mathcal{O}_s \setminus \mathsf{Const}(o^*)$; $\mathcal{O}'_s := \mathcal{O}'_s \setminus \mathsf{Const}(o^*)$;
12. $\mathcal{O}' = \arg\max_{\mathcal{O}_i \in \{\mathcal{O}_1, \mathcal{O}_2\}} \hat{\mathsf{cl}}(O_i)$;
13. **return** $\mathcal{O}'$;

Fig. 3: Algorithm ApproxWhy

(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 4-8). 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 operator o^* is selected if the query editing cost is bounded by B, and the guard condition holds (line 8).

During the process, ApproxWhy applies the following *pruning strategy* (lines 9-11; see "Optimization"): (a) if an operator o^* is selected, it prunes all its *conflicting set* (Confl) from the candidate set which leads to empty answers; and (b) if o^* is to be discarded due to violating the guard condition, it prunes all its *constrained set* (Const) which will also violate the guarded condition. The process repeats until all the picky operators are processed.

(3) It then set and return $\mathcal{O}'$ as the set of $\mathcal{O}_1$ or $\mathcal{O}_2$, whichever has larger estimated answer closeness (lines 13-14).

Procedure GenPicky. 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, GenPicky generates AddE first, followed by AddL and RfL. It 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.

GenPicky applies the following rules.

Generating AddE. GenPicky 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), such that $v_1 \in N(Q(u_o,G),u_1)$, 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 include 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. For query 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 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 $a \in$ dom($A, N(V_{N_u}, u')$), and $c \geq a$, add RfL(l, $u'.A < a$);
- if op $\in \{>, \geq\}$, for each $a \in$ dom($A, N(V_{N_u}, u')$) and $c \leq a$, add RfL($l, u'.A > a$);
- if op is '=', for each $a \in$ 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 GenPicky 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 6: 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>$120, is generated by GenPicky as follows. (1) It first follows rules for adding AddE, and add o_1 ={AddE(Cellphone, Series). As a part of composite operator, it also adds o_4 = AddL(Series.val, '$\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).

GenPicky 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. □

Procedure EstMatch. Given operators O and a single operator o, EstMatch efficiently estimates cl(O) and cl($O \cup \{o\}$) (as $\hat{\text{cl}}(O)$ and $\hat{\text{cl}}(O \cup \{o\})$, respectively) and computes $\hat{\text{mg}}(o)$ as $\hat{\text{cl}}(O \cup \{o\})$ - $\hat{\text{cl}}(O)$, without actual query processing by subgraph isomorphism tests.

(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 cl(o) is computed (line 2). (2) It then estimates Aff(V_{N_u}), a fraction of V_{N_u} that becomes non-matches due to O. (a) Given O, it first sets Aff(V_{N_u}) = $\bigcup_{o \in O}$(Aff(o) $\cap V_{N_u}$), *i.e.,* all the non-matches of u already identified by Aff(o). (b) It then extends $\overline{\text{Aff}}(V_{N_u})$ and updates $\hat{\text{cl}}(O)$, by checking if the nodes in $\overline{\text{Aff}}(V_{N_u})$ = $V_{N_u} \setminus$ Aff(V_{N_u}) becomes a non-match. This is efficiently conducted by consulting a path index [9] that samples paths from Q and verifies path matches. The step continues until all the nodes in $\overline{\text{Aff}}(V_{N_u})$ are processed.

Example 7: Consider the picky set $\mathcal{O}_s$ in Example 6. 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 $\frac{\hat{\text{mg}}(o_2)}{c(o_2)}$ = 0.5. Other picky operators (*e.g.*, o_8 = AddL(Cellphone.Price > $250)) are not as good (*e.g.*, $\hat{\text{mg}}(o_8)$=0) given o_3, thus are not selected. □

Optimization. We next introduce the pruning techniques.

Early Pruning. GenPicky extends its counterpart in [9] by early pruning the picky operators that lead to violation of the guard conditions. It performs a case analysis of a picky operator o to estimate a lower bound of the guard ratio guard(o), which is defined as guard(o) = $\frac{|Q^r(u_o,G)|}{|Q(u_o,G)|}$, where $Q^r(u_o, G)$ refers to a fraction of the matches in $Q(u_o, G) \setminus V_{N_u}$ that will be removed if o is applied. If the lower bound $\frac{|Q^r(u_o,G)|}{|Q(u_o,G)|} > \delta$, o is pruned. GenPicky identifies $Q^r(u_o, G)$ in polynomial time with the following cases.

operator o	$Q^r(u_o, G)$
AddL, RfL or AddE at u_o	non-candidates of u_o
AddL or RfL at u ($u \neq u_o$)	$\{v \mid u$ has no candidate in $N_{d_Q+1}(v)\}$
AddE(u, u')	$\{v \mid (u, u')$ has no candidate in $N_{d_Q+1}(v)\}$

Pruning "Conflict" operators (lines 10, ApproxWhy). If o^* is selected to be applied, ApproxWhy induces a *conflict operator set* Confl(o^*) $\subseteq \mathcal{O}_s$, where for each operator $o \in$ Confl(o^*), $Q \oplus \{o^*, o\}(G) = \emptyset$. That is, applying both o and o^* to Q leads to a query rewrite with empty answer. Specifically, ApproxWhy computes Confl(o^*) as follows.

operator o^*	Confl(o^*)
AddL($u.A$>c)	{AddL($u.A$ op c') |op $\in \{<, \leq, =\}, c \geq c'$}
AddL($u.A$<c)	{AddL($u.A$ op c') |op $\in \{>, \geq, =\}, c \leq c'$}
AddL($u.A$=c)	{AddL($u.A = c'$)|$c \neq c'$} $\cup$ {AddL($u.A$ op c'), RfL($u.A$ op $c_2, u.A$ op c')|c op' c', (op, op') $\in \{(>, \leq), (\geq, <), (<, \geq), (\leq, >)\}$}
RfL($u.A$>c_1, $u.A$>c)	RfL($u.A$ op $c_2, u.A$ op c') |op $\in \{<, \leq\}, c \geq c'$}
RfL($u.A$<c_1), $u.A$<c)	RfL($u.A$ op $c_2, u.A$ op c') |op $\in \{>, \geq\}, c \leq c'$}

For o^* (AddL, RfL) with comparison operators $\geq$ (resp. $\leq$), the conflict set Confl($o*$) can be inferred by posing

small changes to its counterpart with $>$ (resp. $<$). For example, if o^*=AddL($u.A \geq c$), Confl(o^*)={AddL($u.A$ op c') |op $\in \{<, \leq, =\}, c>c'$}. If applying two operators o_1 and o_2 achieve a same query rewrite, for example AddL($u.A>c$) and RfL($u.A>c_1$, $u.A>c$), their conflict operator set can be shared, *i.e.*, Confl(o_1) = Confl(o_1) $\cup$ Confl(o_2).

