知识图谱QA：针对用户的自然语言很难和知识图谱中特定的直接连接上。

利用用户反馈提高性能

An Interactive Mechanism to Improve Question Answering Systems via Feedback

Xinbo Zhang
Peking University
Beijing, China
zhangxinbo@pku.edu.cn

Lei Zou
Peking University
Beijing, China
zoulei@pku.edu.cn

Sen Hu
Peking University
Beijing, China
husen@pku.edu.cn

ABSTRACT

Semantic parsing-based RDF question answering (QA) systems are to interpret users' natural language questions as query graphs and return answers over RDF repository. However, due to the complexity of linking natural phrases with specific RDF items (e.g., entities and predicates), it remains difficult to understand users' question sentences precisely, hence QA systems may not meet users' expectation, offering wrong answers and dismissing some correct answers. In this paper, we design an **I**nteractive **M**echanism aiming for **PRO**motion **V**ia users' f**e**edback to **QA** systems **(IMPROVE-QA)**, a whole framework to not only make existing QA systems return more precise answers based on a few feedbacks over the original answers given by RDF QA systems, but also enhance paraphrasing dictionaries to ensure a continuous-learning capability in improving RDF QA systems. To provide better interactivity and online performance, we design a holistic graph mining algorithm **(HWspan)** to automatically refine the query graph. Extensive experiments on both Freebase and DBpedia confirm the effectiveness and superiority of our approach.

CCS CONCEPTS

• **Information systems** → **Graph-based database models**; **Question answering**; *Database query processing*;

KEYWORDS

knowledge graph; question answering; feedback

ACM Reference Format:
Xinbo Zhang, Lei Zou, and Sen Hu. 2019. An Interactive Mechanism to Improve Question Answering Systems via Feedback. In *The 28th ACM International Conference on Information and Knowledge Management (CIKM'19), November 3–7, 2019, Beijing, China.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3357384.3358059

1 INTRODUCTION

RDF question/answering (QA) has received wide attention due to the fact that these QA systems hide the complexity of RDF and SPARQL behind an intuitive and easy-to-use interface for common users. Generally speaking, there are two kinds of RDF QA solutions. The first one is *semantic parsing-based*, where natural language questions are translated into executable queries such as SPARQL [21] or query graphs[30]; while the other one is *information retrieval-based* [7, 20], which is an end-to-end solution without parsing the question to a formal semantic interpretation. In this paper, we focus on the first category, where a natural language question N is translated into a query graph, and is proved to be more suitable for answering complex questions [12, 25]. Due to the complexity and ambiguity of natural language questions, a fundamental challenge is how to understand a user's question precisely, i.e., interpreting the question N as a proper query Q. A commonly used approach is to employ paraphrasing dictionaries and map natural language phrases to corresponding segments in knowledge base [2, 30]. However, due to mistakes, noise and incompleteness of dictionaries, phrases might be linked to incorrect semantic items, which leads to imperfect answers. In existing RDF QA systems, when returned answers $Q(D)$ do not meet the user's expectation, the system can do nothing but leave confusion to her/him.

In this paper, we design an **I**nteractive **M**echanism aiming for **PRO**motion **V**ia f**e**edback to **Q**/**A** systems **(IMPROVE-QA)**, an interactive platform to make existing QA systems return more precise answers. Specifically, when a user does not satisfy the returned answers $Q(D)$, she/he can give feedback about $Q(D)$, including crossing out some wrong answers ($Q^-(D)$), adding missing correct ones ($Q^\Delta(D)$) and marking some existing correct answers ($Q^+(D)$). Note that our system does ***not*** require the user to provide the full list of errors/omissions or mark all correct answers (experiments show 2-3 hints will be just great). IMPROVE-QA learns from original query Q and her/his feedback, generates a more precise query Q' as a new translation for N and returns more precise answers $Q'(D)$. Furthermore, based on the amendment from Q to Q', we conclude the mappings and enhance dictionaries to avoid making similar mistakes, which can fundamentally improve the QA system and ensure itself a continuous-learning capability.

