1 Explaining Why-not Questions

1.1 Problem Statement

In this paper, we work with RDF graphs which is a collection of subject-property-object(SPO) triples. Assume there are infinite sets I(IRIs), B(blank nodes) and L(RDF literals), then each triple t formed as $(s, p, o) \in (I \cup B) \times I \times (I \cup B \cup L)$ is either a pair of entities with a named relationship or an entity with named attribute and value. For user queries, we mainly consider the basic select-project-join(SPJ) fragment of sparql queries where the selection condition is a conjunction of triple patterns $P_1 \wedge \cdots \wedge P_l$. Each P_i is either a selection triple pattern $(S_j\ op\ c)$ or a join triple pattern $(S_j\ op\ S_k)$, where S_i is a resource standing for either an entity or a class, c is a literal, and op is a predicate.

Why-not Questions. Given an input sparql query Q on a RDF graph G, let $Q(G)$ denote the result set of Q on G. In the most basic form, a why-not question on $Q(G)$ is represented by a non-empty collection of why-not keywords $K = \{k_1, , k_n\}, n \geq 1$, where each why-not keyword k_i can be mapped on G with an entity e_i and $e_i \notin Q(G)$. Corresponding to the why-not keywords, why-not instances is denoted as a non-empty set of instances $I = \{e_1, , e_n\}$.

Essentially, the why-not question is asking why I is not a subset of $Q(G)$; i.e., why each $e_i \in I$ is not in $Q(G)$. In the most general case, a user sparql query may contain multiple variables, thus a why-not question may be posed towards different variables which might cause ambiguity. By default, the select clause of sparql query limits the returned result and why-not questions are associated with the selected variables naturally. For simplicity and without loss of generality, we assume each why-not keyword responds to a specific variable within the user sparql query. Therefore, a why-not question on $Q(G)$ is represented by $WN = \{(k_1, v_1), \cdots, (k_n, v_n)\}, n \geq 1$ where each why-not pair (k_i, v_i) is a why-not keyword k_i and its related variable v_i. The default circumstance can be easily processed as the responding variable v_i is set to $null$. In Example 1, the why-not question is represented by $WN = \{(), ()\}$.

Query Pattern. Given a sparql query Q, the main component of Q is the where clause which is a conjunction of triple patterns. Assume there is a set of variables V disjoint from the sets I, B and L, then each triple pattern is a triple $(v_1, v_2, v_3) \in (I \cup V) \times (I \cup V) \times (I \cup V \cup L)$. A query pattern is a set of triple patterns where the same variable in different patterns denotes a join condition. As a triple pattern can be viewed as a directed edge, a query pattern can be represented as a tree structure. In Example 1, the query pattern of user sparql query can be depicted as a tree in Fig2.

Why-not Query. Given a sparql query Q, a why-not query Q^* is a refined query which is derived from Q with some modification operations. Since the modification operations may be various, we try to construct the why-not query with graph matching methods and preserve the most similar query pattern with Q. In essence, why-not query is a similar query pattern of Q which can returns results containing the why-not instances.

Explanation. Given a sparql query Q and why-not questions WN, the explanation for each why-not question is represented by $E = \{(tq_1, tw_1), \cdots, (tq_m, tw_m)\}, m \geq 1$, where (tq_i, tw_i) is a RDF term pair, tq_i is a RDF term in the user sparql query, tw_i is a RDF term in the why-not query, and $tq_i \neq tw_i$. The explanation answers the why-not question by given a suggestion of modifying the RDF term tq_i to a new RDF term tw_i.

Given the above statements, we formalize the problem of answering why-not questions to sparql queries on RDF databases as follows: Given an input sparql query Q, a RDF graph G, and why-not questions WN, the process of seeking why-not answers is to compute a reasonable explanation E. According to the explanation, users can either understand why her interested item is not in the result set or try to change the original query to gain the expected instances.