Pruning "Constrained" operators (lines 11-12, ApproxWhy). If applying o^* already violates the guard condition, enforcing any operator that is "more constrained" than o^* also violates the guard condition (Lemma 1). ApproxWhy induces a set of *constrained operator set* Const(o^*) $\subseteq \mathcal{O}_s$ as follows.

operator o^*	Const(o^*)
AddL($u.A>c$)	{AddL($u.A$ op c'), RfL($u.A$ op $c_2, u.A$ op c'), op $\in \{>, \geq\} \mid c<c'$}
AddL($u.A<c$)	{AddL($u.A$ op c'), RfL($u.A$ op $c_2, u.A$ op c') op $\in \{<, \leq\} \mid c>c'$}

We can verify that RfL($u.A>c_1$, $u.A>c$), AddL($u.A \geq c$), RfL($u.A \geq c_1$, $u.A \geq c$) (resp. RfL($u.A<c_1$, $u.A < c$), AddL($u.A \leq c$), RfL($u.A \leq c_1$, $u.A \leq c$)) all have the same constrained set as AddL($u.A>c$) (resp. AddL($u.A<c$)).

Performance Analysis. The approximation can be verified by constructing a reduction from our problem to the budgeted submodular maximization problem with *estimated marginal gain* [22], when exact marginal gain cannot be efficiently evaluated. Given a fixed operator set $\mathcal{O}$ and a cost function that assigns each operator $o \in \mathcal{O}$ a cost, (1) An optimal set O^* of induces an optimal query rewrite Q'^* with maximized closeness. (2) If EstMatch estimates mg(O) with a bounded error ϵ', *i.e.*, $|\hat{\text{mg}}(O) - \text{mg}(O)| \leq \epsilon'$, then the submodularity function can be maximized with guarantee $\frac{1}{2} \cdot (1 - \frac{1}{e})$ and a factor $(\frac{B(d_Q+2)}{d_Q})\epsilon'$ [22]. As EstMatch ensures $|\hat{\text{cl}}(Q', V_{N_u}) - \text{cl}(Q', V_{N_u})| \leq \epsilon$, let $\hat{\text{mg}}(O) = \hat{\text{cl}}(Q', V_{N_u}) - \hat{\text{cl}}(Q, V_{N_u})$, thus $\hat{\text{mg}}(O)$ approximates mg(O) within error $\epsilon' = 2\epsilon$. ApproxWhy thus ensures to find Q' that approximates Q^* with $\text{cl}(Q', V_{N_u}) \geq \frac{1}{2} \cdot (1 - \frac{1}{e}) \cdot \text{cl}(Q^*, V_{N_u}) - 6B\epsilon$.

We next analyze the time cost of ApproxWhy. 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 $|Q|$. GenPicky 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 $O(|N_{d_Q+1}(Q(u_o, G))|)$. The initialization of set O_1 (line 2) requites $|\mathcal{O}_s|$ times of Match and thus takes time $O(|\mathcal{O}_s||N_{d_Q+1}(V_{N_u})|^{|Q|})$. The iterative selection process uses $|\mathcal{O}_2|$ estimations of $\hat{\text{mg}}(O^*)$, and the estimation takes time $O(|N_{d_Q+1}(V_{N_u})|)$. ApproxWhy thus takes 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 3.

4.2 Answering Why-not

Unlike Why questions, the answer closeness under relaxation for Why-not questions is no longer submodular, and is hard to approximate even with an oracle that reports cl($\cdot$) [23]. We next introduce a fast heuristic algorithm.

Algorithm. The general idea is to compute an operator set 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, performs the following. (1) It invokes a procedure GenPicky to generate picky operators $\mathcal{O}_s$, 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), and greedily selects $\mathcal{O}'$.

Procedure GenPicky. Similar to its counterpart in ApproxWhy, procedure GenPicky generates picky 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 GenPicky 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. GenPicky simply adds RmE(e) (resp. RmL(l)) to $\mathcal{O}$ for each edge e and literal l in Q.

Example 8: Given query Q and the Why-not question that specifies V_{C_u} as $\{S8, S9\}$ (Figure 1), GenPicky 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. □

Match estimation. 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 [9] 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.

Early Pruning. FastWhyNot uses a similar intuition as in ApproxWhy to prune operators that violate the guard condition. The difference is that it capitalizes on Lemma 1 defined on relaxation. Specifically, for each relaxation operator o that is pruned due to the violation of the guard condition, it prunes all the operators in $\mathcal{O}_s$ that are "more relaxed" than o. This set of operators, denoted as Rel(o^*) $\subseteq \mathcal{O}_s$, is generated using the following rules.

operator o	Rel(o^*)
RxL($u.A$ op c_1, $u.A$ op c), op $\in \{>, \geq\}$	{RxL($u.A$ op c_2, $u.A$ op c') |op $\in \{>, \geq\}, c>c'$}
RxL($u.A$ op c_1, $u.A$ op c), op $\in \{<, \leq\}$	{RxL($u.A$ op c_2, $u.A$ op c') |op $\in \{<, \leq\}, c<c'$}
RmL($u.A$ op c)	{RmL($u.A$ op$'$ c')| any op$'$ and c'}

Time cost. Following the analysis of ApproxWhy, FastWhyNot takes $O(|Q||N_{d_Q}(V_{C_u})| + (|\mathcal{O}_s|)^2|N_{d_Q}(V_{C_u})|)$ time to compute $\mathcal{O}_s$ and performs greedy selection.

5 VARIANTS OF WHY AND WHY-NOT

We next discuss several practical variants of Why-questions.

5.1 Multi-Why-Questions

A query may contain multiple output nodes with answers to be clarified. For example, (1) a user may specify both "Cellphones" and "Deal" as output nodes (Figure 1) (by enforcing *e.g.,*'Construct' semantics of SPARQL [24]) for answer clarification; (2) multiple users may access the same query and enforce different search focus. We start with a characterization of Why-questions for such needs.

Multi-Why-Questions. A *Multi-Why-not* question is a pair $(\mathcal{U}, V_C)$, where (1) $\mathcal{U}$ is a set of output nodes in Q, (2) V_C=$\bigcup_{u_o \in \mathcal{U}} V_{C_u}$, where for each output node $u_o \in \mathcal{U}$, a set of "missing matches" V_{C_u} of u_o is specified. Similarly, A *Multi-Why* question is a pair $(\mathcal{U}, V_N)$, where V_N=$\bigcup_{u_o \in \mathcal{U}} V_{N_u}$ is the set of unexpected matches.

Answer Closeness. Given a Multi-Why question $(\mathcal{U}, V_N)$, where $Q(\mathcal{U}, G) = \bigcup_{u_o \in \mathcal{U}} Q(u_o, G)$ and there is no overlap between nodes in each V_{N_u}, we extend the notion of the answer closeness $\mathsf{cl}_m(O, V_N)$ as

$$\mathsf{cl}_m(O, V_N) = \frac{|(Q(\mathcal{U}, G) \setminus Q'(\mathcal{U}, G)) \bigcap V_N|}{|V_N|}$$

Similar to answering Why questions, a *guard ratio* is defined as $\mathsf{guard}(o) = |(\bigcup_{u_o \in \mathcal{U}}(Q(u_o, G) \setminus Q'(u_o, G))) \cap (\bigcup_{u_o \in \mathcal{U}} Q(u_o, G) \setminus V_N)| / |\bigcup_{u_o \in \mathcal{U}}(Q(u_o, G)) \setminus V_N)|$.