For ease of presentation, we assume feedbacks are accurate in this section. We will discuss how to deal with the real-world scenario and select high quality feedbacks to avoid noises at the end of Section 2. In this work, we concentrate our attention on designing an efficient algorithm to refine the query from Q to Q' and return answers $Q'(D)$ which fit the user's expectation *more closely* based on her/his feedback.

Example 1. *Consider a question N* ***"Which actress was born in countries in Europe?"*** *issued by a big fan of Elizabeth Taylor. Figure 1 shows the answer set $Q(D)$ which includes six answers returned by an RDF QA system over a knowledge graph D. "Elizabeth Taylor" (London, England, 1932) is unexpectedly excluded from the answers. To her more bewilderment, "Marilyn Monroe" (Los Angeles,*

CIKM '19, November 3–7, 2019, Beijing, China

ACM ISBN 978-1-4503-6976-3/19/11…$15.00
https://doi.org/10.1145/3357384.3358059

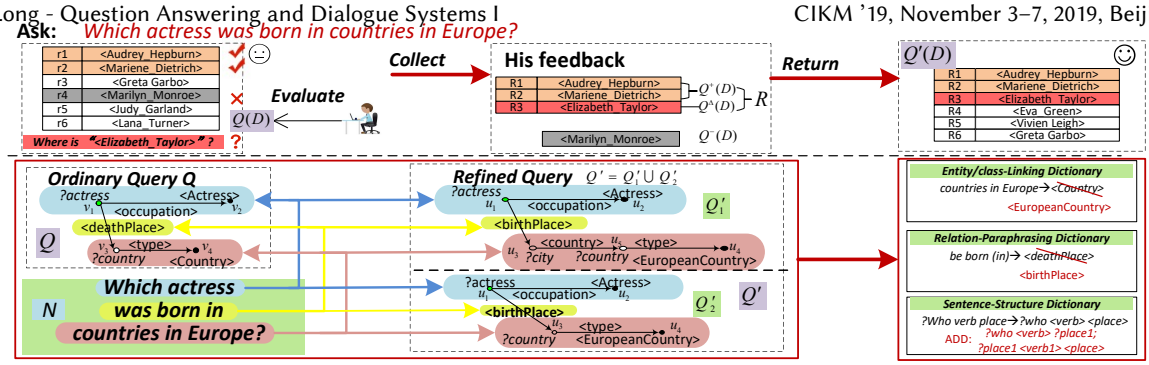

Figure 1: Refine Queries based on the User's Feedback.

America, 1926) is shown in the answer list. What happens to the RDF QA system? Then she gives feedback over $Q(D)$. As shown in Figure 1, she adds one missing correct answer $Q^{\Delta}(D)$ (Elizabeth Taylor), crosses out one wrong answer $Q^{-}(D)$ (Marilyn Monroe) and marks two existing correct answers $Q^{+}(D)$ (Audrey Hepburn and Mariene Dietrich). Let $\mathcal{R} = Q^{\Delta}(D) \cup Q^{+}(D)$ denote all positive answers. Based on her feedback, IMPROVE-QA finds a new query Q' which contains two BGP[1] queries Q'_1 and Q'_2, i.e., $Q' = Q'_1$ UNION Q'_2, each of which is denoted as a query graph. In such cases, it is impossible to generate a single BGP query that covers all positive answers in $\mathcal{R}$ and meanwhile excludes any negative answer in $Q^{-}(D)$. Thus, we allow for multiple BGP queries in the revised query Q'. And, to avoid overfitting, we require the number of BGP queries and the total ***edits*** *from Q to Q' should be minimized. We formally define the problem in Section 3.*

In the above running example, the revised answer set $Q'(D) = Q'_1(D) \cup Q'_2(D)$ satisfies the fan's requirement. Thus, $Q'(D)$ is better than the original $Q(D)$. If she would like to make further revision about $Q'(D)$, IMPROVE-QA can iterate the above process until the returned answers are completely satisfactory. The *edit* from Q to Q' specifies a *graph alignment*. In the lower half of Figure 1, we highlight these alignments, based on which, IMPROVE-QA learns a "lesson" from this query to benefit answering other questions.