1.2 Framework and Algorithms

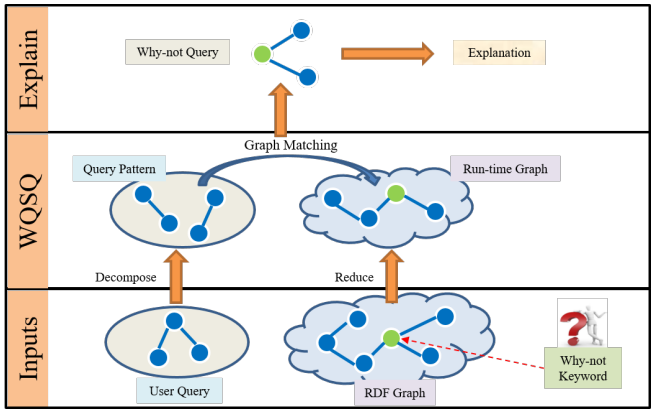

Figure 3: Framework of WQSQ

In this section, we propose a unified framework called $WQSQ$ (i.e., **W**hy-not **Q**uestions on **S**parql **Q**ueries) and present an overview of our approach to compute explanations for why-not questions by heuristically generating a refined why-not query. As illustrated in Figure 3, $WQSQ$ takes a sparql query, a RDF graph and why-not questions as input, and returns the refined sparql query with maximum similarity to users. Specifically, the explanation framework mainly consists of the following three steps: (1) Mapping why-not keyword on RDF graph. (2) Constructing why-not query. (3) Computing explanations. Details of the framework will be discussed subsequently.

In the first step, we need to find the correct instance the user informed. The most simple way is to search the keyword on the RDF graph and returns the instance whose certain attribute value is the keyword. As the search method is executed as an exact matching process, this method always return empty or irrelevant results. For another aspect, if we use a similarity-based method to obtain all relevant instances whose value is similar with the keyword, then too many results would be returned. As a compromise, we use the latter approach to get some instances with a similarity threshold τ limited. After that, the syntax information of edges in query pattern is exploited to disambiguate the instance. Consequently, only a few instances are mapped from the why-not keyword, and we can utilize the similarity score of final constructed why-not graph to further differentiate them. What

should be noticed is that, if no instances are mapped with the why-not keyword, then the framework finished at this step because it's meaningless to explain an absent instance in the RDF graph.

Secondly, the system adopts a graph pattern matching method to construct a why-not query, which is outlined in Algorithm 1. Before the matching, we decompose the query to several graph patterns(Line 1). The advantage of the pattern decomposing is that the whole matching procedure can be divided into series of iterations, where each iteration corresponds to a subgraph pattern matching. Hence the median result can be utilized to guide the matching direction. Several techniques concerning graph pattern decomposing has been proposed, such as edge based, star based, multi-way based, twin-twig based approach. In this paper, we adopt the edge based approach for simplicity and efficiency. The matching process is started from the why-not instance, so that we can guarantee the final constructed why-not query can return the user expected item. In the first iteration, we execute the match between all edges linked with why-not instance and all query patterns. For each matching process, we compute the similarity score between two edges. After the iteration, the edge and query pattern with the highest similarity score is preserved. Next for each iteration(lines 5-11), we find the matched edge to join into the graph generated in the former iteration(line 9). At last, a tree structure similar with user query rooted at the why-not instance is constructed. With the corresponding nodes been replaced with different variables(line 16), a why-not query is generated.

Algorithm 1 Construct a why-not query.