Similarly, given a Multi-Why-not question $(\mathcal{U}, V_C)$, where $V_C = \bigcup_{u_o \in \mathcal{U}} V_{C_u}$, $Q(\mathcal{U}, G) = \bigcup_{u_o \in \mathcal{U}} Q(u_o, G)$, we define the closeness $\mathsf{cl}_m(O, V_C)$ as

$$\mathsf{cl}_m(O, V_C) = \frac{|Q'(\mathcal{U}, G)) \bigcap V_C|}{|V_C|}$$

The *guard ratio* for answering Multi-Why-not questions is defined as $\mathsf{guard}(O) = |\bigcup_{u_o \in \mathcal{U}}(Q'(u_o, G) \setminus (Q(u_o, G) \cup V_{C_u}))| / |\bigcup_{u_o \in \mathcal{U}}(Q(u_o, G))|$.

Problem Statement. Given a query Q, answer $Q(\mathcal{U}, G)$, graph G, a Multi-Why-question W and an editing budget B, the problem of *answering Multi-Why-question* is to compute an operator set O^* such that

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

where set $V_{\mathcal{U}}$ refers to V_C (resp. V_N) for a Multi-Why-not question (resp. Multi-Why question) W with a bounded guard ratio. We can verify that answering Multi-Why and Multi-Why-not questions are both NP-hard.

Answering Multi-Why. One may process each missing match set "one-by-one". This alone does not ensure good queries that can optimize the overall answer closeness. We outline an algorithm that processes the match sets in a single batch with desirable quality guarantee.

We start by showing that the extended closeness $\mathsf{cl}_m(\cdot)$ for Multi-Why preserves the submodularity.

Lemma 5: $\mathsf{cl}_m(\cdot)$ *is submodular over picky set* $\mathcal{O}_s$. □

Proof: As the sets V_{N_u} specified for each output node are pairwisely disjoint, $\mathsf{cl}_m(O) = \frac{\sum_{u \in \mathcal{U}} \mathsf{cl}(O)|V_{N_u}|}{|V_N|}$ is a linear combination of $\mathsf{cl}(O)$. Since $\mathsf{cl}(O)$ is submodular (Lemma 4), $\mathsf{cl}_m(O)$ is also submodular for refinement operators. □

Approximation. We next outline an approximation algorithm, denoted as MultiApproxWhy. MultiApproxWhy follows the similar steps as ApproxWhy. There are several differences. (1) It invokes GenPicky to generate operators by exploring at most $d_Q + 1$ hop neighbors of nodes in $Q(\mathcal{U}, G)$. (2) $\mathsf{cl}_m(\cdot)$ is used instead of $\mathsf{cl}(\cdot)$ and EstMatch is used to estimated the marginal gain $\hat{\mathsf{mg}}_m(o) = \hat{\mathsf{cl}}_m(\mathcal{O}_2 \cup \{o\}) - \hat{\mathsf{cl}}_m(\mathcal{O}_2)$.

Analysis. MultiApproxWhy ensures closeness at least $\frac{1}{2}(1 - \frac{1}{e})\mathsf{cl}_m(O^*, V_N) - 6B\epsilon$. This can be verified by a reduction to budgeted submodular maximization following the analysis of ApproxWhy. For each literal and edge in $N_{d_Q+1}(Q(\mathcal{U}, G))$, GenPicky generates picky operators in $O(|Q||N_{d_Q+1}Q(\mathcal{U}, G)|)$ time. The initialization takes $O(|\mathcal{O}_s||N_{d_Q+1}(V_N)|^{|Q|})$ time. The estimation of each $\hat{\mathsf{mg}}(o)$ takes $O(|N_{d_Q+1}(V_N)|)$ time. Thus MultiApproxWhy is in $O(|Q||N_{d_Q+1}(Q(\mathcal{U}, G))| + |\mathcal{O}_s||N_{d_Q+1}(V_N)|^{|Q|} + |\mathcal{O}_s|^2|N_{d_Q+1}(V_N)|)$ time.

Answering Multi-Why-not. Algorithm FastWhyNot can be readily extended to handle the case where there are multiple Why-not questions, where user give a set of query nodes $\mathcal{U}$ and for each query nodes a set of missing matches. We extend algorithm FastWhyNot to answer Multi-Why-not questions. The algorithm, denoted as MultiFastWhyNot, invokes GenPicky to generate picky operators from $N(V_C, u')$ for all $u' \in \mathcal{U}$. It then iteratively selects an operator o^* that (1) maximizes the ratio of $\mathsf{cl}_m(o^*)$ to estimated marginal gain, and (2) satisfies the guard condition. Following a similar analysis of MultiApproxWhy, MultiFastWhyNot is in $O(|Q||N_{d_Q}(V_C)| + |\mathcal{O}_s|^2 N_{d_Q}(V_C))$ time.

Mixed Why and Why-not. We next outline an algorithm to answer a set of mixed Why and Why-not questions. The algorithm first group all Why (resp. Why-not) questions to a single Multi-Why (resp. Multi-Why-Not) question. It then invokes MultiFastWhyNot which first relaxes the query Q to intermediate counterpart Q_c to include desired answers for Multi-Why, and then refine Q_c to Q' with MultiApproxWhy to exclude those from Multi-Why-Not.

5.2 "Why-empty" and "Why-so-many"

Why-empty. A Why-empty question [25] is a special case of a Why-not question W =(u_o, V_{C_u}), where $Q(u_o, G)$ = $\emptyset$, and V_{C_u} may be $\emptyset$. A user asks for an operator set that results in a query rewrite Q' that returns at least one entity as answer.

Algorithm FastWhyNot can be extended to answer Why-empty questions as follows. (1) It first initializes V_{C_u} as the candidates of u_o if $V_{C_u} = \emptyset$, *i.e.,* nodes with label $L(u_o)$. (2) It then generates picky operators based on V_{C_u} and iteratively selects operators. (3) It early terminates when the answer closeness $\mathsf{cl}(O)$ reaches a threshold α=$\frac{1}{|V_{C_u}|}$.

Why-so-many. "Why-so-many" (or information-overload problem [25], [26]) for subgraph queries is a case of Why question W=(u_o, V_{N_u}) without explicitly specified V_{N_u}, *i.e.,* the number of results exceeds the user's expectation. An answer to this question is a more specific query rewrite that delivers a bounded number of query answers, by setting a threshold β (β=$\frac{1}{|V_{N_u}|}$ by default). Algorithm ApproxWhy readily supports "Why-so-many" by setting V_{N_u}=$Q(u_o, G)$, and early terminates when answer closeness reaches β.

6 Answering Why-rank Questions

A new challenge of answering Why-rank questions is to identify query manipulation that can re-rank matches according to desired counterparts in I. Algorithms ApproxWhy and FastWhyNot cannot be applied for such cases.