Example 2. *Continuing with the previous example. The vertex label $L(v_4)$="⟨ Country ⟩" is relabeled as $L(u_4)$="⟨ EuropeanCountry ⟩" in both Q'_1 and Q'_2. Actually, this is an* ***entity/class linking error*** *in Q. In the original query, "countries in Europe" in natural language question N is linked to ⟨ Country ⟩ but the correct one should be ⟨ EuropeanCountry ⟩. We record "countries in Europe"→ ⟨ EuropeanCountry ⟩ into the* ***entity/class-linking dictionary*** *for further querying. The edge label $L(\overrightarrow{v_1v_3})$="⟨ deathplace ⟩" in Q is relabeled as $L(\overrightarrow{u_1u_3})$="⟨ birthplace ⟩" in Q'_1 and Q'_2. This is a* ***relation paraphrasing error*** *which maps phrase "(be) born in" to "⟨deathplace⟩". Based on this, we can correct the error in the* ***relation-paraphrasing dictionary*** *[30]. The edge $\overrightarrow{v_3v_4}$ in Q is extended to a path $(\overrightarrow{u_3u_5}, \overrightarrow{u_5u_4})$ of length-2 in Q'_1. So we can add a template pattern "?Who verb place?"→ "(?who ⟨ verb ⟩ ?place1. ?place1 ⟨ verb1 ⟩ ⟨ place ⟩)." into* ***sentence-structure dictionary****, which can also benefit the translation for QA systems [19].*

The bottom right corner of Figure 1 shows amendments of the three mentioned dictionaries. Obviously, these amendments can be used to avoid similar errors in answering the following questions, such as *"Give me all European countries."* (*"European countries"* should be mapped to ⟨ EuropeanCountry ⟩ rather than ⟨ Country ⟩), *"Which NBA player was born in USA?"* (*"was born in"* should be mapped to ⟨ birthplace ⟩ rather than ⟨ deathplace ⟩) and so on.

Although the refining process seems similar to querying database by examples (QBE) [4, 11, 15], IMPROVE-QA and QBE are based on completely different assumptions and intentions. QBE provides convenient interfaces for users to query [4] or explore [11, 17] knowledge base, while IMPROVE-QA aims to aid RDF QA systems to understand natural language questions better and return more precise answers. For QBE problems, sample answers are the only inputs, while IMPROVE-QA considers not only the seed answers (including positive and negative) given by the user but also the original translated query Q. This is because many translation errors from question N to query graph Q are reflected in some parts of Q, such as entity/class linking (node error), relation paraphrasing (edge error) and sentence structure (structure error). We want to take advantage of existing query Q and refine Q according to the user. Thus, our goal is to find "minimum edit" from original query Q to a new query Q', where $Q'(D)$ provides more precise answers. This is different from the intuition of QBE work. Although considering a very special case when original query Q is null, it seems that IMPROVE-QA devolves into querying database by examples, we further focus on the "lessons" learned from this query and explore reasons why QA systems cannot accomplish the translation task. We also enhance dictionaries and improve RDF QA systems on *subsequent* questions. All of these issues are not covered by existing QBE work. In a nutshell, we make the following contributions:

(1) We propose IMPROVE-QA, a whole framework for existing RDF QA systems, to not only correct errors during translation from N to Q, return more precise answers based on Q and a few feedbacks over original answers, but also enhance paraphrasing dictionaries to improve QA systems for answering other questions.

(2) To refine the query graph based on original query Q and the user's feedback, we propose an efficient holistic graph mining algorithm *HWspan*, to improve online performance and effectiveness.

(3) We evaluate IMPROVE-QA in two real RDF graphs and corresponding QA benchmark datasets. Extensive experiments confirm the effectiveness and superiority of our method.

2 OVERVIEW

Feedback Collection: When RDF QA system returns answer set $Q(D)$ to a user, this component shows an interface to allow feedback. Based on the user's feedback, this component collects some positive answers $\mathcal{R} = Q^{\Delta}(D) \cup Q^{+}(D)$ and negative answers $Q^{-}(D)$.