Require:
Run-time Graph, G_R
User Sparql Query, Q
Why-not Instances of a certain keyword, $WI = \{wi_1, wi_2, \cdots, wi_m\}$
Ensure:
Ensemble of classifiers on the current batch, E_n;
1: Decompose Q into a collection of edges pattern $P = \{p_0, p_1, \cdots, p_l\}$;
2: Initial similar graph collection $G_s \leftarrow \emptyset$;
3: **for** each why-not instance $wi_j \in WI$ **do**
4: $G_{wi_j} = (N_j, M_j)$; $N_j = \{wi_j\}$, $M_j = \emptyset$; $score = 0$;
5: **for** each pattern $p_i \in P$ **do**
6: **for** each edge e in linked edges of unmatched node v in N_j **do**
7: $score+ = sim(e, p_i)$;
8: **if** $p_i.match(e)$ **then**
9: $M_j.join(e)$; $N_j.add(e.neighbor(v))$;
10: **end if**
11: **end for**
12: **end for**
13: $G_s.push(G_{wi_j})$;
14: **end for**
15: $sort(G_s)$; $Q_w = G_s.top1$;
16: replace the why-not instance in Q_w with a variable $?x$;
17: **return** Q_w;

After the why-not query is generated, we can return this refined query to the users as an answer to the why-not questions. However, the explanation is a little vague because of the coarsness. Thus we refined our work to compute a more clear and fine-grained explanation. The key idea is that we make a comparison to find the difference between constructed why-not query and user query. And each different place picked out as a term pair consists of the final explanation. The detailed process is illustrated in Algorithm 2.

Algorithm 2 Compute explanations.

Require:
User Sparql Query, Q
Why-not Query, Q_w
Ensure:
Explanation Set of a why-not question, E;
1: **for** each triple $t_w \in Q_w$ **do**
2: find the matched triple t in Q;
3: store the label pair $(t_w.s, t.s)$ $(t_w.p, t.p)$ $(t_w.o, t.o)$ in a map;
4: **end for**
5: **for** each label pair (l, r) in the map **do**
6: **if** l and r are not variables && $l \neq r$ **then**
7: $E.add(pair(l, r))$
8: **end if**
9: **end for**
10: **return** E;

1.3 Score Function

In Algorithm 1, for each why-not instance, we construct a why-not query that can generate results containing the instance. Also we need the constructed query to be the most similar one to the user query. We use this idea as heuristic information and add it to the score function. For each iteration, the score of currently matched sub-graph is defined as follows:

$$score(M_i) = score(M_{i-1}) + sim(t, p_i), (1-1)$$

where,

$$sim(t, p) = sim(t_s, p_s) + sim(t_p, p_p) + sim(t_o, p_o),$$

means the similarity between an edge t and sub pattern p_i .

Theorem 1.1 *Given the score function, the problem of construct a why-not query can be reduced to a dynamic programming problem.*

Proof Sketch: To prove this theorem, we need to verify whether the two key attributes as a dynamic programming problem must have in order applicable to this problem, which is optimal substructure and overlapping sub-problems, respectively. We will state that from two aspects: (1) Optimal Substructure. Suppose the size of decomposed patterns is t, and after i times matching, the constructed subgraph is Gs_i with currently maximum score, $score(M_i)$. In next iteration, we explore the solution space to find an edge that is the most similar with query pattern p_i, which makes the second addend of formula (1-1) maximized, then

$score(M_{i+1})$ is maximized, the matched edge is selected and joined to Gs_i to generate Gs_{i+1}. Hence, the solution to a given optimization problem can be obtained by the combination of optimal solutions to its sub-problems. (2) Overlapping Sub-problems. The process of constructing the why-not query can be decomposed a series of subroutines, we find a matched edge of a given query pattern and join into currently constructed subgraph. Suppose the final why-not graph is $Gs = e_1 \bowtie e_2 \bowtie \cdots \bowtie e_n$, then for the ith iteration, its sub-problems contains randomly joined $(i-1)$ edges. That is, for different iterations, the sub-problems may be overlapping. Based on (1) and (2), the problem of constructing a why-not query satisfies the quality of optimal substructure and overlapping sub-problems, thus can be reduced to a dynamic programming problem, Theorem 3.1 holds.

From theorem 3.1, it is easy to deduce that with the well-defined score function, we can always find the most optimal solution in global. This guarantees that our algorithms can work in a correct and reasonable way.