We next introduce an algorithm, denoted as FastWhyRank, to answer Why-rank questions. The algorithm fine-tunes the query towards desired answer rank, by a two-phase "refinement-relaxation" process. In each phase, it (1) dynamically infers specific classes of picky operators that may re-rank the current matches, and (2) greedily selects the operator with the best marginal gain determined by the verified results and I.

Picky Operators. Given the top-k answer $Q_k(u_o, G)$ *w.r.t.* ranking function R and ordered pairs I, an operator o is *picky* if (1) Q'=$Q \oplus o$ is a query rewrite, and (2) there exists at least an ordered pair $\langle v, v' \rangle \in I$, such that either $v' \notin Q'_k(u_o, G)$, or $v' \in Q'_k(u_o, G)$ and $R(Q', v) \geq R(Q', v')$.

Our first result identifies a set of "non-picky" operators that can be safely pruned for a given Why-rank question.

Lemma 6: *Given $Q(u_o, G)$ and a Why-rank question (u_o, I), for any relaxation operator o (*RmE, RxL, *or* RmL*) defined on nodes other than u_o in Q, $v' \in Q'_k(u_o, G)$ and $R(Q', v) < R(Q', v')$ for any pair $(v, v') \in I$.* □

That is, relaxing Q on nodes and edges not involving u_o has no impact on the ranking of any two specific matches in $Q(u_o, G)$. This can be verified as follows. (1) Relaxation does not remove matches, thus v and v' remain to be in $Q(u_o, G)$ (Lemma 1). (2) Any relaxation operator that does not involve u_o has no impact on the ranking scores of any candidate of u_o. Thus $R(Q', v) < R(Q', v')$ for any pair $\langle v, v' \rangle \in I$.

Algorithm. Lemma 6 indicates that it is more effective to take a "*refinement-and-relaxation*" strategy to identify promising query rewrites early. We next outline the algorithm FastWhyRank (illustrated in Figure 4) as follows.

(1) **Refinement Phase**. It invokes a procedure GenRef to generate a picky set $\mathcal{O}_s$ which contains only refinement operators (line 2). FastWhyRank then iteratively selects a refinement operator o^* from $\mathcal{O}_s$ that maximizes a marginal closeness gain (lines 3-5). This process repeats until no picky refinement operator can be applied.

(2) **Relaxation Phase**. If there is available budget for editing and there are pairs in I not re-ranked as desired, it invokes GenRel to generate a set of picky relaxation operators (lines 6-8). It then iterates all relaxation operators and selects the one that can maximize the closeness gain (lines 9-11), until no relaxation can be applied or the editing cost reaches B.

Optimization. To further reduce the cost incurred by verifying the closeness gain in refinement (line 4) and relaxation phase (line 10), FastWhyRank uses two optimization strategies. Denote $Q' = Q \oplus \mathcal{O}$ and $Q'' = Q' \oplus \{o\}$.

Incremental Verification. For AddL and RfL over u_o, it is in polynomial time to check if a match in I is removed from $Q'(u_o, G)$. For other refinement operators (any AddE, and AddL and RfL not involving u_o), we incrementally verify whether each node $v \in Q'_k(u_o, G)$ remains to be a match of Q'', without recomputing $Q''_k(u_o, G)$ from scratch. It early terminates whenever a subgraph isomorphism is found for v, and updates $\mathsf{cl}(\mathcal{O}' \cup \{o\})$ accordingly.

Algorithm FastWhyRank
Input: graph G, query Q, cost bound $B > 0$,
a Why-rank question W=(u_o, L);
Output: the optimal operator set $\mathcal{O}'$ that answers W.
1. set $\mathcal{O}' := \varnothing$; Q′ := Q
2. $\mathcal{O}_s := \mathsf{GenRef}(Q, G, B)$;
*/*greedy selection of refinement operators*/*
3. **while** $\mathcal{O}_s \neq \varnothing$ **and** $c(\mathcal{O}') \leq B$ **do**
4. $o^* := \arg\max\{\mathsf{cl}(O' \cup \{o\}) - \mathsf{cl}(O') : o \in \mathcal{O}_s\}$;
5. $\mathcal{O}' := \mathcal{O}' \cup \{o^*\}$; $\mathcal{O}'_s := \mathcal{O}'_s \setminus \{o^*\}$; $Q' := Q' \oplus \{o^*\}$
6. **if** $\mathsf{cl}(\mathcal{O}', I) = 1$ **or** $c(\mathcal{O}') \geq B$ **do**
7. **return** $\mathcal{O}'$;
8. $\mathcal{O}_s := \mathsf{GenRel}(Q, G, B)$;
*/*greedy selection of relaxation operators*/*
9. **while** $\mathcal{O}_s \neq \varnothing$ **and** $c(\mathcal{O}') \leq B$ **do**
10. $o^* := \arg\max\{\mathsf{cl}(O' \cup \{o\}) - \mathsf{cl}(O') : o \in \mathcal{O}_s\}$;
11. $\mathcal{O}' := \mathcal{O}' \cup \{o^*\}$; $\mathcal{O}'_s := \mathcal{O}'_s \setminus \{o^*\}$; $Q' := Q' \oplus \{o^*\}$
12. **return** $\mathcal{O}'$;

Fig. 4: Algorithm FastWhyRank

Early Stopping. Verifying the marginal gain of each operator o may require $2 \times |I|$ subgraph isomorphism tests to verify if v and v' remain in $Q'_k(u_o, G)$ for each pair $\langle v, v' \rangle \in I$. FastWhyRank further reduces unnecessary verification in both refinement and relaxation phases as follows.

Lemma 7: *Given query Q and operator o, for any $\langle v, v' \rangle \in I$,*
(1) If o is an AddE, RfL *or* AddL *not involving u_o in Q, and either (a) $R(Q, v) \geq R(Q, v')$, $v' \in Q_k(u_o, G)$, or (b) $R(Q, v) < R(Q, v')$, $v \notin Q_k(u_o, G)$, $v' \in Q_k(u_o, G)$, then verifying $v' \in Q'_k(u_o, G)$ is unnecessary;*
(2) If o is a RmE, RxL, *or* RmL *involving u_o in Q, verifying $v' \in Q'_k(u_o, G)$ (resp. $v \in Q'_k(u_o, G)$) is unnecessary if $R(Q', v) \geq R(Q', v')$ and $v' \notin Q_k(u_o, G)$ (resp. $R(Q', v) < R(Q', v')$ and $v \notin Q_k(u_o, G)$, $v' \in Q_k(u_o, G)$).* □

Proof: For refinement operators in (1), if $R(Q, v) \geq R(Q, v')$, one only needs to verify whether v is excluded from $Q'_k(u_o, G)$, as o has no impact on the ranking scores. This reduces the unnecessary verification for testing whether $v' \in Q'_k(u_o, G)$. Similarly, Lemma 7 (2) follows. □

Time cost. Let $N_d(I)$ denotes the d-hop neighbors of the nodes involved in I. GenPicky takes $|Q||N_{d_Q+1}(I)|$ time to generate the picky sets for $N_{d_Q+1}(I)$. The iterative selection takes $|\mathcal{O}_s|^2|N_{d_Q+1}(I)|$ time. FastWhyRank is thus in total $O(|Q||N_{d_Q+1}(I)| + |\mathcal{O}_s|^2|N_{d_Q+1}(I)|)$ time.