Query Refinement: This is the core component of IMPROVE-QA. According to the original query graph Q (given by RDF QA system) and the user's feedback ($\mathcal{R}$ and $Q^{-}(D)$, collected in the previous

[1]basic graph pattern; formally defined in https://www.w3.org/2001/sw/DataAccess/rq23/sparql-defns.html

component), this component aims to find a refined query Q', where $Q'(D)$ covers all positive answers in $\mathcal{R}$ and excludes any negative one in $Q^-(D)$. More importantly, we design an efficient algorithm (HWspan) to find the minimum *edits* from Q to Q'.

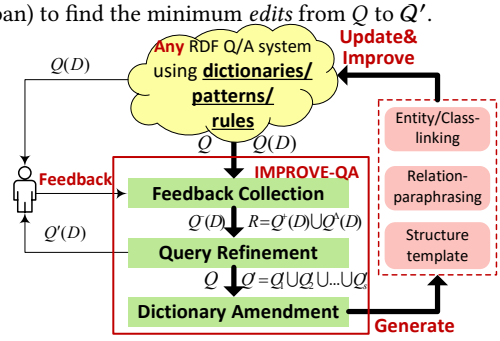

Figure 2: Framework.

Dictionary Amendment: The refined query based on a user's feedback for question N is applied for answering *subsequent* questions in this component. Generally, we utilize the alignment between the original query Q and the refined query Q'.

Above components can be iterative. If a user is still unsatisfied with refined answers $Q'(D)$, she/he can provide more feedbacks over new answers and IMPROVE-QA iterates the above process.

Feedback Quality Issue: There are two different contexts in our IMPROVE-QA. One is for the individual user. IMPROVE-QA will return refined answers $Q'(D)$ which fit the user's expectation *more closely* based on her/his feedback, no matter if the user's feedbacks are correct or not; just like "garbage in, garbage out". The second context is to improve the quality of paraphrasing dictionaries (i.e., "dictionary amendment"). This can be done in the offline phase. For a question, our system adopts a voting mechanism to select high quality feedbacks based on a *group* of users' feedbacks. For example, we only consider the positive and negative feedback answers (i.e., $Q^+(D)$ and $Q^-(D)$) that are annotated by most users. More sophisticated quality control techniques [3] in crowdsourcing area can also be used, but that is beyond the scope of this work. In other words, only "good" feedbacks are fed into "dictionary amendment" component. Due to the space limit, we concentrate ourselves on designing an efficient "query refinement" algorithm in this paper.

3 QUERY REFINEMENT

Definition 1. ***Query Refinement.*** *Given a SPARQL query graph Q over RDF graph D and the user's specified (partial) positive answer set $\mathcal{R}$ and (partial) negative answer set $Q^-(D)$, where $\mathcal{R} = Q^+(D) \cup Q^\Delta(D)$ and $Q^-(D) \cap \mathcal{R} = \emptyset$. Our goal is to find a refined SPARQL query Q', where $Q' = Q'_1 \cup Q'_2 \cup \ldots \cup Q'_s, s \geq 1$, and each of Q'_i is a BGP query[2]. Furthermore, the following conditions hold:*

(1) $Q'(\mathcal{D}) \supset \mathcal{R}, Q'(\mathcal{D}) \cap Q^-(\mathcal{D}) = \emptyset$, *where $Q'(\mathcal{D})$ denotes the answer set by evaluating refined query Q' over RDF graph D.*
(2) $MIN(\sum_{i=1}^{s} D(Q'_i, Q))$, *where $D(Q'_i, Q)$ denotes the* minimum graph edit distance*(see Definition 5) between Q'_i and Q.*

The first condition in Definition 1 specifies that the refined answer set $Q'(\mathcal{D})$ should satisfy the user's feedback. If we only consider the first condition, the most trivial solution is to construct a revised query Q'_i for each given positive answer meanwhile dismissing the negative answers. Obviously, this is overfitting. The second condition aims to select the revised queries with the minimum total number of edits. Essentially, this also obeys the "Occam's razor", i.e., "More things should not be used than minimum needed."