2 Optimization Strategies

2.1 Run-time Graph Construction

Instead of loading the whole RDF graph G into the main memory, we propose a method to load in a subgraph of G without loss of useful information, which is called the run-time graph G_R regarding Q. Considering the why-not question answering procedure, the necessary knowledge mainly covers the following two types of instance :

- Type-1 instance: which is mapped on the RDF graph by the why not keyword in the why-not question ,
- Type-2 instance: which is covered by types in the user query.

Besides, the relation between the two types instance is also useful as we want to construct a query whose result covers the original instance and why-not instance as much as possible. Therefore, we extract the two kinds of instance at first, then add the linked path between type-1 instance and type-2 instance as auxiliary information. In a further step, for each type-2 instance, we only consider the linked edges labeled the same as the predicates in query pattern. Thus we use the edge information to remove useless relations and attributes of the type-2 instances. The detailed process is presented in Algorithm 3.

2.2 Order-aware Graph Pattern Matching

Definition: (Operator $\prec$) For any two decomposed edge patterns p_i and p_j in Q, $p_i \prec p_j$ if and only if one of the three conditions holds:

- $d(p_i) < d(p_j)$,
- $d(p_i) = d(p_j)$ and $|\sigma_G(p_i)| < |\sigma_G(p_j)|$,
- $d(p_i) = d(p_j)$ and $|\sigma_G(p_i)| = |\sigma_G(p_j)|$ and $id(p_i) < id(p_j)$.

where $d(p_i)$ is the depth of p_i in the query tree, $|\sigma_G(p_i)|$ is the frequency of occurrences of pattern p_i in the RDF graph, and $id(p_i)$ is the unique identifier of pattern p_i assigned with an integer.

Algorithm 3 Construct a run-time graph.

Require:
Global RDF Graph, G
User Sparql Query, Q
Why-not keywords, $K = \{k_1, k_2, \cdots, k_n\}$
Ensure:
Run-time Graph, G_R;
1: $QI \leftarrow \emptyset$; $WI \leftarrow \emptyset$; $G_R \leftarrow \emptyset$;;
2: **for** each variable v_i of Q **do**
3: query instances $QI = \rho_G(v_i)$;
4: **end for**
5: construct query instances graph G_QI with QI;
6: **for** each keyword $k_i \in K$ **do**
7: $G_{R_i} = (N_i, M_i)$;
8: why-not instances $WI_{k_i} = \rho_G(k_i)$;
9: disambiguate why-not instances with edges in Q;
10: **for** each instance $t \in QI$ **do**
11: $N_i.add(t)$; $M_i.add(linkededgesoft)$;
12: compute joint path p between t and remain instances;
13: $N_i.add(p.nodes)$; $M_i.add(p.edges)$;
14: add nodes and edges of G_QI into G_{R_i};
15: **end for**
16: $G_R.push(G_{R_i})$;
17: **end for**
18: **return** G_R;

Theorem 2.1 *Given a pattern decomposition* $\mathcal{D}' = \{p'_0, p'_1, \cdots, p'_t\}$ *of* Q *where each* $p'_i (0 \leq i \leq t)$ *is an edge, let* $\mathcal{D} = \{p_0, p_1, \cdots, p_t\}$ *be the decomposition constructed with order-aware strategy, then* $\mathcal{D}$ *is better than* $\mathcal{D}'$ *to find the best match of* Q.

Proof Sketch:
Intuitively, as the query can be regarded as a tree, it is helpful to match the pattern with shallow and exact information earlier. For the first aspect, the why-not instance is more likely to be the root node or in a top layer, therefore, choose the edge pattern with lower depth in prior is better than the edge pattern with deeper depth. This can be depicted by the first condition. For another part, if two patterns have the same depth, then the pattern with lower frequency of occurrence is better because of the lower space to be selected. And this can be described by the second condition. At last, if the depth and frequency are accidently the same of the two patterns, then its not important to choose which pattern to be matched, thus we just use the pattern id to differentiate the patterns, which is reflected by the third condition. By the above analysis, it is apparently the order-aware pattern matching is better than no orders as the former reduce the unrelated matches in each iteration as much as possible.