Our main results are summarized in Table 2.

7 Experiment

Using real-world graphs, we experimentally verify the effectiveness and efficiency of our algorithms.

Experiment Setting. We used the following setting.

Datasets. We use the following real-world graphs.
(1) *Knowledge bases*. These include (a) *DBpedia*[1], which consists of 4.86M entities, 15M edges, 676 labels (*e.g.,* Person, Building), and on average 9 attributes per node; (b) *Yago*[2], with 1.54M nodes and 2.37M edges (sparser compared to

1. http://dbpedia.org
2. http://www.mpi-inf.mpg.de/yago

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2020.3046436, IEEE Transactions on Knowledge and Data Engineering

Problem	Hardness	Algorithm	Time cost
Why	NP-hard, APX-hard	Approximation: $\mathsf{cl}(Q') \geq \frac{1}{2}\cdot(1-\frac{1}{e})\cdot\mathsf{cl}(OPT)$ - $6B\epsilon$.	$O(\lvert Q\rvert\lvert N_d(Q(u_o,G))\rvert + \lvert\mathcal{O}_s\rvert\lvert N_d(V_{N_u})\rvert^{\lvert Q\rvert} + \lvert\mathcal{O}_s\rvert^2\lvert N_d(V_{N_u})\rvert)$
Why-not	NP-hard	Heuristic	$O(\lvert Q\rvert\lvert N_d(V_{C_u})\rvert + (\lvert\mathcal{O}_s\rvert)^2\lvert N_d(V_{C_u})\rvert)$
Why-rank	NP-hard	Heuristic	$O(\lvert Q\rvert\lvert N_d(I)\rvert + \lvert\mathcal{O}_s\rvert^2\lvert N_d(I)\rvert)$

TABLE 2: Main Results: Hardness and Algorithms

DBpedia), but with more diversified (324K) labels and on average 5 attributes per node, and (c) *Freebase* (version 14-04-14)[3], with 40.32M entities and on average 8 attributes per node, 63.2M relationships, and 9630 labels.

(2) *Social recommendation* networks include (a) *Pokec*[4] a social network with 1.6M users, 30.6M edges and 60 attributes per node; and (b) *IMDb*[5], with 1.7M nodes (*e.g.*, movies), 5.2M edges, and on average 6 attributes per node.

(3) *E-commerce*. (a) We adopted *BSBM*[6] e-commerce benchmark to generate synthetic products graphs with tunable size (up to $50M$ nodes, $126M$ edges and 3080 labels). (b) We perform use case study on a large real online retailer data[7].

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 generates summaries [27] as query templates; (2) for each template, it randomly selects a query node 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 initial $Q(u_o, G)$. For the real online retailer data, we constructed subgraph queries from the search log.

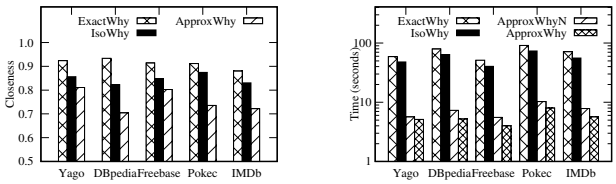
To generate Why-questions, we randomly select a set of nodes in $Q(u_o, G)$ as V_{N_u}. For Why-not questions, we select V_{C_u} with the same type as u_o. For Why-rank questions, we randomly select node pairs $\langle v_i, v_j\rangle$ which satisfy $R(Q, v_i) < R(Q, v_j)$. We remove all conflict node pairs in I.

By default, we set Q with 4 edges, 2 literals per node, both $|V_{C_u}|$ and $|V_{N_u}|$ as 3, editing budget $B = 4$, $k = 10$, $\delta = 25\%$ for guard conditions and I with 4 ordered pairs.

Algorithms. We implemented the following in Java. (1) For Why-questions, we compare the approximation ApproxWhy with (a) its exact counterpart ExactWhy [9], which enumerates and verifies all possible operators within budget B, and (b) a counterpart ApproxWhyN [9] with no pruning strategy (Section 4.1), and (c) 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, its variant FastWhyNotN proposed in [9] without pruning strategy (Section 4.2), and its variant IsoWhyNot, which applies exact Match. (3) For Multi-Why questions, we extend ExactWhy, ApproxWhy, and IsoWhy to MultiExactWhy, MultiApproxWhy, and MultiIsoWhy, respectively, following the strategy introduced in Section 5. (4) For Why-rank questions, we compare FastWhyRank with the exact algorithm ExactWhyRank, and its variant FastWhyRankN without the early stopping strategy (Section 6).

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.

3. http://freebase-easy.cs.uni-freiburg.de/dump/
4. https://snap.stanford.edu/data/soc-pokec.html
5. https://www.imdb.com/interfaces/
6. http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/
7. www.JD.com



(a) Answer Closeness (b) Efficiency

Fig. 5: Answering Why questions

Experimental results. We next present our findings.

Exp-1: Performance on Why. We first evaluate the performance of ExactWhy, ApproxWhy, ApproxWhyN and IsoWhy, for answering Why questions.

Effectiveness. We report the absolute values of answer closeness achieved by the algorithms over all real-world datasets. We calibrate "ground truth" optimal cases with closeness ≈ 1 for pragmatic referenced comparison. Figure 5(a) verifies the following. (1) ApproxWhy preserves the quality guarantees of ApproxWhyN [9]. It reports query rewrites with closeness at least 85% to their optimal counterpart, in all cases (thus only ApproxWhy is shown). (2) While ExactWhy reports optimal answers with exact closeness value by Match (standard subgraph isomorphism), EstMatch is quite accurate on estimating the answer closeness ($\epsilon \leq 0.02$ on average). These verify the effectiveness of ApproxWhy.

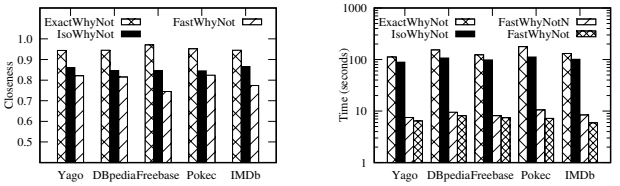
Efficiency. Using the same setting, Figure 5(b) verifies the following. (1) The pruning techniques effectively improves the efficiency of ApproxWhyN by 23.2%. (2) ApproxWhy is feasible on large graphs. It takes on average 5.6 seconds to achieve near optimal results, and outperforms ApproxWhyN, ExactWhy and IsoWhy, by 1.3, 12.7, and 10.2 times, respectively. These suggest the practical application of ApproxWhy in interactive graph exploration.

Exp-2: Performance on Why-not. We next report the performance of ExactWhyNot, FastWhyNot and IsoWhyNot.