To address the query refinement problem, we propose a graph mining solution. Evaluating a SPARQL query graph is equal to finding subgraph matches of query graph over RDF graph[30]. Thus, if a vertex v is contained in answer set $\mathcal{R}$, it is easy to know the query graph must be contained in (subgraph isomorphism to) the *neighborhood structure* around v. In contrast, if a vertex v' is a negative answer, the query graph is *not* a subgraph of the *neighborhood structure* around v'. Intuitively, our method is to find a refined query Q' which is contained in neighborhood structures of all positive answers in $\mathcal{R}$ but are excluded in neighborhood structures of any negative answer in $Q^-(D)$. Furthermore, we should minimize the edit distance from the original query Q to the revised query Q'.

Given a vertex v in RDF graph D, we define the neighborhood structure around v as a *K-hop Neighborhood Graph* $G_N(V_N, E_N)$, which is a subgraph induced by all vertices reachable in k-hops from the vertex v. However, k-hop neighborhood graph may be very large even $k = 2$. Consider a natural language question N. It can be observed that there are many irrelevant vertices (entities) and edges (relations) with respect to N in a k-hop neighborhood graph. To shrink the neighborhood structure around v, we propose *Primary Neighborhood Graph* around a vertex v. Intuitively, if a path in k-hop neighborhood graph does not contain any entity or class mentioned in N, this path can be safely pruned.

Let $T = \{t_1, ..., t_n\}$ be all mentioned entity phrases in a natural language question N[3]. According to entity linking results, each entity phrase t_i $(i = 1, ..., n)$ is linked to multiple entities or classes in RDF graph D, denoted as V_{t_i}. Intuitively, the *Primary Neighborhood Graph* around a vertex v prunes all paths (in the k-hop neighborhood graph) without any mentioned entity/class in N.

Definition 2. ***Primary Neighborhood Graph.*** *Given a vertex v and candidate vertex set $\bigcup_{i=1,..,n} V_{t_i}$, the Primary Neighborhood Graph of vertex v is defined as $G_P(V_P, E_P)$, where the following conditions hold:*

(1) $G_P \subset G_N$, $v \in V_P$, *and G_P is a connected graph;*
(2) $\forall v_j \in (V_N \cap V_{t_i}), i = 1, .., n \Rightarrow v_j \in V_P$
(3) $\nexists G'_P \subset G_P$, *where G'_P also satisfies the above two conditions.*

and vertex v is called the ***center*** *of the primary neighborhood graph.*

Figure 3 shows an example of 2-hop neighborhood graph and primary neighborhood graph. The shaded vertices in Figure 3 refer to mentioned entities in the natural language question N. Given a center vertex v, to obtain primary neighborhood graph G_P around v, we prune all k-hop paths without containing any shaded vertices from k-hop neighborhood graph. In practice, we set k=2, since most questions refer to a concentrated subgraph of knowledge graph.

3.1 Baseline: A Two-step Method

3.1.1 ***Pattern Generation.*** Let $G_{P_{\mathcal{R}}}$ and $G_{P_{Q^-(D)}}$ denote all primary neighborhood graphs of positive answers in $\mathcal{R}$ and negative ones in $Q^-(D)$, respectively. We devise to extract frequent

[2]BGP: a basic graph pattern; ∪ corresponds to "UNION" in SPARQL syntax.

[3]Some work[23, 28] discussed how to detect mentioned entity phrases in N.

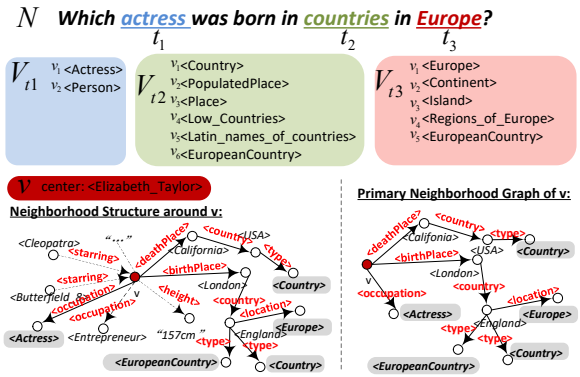

Figure 3: Primary Neighborhood Graph.

patterns in $G_{P_{\mathcal{R}}}$ so that we can deduce the expected query graphs distinguished from the negative answers. To achieve this goal, we first unify the label of each center node v ($L(v)$) in all primary neighborhood graphs (in $G_{P_{\mathcal{R}}}$ and $G_{P_{Q^-(D)}}$) as a wildcard character "?x". Actually, these *centers* finally map to variable vertices in the refined SPARQL queries. We mine frequent subgraph patterns from the primary neighborhood graph set $G_{P_{\mathcal{R}}}$ (for positive answers) using any frequent subgraph mining algorithm (such as gSpan [22]). We only keep frequent patterns containing the center vertex "?x", which are denoted as F_P. A pattern that can MATCH (Definition 3) some positive answers (in $\mathcal{R}$) and does not MATCH *any* negative answer (in $Q^-(D)$) is called a *qualified pattern* (Definition 4). All qualified patterns in F_P are collected to form candidate set PS. We will discuss how to select some qualified patterns (in PS) to MATCH all answer in $\mathcal{R}$ in the "pattern selection" step.

Definition 3. ***MATCH.*** *Given a pattern graph $P = (V_T, E_T)$ with center vertex P_{center} and a node v in graph $G = (V_G, E_G)$, P MATCH v, denoted as $MATCH(P, v)$, if and only if there exists an injective function $f: V_T \to V_G$, where the following conditions hold:*

(1) $f(P_{center}) = v$
(2) $\forall v \in V_T \wedge v \neq P_{center}, L(v) = L(f(v))$
(3) $\forall \overrightarrow{v_1 v_2} \in E_T, \overrightarrow{f(v_1) f(v_2)} \in E_G \wedge L(\overrightarrow{v_1 v_2}) = L(\overrightarrow{f(v_1) f(v_2)})$
where $L(\cdot)$ denotes the label of vertex or edge.

Definition 4. ***Qualified Pattern.*** *A pattern p is a qualified pattern if and only if*

- *p MATCH at least one positive answer in $\mathcal{R}$;*
- *p does not MATCH any negative answer in $Q^-(D)$.*

Example 3. *See Figure 1, there are three pattern graphs: Q (with center vertex v_1 "?actress"), Q'_1 and Q'_2 (with center vertex u_1 "?actress"). Given nodes R_1 "⟨Audrey_Hepburn⟩", R_3 "⟨Elizabeth_Taylor⟩" in the graph shown in Figure 3, then we say Q'_1 can* MATCH *R_3, Q can* MATCH *R_1, but Q'_2 or Q cannot MATCH R_3. Q'_1 is a qualified pattern because Q'_1 can MATCH the positive answer R_3, and does not MATCH any negative answer in $Q^-(D)$.*

3.1.2 ***Pattern Selection.*** We select several patterns from the candidate set PS to obtain a refined query set $\mathcal{Q} = \{Q'_1, ..., Q'_s\}$ to cover all positive answers. To satisfy the second condition of Definition 1, we utilize *Minimum Graph Edit Distance* to quantify the modification from original query Q to $\mathcal{Q}'$.

Definition 5. ***Minimum Graph Edit Distance(MGED)*[4] *with Center.*** *Given a pattern graph $P(V_P, E_P)$ with center vertex P_{center} and an original query graph $Q(V_Q, E_Q)$, the MGED between P and Q is defined under a bijective function*[5] *$f: V_P \to V_Q$ as follows:*

$$D(P, Q) = \min_f \left(\sum_{\substack{v \in V_P \\ v \neq P_{center}}} D(v, f(v)) + \sum_{\substack{v_1, v_2 \in V_P \\ v_1 \neq v_2}} D(\overrightarrow{v_1 v_2}, \overrightarrow{f(v_1) f(v_2)}) \right), \quad (1)$$

$$D(v, f(v)) = \begin{cases} 1 & L(v) \neq L(f(v)) \\ 0 & L(v) = L(f(v)) \end{cases} \quad (2)$$