Effectiveness. Figure 6(a) reports the closeness of query rewrites achieved by ExactWhyNot and its counterparts. The closeness of query rewrites reported by ExactWhyNot is referred as optimal results, which is on average > 0.95. Under a small budget B = 4, FastWhyNot can achieve a quite good result, with answer closeness at least 84% of the optimal counterparts, for all the cases.

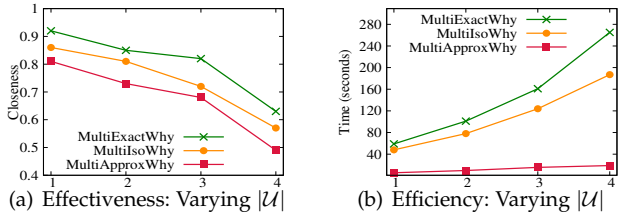
Efficiency. Figure 6(b) shows that it takes on average 2 minutes, 100 seconds and 7 seconds for ExactWhyNot, IsoWhyNot and FastWhyNot to answer a Why-not question with 3 missing matches under budget B=4. FastWhyNot is 19.5 times and 14.6 times faster than ExactWhyNot and IsoWhyNot, respectively, and computes query rewrites with good answer closeness. FastWhyNot improves the efficiency by 19.8% compared with FastWhyNotN. This verifies the effectiveness of the pruning techniques.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2020.3046436, IEEE Transactions on Knowledge and Data Engineering



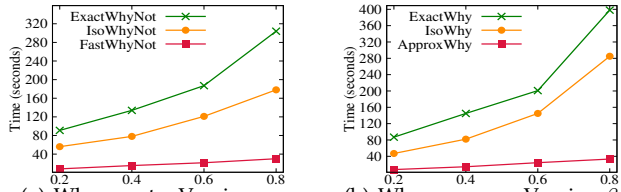
(a) Average Closeness (b) Efficiency

Fig. 6: Answering Why-not questions



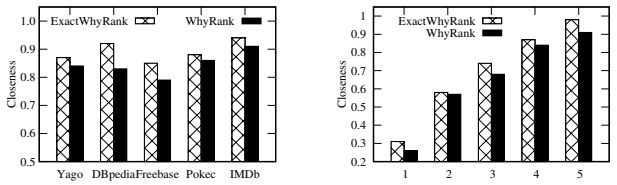
(a) Effectiveness: Varying $|\mathcal{U}|$ (b) Efficiency: Varying $|\mathcal{U}|$

Fig. 7: Answering Multiple Why questions (Yago)



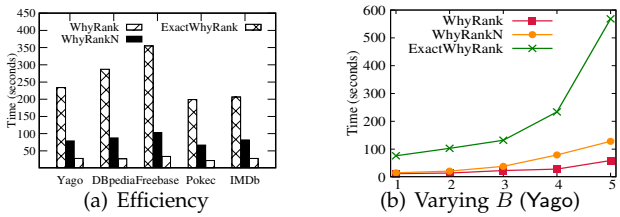
(a) Why-empty: Varying α (b) Why-so-many: Varying β

Fig. 8: Why-so-many and Why-empty: Efficiency (Yago)

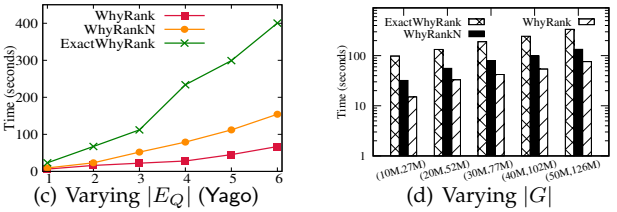


(a) Effectiveness (b) Varying B (Yago)

Fig. 9: Answering Why-rank: Effectiveness



(a) Efficiency (b) Varying B (Yago)



(c) Varying $|E_Q|$ (Yago) (d) Varying $|G|$

Fig. 10: Answering Why-rank questions: Efficiency

Impact of factors. The impact of varying the complexity of queries (size, topology), editing budget, size of Why-questions (*e.g.*, number of missing or unexpected entities), and graph size to the algorithms with pruning strategies is consistent with the results in [9]. We thus omit the results.

Exp-3: Answering Variants of Why-questions.

Answering Multi-Why-questions. We next evaluate the impact of the number of output nodes $|\mathcal{U}|$ to the performance of MultiExactWhy, MultiApproxWhy and MultiIsoWhy. Figure 7(a) shows that the answer closeness achieved by all the algorithms decreases for larger $|\mathcal{U}|$. Indeed, it is more difficult to make a single query rewrite to simultaneously satisfy answer closeness requirement posed on multiple output nodes. On the other hand, MultiApproxWhy is able to obtain query rewrites with answer closeness at least 84% compared with MultiExactWhy.

As shown in Figure 7(b), ApproxWhy takes 19 (resp. 5.7) seconds with 4 (resp. 1) output nodes. All algorithms take more time when the number of output nodes increases. The reason is that multiple output nodes carry more unexpected results, thus the algorithm induces a larger picky set.

The performance of the algorithms answering Multi-Why-not questions are consistent with algorithms answering Multi-Why algorithms. We omit the results here.

Answering Why-empty and Why-so-many. Figure 8 shows the efficiency result of answering Why-empty (resp. Why-so-many) questions by varying the termination condition α (resp. β). The results show that it takes more time for all algorithms over larger α (resp. β) as the algorithms need to include more nodes as new answers (resp. exclude more nodes from the answer set) in order to achieve higher answer closeness. Similar to the results in answering Why and Why-not questions, FastWhyNot and ApproxWhy are the least sensitive ones.

Exp-4: Performance on Why-Rank. We next evaluate the performance of FastWhyRank and ExactWhyRank on answering Why-rank questions, and the impact of varying budget B, question size $|I|$ (the number of node pairs to be "re-ranked"), query (literal) size and data size $|G|$.

Effectiveness. We calibrate the optimal cases by ensuring the existence of "ground truth" queries with desired ranked matches. Figure 9(a) reports the answer closeness achieved by FastWhyRank and ExactWhyRank within budget B (set as 5). FastWhyRank computes near optimal query rewrites that achieve on average 95% of its optimal counterpart reported by ExactWhyRank.

Impact of factors: effectiveness. As shown in Figure 9(b) (over Yago), both FastWhyRank and ExactWhyRank identify queries with more matches to be properly reranked, when more operators are allowed (B is larger). We also observed that the closeness often converges at small B (typically at 5). This suggests that Why-rank questions can be answered by posing a small modification to Q in practice.

Efficiency. Figure 10(a) reports the efficiency of FastWhyRank and its counterparts. It is quite feasible to answer Why-Rank questions for large graphs. For example, FastWhyRank takes 28 seconds over *IMDb* with size (1.7M, 5.2M) to compute query rewrites with answer closeness at least 95%.FastWhyRank outperforms FastWhyRankN by 3 times on average, due to the early stopping strategy.

Impact of factors: efficiency. Figure 10(b) tells us that all algorithms take longer time given larger budget. FastWhyRank is on average 8.16 (resp. 2) times faster than ExactWhyRank (resp. FastWhyRankN). Figure 10(c) verifies that while all the algorithms take longer time for larger queries, FastWhyRank is the least sensitive due to the pruning.

Scalability. We verify the efficiency using larger benchmark graphs *BSBM*, by varying $|G|$ from $(10M, 27M)$

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2020.3046436, IEEE Transactions on Knowledge and Data Engineering

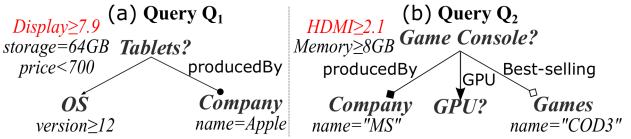


Fig. 11: Real-world Why-questions

to $(50M, 126M)$. Figure 10(d) (in log scale) verifies that FastWhyRank scales well with $|G|$, and is less sensitive compared with FastWhyRankN and ExactWhyRank. It improves FastWhyRankN and ExactWhyRank better over larger $|G|$, and is 1.9 and 4.8 times faster than FastWhyRankN and ExactWhyRank over graphs of size $(50M, 126M)$.

Exp-5: Case Analysis. We also verify the application of our techniques in real online e-commerce recommendation scenarios from our industry partner. We show two cases.

Why-rank. A user aims to find new tablet models (OS version >12) produced by Apple with 64-GB storage, and price less than \$700. Q_1, as shown in Figure 11, returns {iPad Mini 5, iPad Air 3}. The author asks a follow-up Why-rank questions, wishing to find why Mini 5 is ranked higher. FastWhyRank revises Q_1 by an AddL on the output node. The revised query returns {iPad Air 3, iPad Mini 5} and suggests that the user may interest more on a larger display.

Multi-Why. A query Q_2 aims to find a Game Console along with its GPU produced by Microsoft and have a best selling game as "Call of Duty 3". While {(Xbox one,Durango),(Xbox one S, Edmonton),(Xbox one X, Scorpio)} are returned, the user asks why return {(Xbox one,Durango)} given it is an old version. MultiApproxWhy rewrites Q_2 to Q_2' by an AddL on Game Console. This operator suggests that the user may interest more on a more advanced console, *i.e.,* has a higher HDMI version, as verified by follow up search queries.

8 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 [14], [15], [17]–[19]. There are typically two methods: (1) Data editing 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 [14] (for Why questions) or introduce new ones [15], and update queries accordingly [19]. Our work approaches query manipulation to enable exploratory graph search, which requires a better understanding on the impact of operators to explain unexpected and missing answers. While data editing is orthogonal to our work, our algorithms readily extend to track the modified fraction of query answers and responsible operators to suggest relevant editing of graph data.

Skyline operators are used to rewrite queries to deal with user feedback which contains both unexpected and desired answers [28]. Instead of asking users for a complete set of feedback tuples, a pointing domination theory based framework is proposed to find "implicit" user feedback [29]. Such techniques have been studied for (reversed) top-k [18] and reverse skyline queries [17], spatial keyword queries [30] and metric probabilistic range queries [31].

These works cannot be directly applied for subgraph queries over general attributed graphs. To make use of conventional Why-provenance for relational queries, one may express subgraph queries with literals with relational queries [32], [33]. This alone may rely on enhanced algebra [33] or nontrivial encoding [32], and can be expensive for large graphs. with additional transformation cost; Moreover, we develop effective pruning techniques that exploit topological information and query locality to reduce verification cost, which is not readily applicable for relational queries.

Why-questions for graphs. "Why-empty" and "Why-so-many" have been studied to support graph search with SPARQL [34], keywords [35] and pattern queries [25], [26], [36]. (1) For 'Why-empty', SPARQL queries are decomposed into basic operators [34]. Operators that lead to empty answers are identified, and relaxed to include more entities of interests. (2) To explain missing entities for keyword queries in XML data, "You May Also Like" queries are extracted using an A* algorithm [35]. (3) Query patterns are reformulated into supergraphs with maximized diversity [26]. Maximal common subgraphs between queries and data graphs are computed to answer "too many" or "too few" questions [25].

Closer to our work is answering Why-not queries in graph databases [36] with different query semantics. Given a graph database, a query 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 different from searching subgraphs in graph databases. (2) Our methods support a rich set of structural and semantic editing operators, not limited to edge insertion and deletion [36]. 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 [37], [38]. (1) Instead of frequent subgraphs [37], 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 [38] is developed for relational queries only, and does not apply for general subgraph queries. (3) Querying by example [39] finding similar answers to specified examples rather than computing queries that identify them. Why-questions cannot be directly answered by these methods.

9 Conclusions

We have formalized the problem of explaining Why-questions for subgraph queries. We have shown that these problems are in general intractable. We have developed feasible algorithms, from exact and approximations to fast heuristics, with properties such as relative approximation ratio, and early termination. As verified by our experimental study, our methods are efficient and report useful explanations. One future topic is to support multiple Why and Why-not questions with *e.g.,* multi-query optimization and implicit user feedback. Another topic is to explore data editing models for answering Why-questions.

REFERENCES

[1] G. Kasneci, F. M. Suchanek, G. Ifrim, M. Ramanath, and G. Weikum, "Naga: Searching and ranking knowledge," in *IEEE 24th Int. Conf. on Data Eng.*, 2008, pp. 953–962.

[2] M. H. Namaki, Y. Wu, Q. Song, P. Lin, and T. Ge, "Discovering graph temporal association rules," in *ACM Conf. on Info. and Knowl. Management*, 2017, pp. 1697–1706.

[3] W. Fan, X. Wang, and Y. Wu, "Querying big graphs within bounded resources," in *Proceedings of the 2014 ACM SIGMOD Int. Conf. on Management of data*, 2014, pp. 301–312.

[4] G. Zheng, S. Mukherjee, X. L. Dong, and F. Li, "Opentag: Open attribute value extraction from product profiles," in *Proceedings of the 24th ACM SIGKDD Int. Conf. on Knowledge Discovery & Data Mining*, 2018, pp. 1049–1058.

[5] P. Wang, X. Mi, X. Liao, X. Wang, K. Yuan, F. Qian, and R. A. Beyah, "Game of missuggestions: Semantic analysis of search-autocomplete manipulations." in *NDSS*, 2018.

[6] R. E. Kranton and D. F. Minehart, "A theory of buyer-seller networks," *American economic review*, vol. 91, no. 3, 2001.

[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 *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 2018, pp. 927–942.

[8] M. H. Namaki, Y. Wu, and X. Zhang, "Gexp: Cost-aware graph exploration with keywords," in *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 2018, pp. 1729–1732.

[9] Q. Song, M. H. Namaki, and Y. Wu, "Answering why-questions for subgraph queries in multi-attributed graphs," in *IEEE Int. Conf. on Data Eng.*, 2019.

[10] W. Fan, X. Wang, and Y. Wu, "Diversified top-k graph pattern matching," *Proc. VLDB Endowment*, vol. 6, no. 13, pp. 1510–1521, 2013.

[11] L. Zou, J. Mo, L. Chen, M. T. Özsu, and D. Zhao, "gstore: answering sparql queries via subgraph matching," *Proc. VLDB Endowment*, vol. 4, no. 8, pp. 482–493, 2011.

[12] I. F. Ilyas, G. Beskales, and M. A. Soliman, "A survey of top-k query processing techniques in relational database systems," *ACM Computing Surveys (CSUR)*, vol. 40, no. 4, p. 11, 2008.

[13] W. H. Gomaa and A. A. Fahmy, "A survey of text similarity approaches," *Int. J. of Computer Applications*, vol. 68, no. 13, pp. 13–18, 2013.

[14] P. Buneman, S. Khanna, and T. Wang-Chiew, "Why and where: A characterization of data provenance," in *Int. conf. on database theory*, 2001, pp. 316–330.

[15] A. Chapman and H. Jagadish, "Why not?" in *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 2009, pp. 523–534.

[16] Z. He and E. Lo, "Answering why-not questions on top-k queries," *IEEE Trans. on Knowledge and Data Eng.*, vol. 26, no. 6, pp. 1300–1315, 2014.

[17] M. S. Islam, R. Zhou, and C. Liu, "On answering why-not questions in reverse skyline queries," in *IEEE 29th Int. Conf. on Data Eng.*, 2013, pp. 973–984.

[18] Q. Liu, Y. Gao, G. Chen, B. Zheng, and L. Zhou, "Answering why-not and why questions on reverse top-k queries," *The VLDB J.*, vol. 25, no. 6, pp. 867–892, 2016.

[19] Q. T. Tran and C.-Y. Chan, "How to conquer why-not questions," in *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 2010, pp. 15–26.

[20] M. H. Namaki, Q. Song, Y. Wu, and S. Yang, "Answering why-questions by exemplars in attributed graphs," in *Proceedings of the Int. Conf. on Management of Data*, 2019, pp. 1481–1498.

[21] G. Zhu and C. A. Iglesias, "Computing semantic similarity of concepts in knowledge graphs," *IEEE Trans. on Knowledge and Data Eng.*, vol. 29, no. 1, pp. 72–85, 2017.

[22] A. Krause and C. Guestrin, *A note on the budgeted maximization of submodular functions*. Carnegie Mellon University., 2005.

[23] Z. Svitkina and L. Fleischer, "Submodular approximation: Sampling-based algorithms and lower bounds," *SIAM J. on Computing*, vol. 40, no. 6, pp. 1715–1737, 2011.

[24] A. Bonifati, W. Martens, and T. Timm, "An analytical study of large sparql query logs," *The VLDB Journal*, pp. 1–25, 2017.

[25] E. Vasilyeva, M. Thiele, C. Bornhövd, and W. Lehner, "Answering "why empty?" and "why so many?" queries in graph databases," *J of Computer and System Sciences*, vol. 82, no. 1, pp. 3–22, 2016.

[26] D. Mottin, F. Bonchi, and F. Gullo, "Graph query reformulation with diversity," in *Proc. 21th ACM SIGKDD Int. Conf. on Knowl. Discovery and Data Mining*, 2015, pp. 825–834.

[27] Q. Song, Y. Wu, P. Lin, L. X. Dong, and H. Sun, "Mining summaries for knowledge graph search," *IEEE Trans. on Knowledge and Data Eng.*, vol. 30, no. 10, pp. 1887–1900, 2018.

[28] M. S. Islam, C. Liu, and R. Zhou, "Flexiq: A flexible interactive querying framework by exploiting the skyline operator," *J. of Systems and Software*, vol. 97, pp. 97–117, 2014.

[29] ——, "A framework for query refinement with user feedback," *J. of Systems and Software*, vol. 86, no. 6, pp. 1580–1595, 2013.

[30] B. Zheng, K. Zheng, C. S. Jensen, N. Q. V. Hung, H. Su, G. Li, and X. Zhou, "Answering why-not group spatial keyword queries," *IEEE Trans. on Know. and Data Eng.*, vol. 32, no. 1, pp. 26–39, 2018.

[31] L. Chen, Y. Gao, K. Wang, C. S. Jensen, and G. Chen, "Answering why-not questions on metric probabilistic range queries," in *IEEE 32nd Int. Conf. on Data Eng.*, 2016, pp. 767–778.

[32] J. Cortadella and G. A. Valiente Feruglio, "A relational view of subgraph isomorphism," 1999.

[33] G. Szárnyas, J. Marton, J. Maginecz, and D. Varró, "Reducing property graph queries to relational algebra for incremental view maintenance," *arXiv preprint arXiv:1806.07344*, 2018.

[34] M. Wang, J. Liu, B. Wei, S. Yao, H. Zeng, and L. Shi, "Answering why-not questions on sparql queries," *Knowledge and Information Systems*, pp. 1–40, 2018.

[35] M. Naseriparsa, C. Liu, M. S. Islam, and R. Zhou, "Xplorerank: exploring xml data via you may also like queries," *World Wide Web*, vol. 22, no. 4, pp. 1727–1750, 2019.

[36] M. S. Islam, C. Liu, and J. Li, "Efficient answering of why-not questions in similar graph matching," *IEEE Trans. on Knowledge and Data Eng.*, vol. 27, no. 10, pp. 2672–2686, 2015.

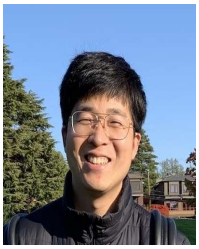
[37] P. Yi, B. Choi, S. S. Bhowmick, and J. Xu, "Autog: a visual query autocompletion framework for graph databases," *The VLDB J.*, vol. 26, no. 3, pp. 347–372, 2017.

[38] Y. Y. Weiss and S. Cohen, "Reverse engineering spj-queries from examples," in *Proc. of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, 2017, pp. 151–166.

[39] N. Jayaram, A. Khan, C. Li, X. Yan, and R. Elmasri, "Querying knowledge graphs by example entity tuples," *IEEE Trans. on Knowledge and Data Eng.*, vol. 27, no. 10, pp. 2797–2811, 2015.

Qi Song is currently working toward the PhD degree in computer science at Washington State University. He received his BS and MS degrees in computer science from Beihang university in China. His research interests include distributed graph data mining and neural network based machine learning for graphs.

Mohammad Hossein Namaki is currently working toward the PhD degree in computer science at Washington State University. He received his BS and MS degrees in computer science from Amirkabir University of Technology in Iran. His research interests are in the broad fields of database systems and data mining with an emphasis on graph databases.



Peng Lin is a Ph.D. student at Washington State University. He received BS and MS degrees in Automation from Shanghai Jiao Tong University in China. His current research interests are knowledge base management and data quality in graphs.

Yinghui Wu is an assistant professor at Case School of Engineering, Case Western Reserve University, US. His current research focus on data systems for knowledge management and exploration, with applications in social and information networks and security. He receives his PhD from the University of Edinburgh in 2011.