$$D(\overrightarrow{v_1 v_2}, \overrightarrow{f(v_1) f(v_2)}) = \begin{cases} 1 & L(\overrightarrow{v_1 v_2}) \neq L(\overrightarrow{f(v_1) f(v_2)}) \\ 0 & L(\overrightarrow{v_1 v_2}) = L(\overrightarrow{f(v_1) f(v_2)}) \end{cases}$$

where, (a) vertex labels and edge labels are their corresponding entities and predicates; (b) if there is no edge from v_1 to v_2 in E_P, we define $L(\overrightarrow{v_1 v_2})$ as a special label ϕ. That is the same for defining $L(\overrightarrow{f(v_1) f(v_2)})$ if there is no edge from $f(v_1)$ to $f(v_2)$ in E_Q; (c) if a vertex/edge label is a wildcard (i.e., variable in SPARQL), it can match any vertex/edge label without any cost.

Under this function, the pattern graph P can be transformed to Q by at least $D(P, Q)$ steps using a set of graph edit operators including vertex/edge deletion, insertion and label substitution. For each pattern graph $P_i \in PS$, let $\{v \in \mathcal{R} | MATCH(P_i, v)\}$ denote all answers in $\mathcal{R}$ that are MATCH-ed by pattern P_i. The weight of each graph P_i is defined as the minimum graph edit distance between P_i and the original query Q (i.e., $D(P_i, Q)$). Essentially, the query refinement problem (defined in Definition 1) is to find a set of patterns in PS that MATCH all answers in $\mathcal{R}$, which is a weighted-set-cover problem. Due to its NP-complete hardness, we adopt a classical greedy weighted-set-cover algorithm for pattern selection. Let $\mathcal{R}'$ denotes all answers in $\mathcal{R}$ that are not yet MATCH-ed by chosen patterns, initially, $\mathcal{R}' = \mathcal{R}$. For each step, we select a pattern $P_i \in PS$ with the minimized *weighted coverage* $wc(P_i)$.

$$wc(P_i) = \frac{D(P_i, Q)}{|\{v \in \mathcal{R} | MATCH(P_i, v)\} \cap \mathcal{R}'|} \quad (1)$$

where $MATCH(P_i, v)$ is defined as Definition 3.

Then, we set $\mathcal{R}' = \mathcal{R}' - \{v \in \mathcal{R} | MATCH(P_i, v)\}$, i.e., removing all answers (in $\mathcal{R}'$) that can be MATCH-ed by the selected pattern P_i. The above process is iterated until that $\mathcal{R}' = \phi$, i.e., all answers are MATCH-ed by the selected patterns. The approximation ratio of the greedy algorithm is $O(ln|R| + 1)$.

3.1.3 ***Limitations of Two-step Method.*** Although the *two-step method* is a conceptually simple scheme, it has some issues in practice, especially online performance.

Expensive online processing time. In practice, many frequent patterns mined in the first step are not selected in the pattern selection step, which means that the two-step strategy misses the opportunity to prune unpromising patterns in the first step.

Redundancy searching space. Frequent subgraph mining algorithms always adopt BFS (such as FSG [16]) or DFS (such as gSpan [22]) search strategy to generate frequent subgraph patterns. Obviously, these pattern generation orders do not favor our problem,

[4]Classical definition of MGED refers to the minimum number of primitive operations to transform a graph to another, which is easy to be proved equivalent to Definition 5.
[5]If $|V_P| < |V_Q|$, introduce $|V_Q| - |V_P|$ isolated pseudo vertices in P, or vice versa.

since we hope the promising patterns are generated as early as possible, where promising patterns refer to those with low *weighted coverage wc*. Furthermore, once some of the generated patterns can MATCH all positive answers in $\mathcal{R}$, we can stop the searching, which definitely saves unnecessary searching cost.

Uncertain threshold. Mining frequent subgraph patterns need to specify the minimum support *minSup*. In our problem, *minSup* is not perceptible in advance, since it is essentially decided by the "pattern selection" step. A large threshold may lead to insufficient patterns to MATCH all answers in $\mathcal{R}$, but a small threshold brings a large search space and expensive processing time. Actually, it is impossible for us to set an optimal threshold in advance, which remains a stubborn defect of the two-step method.

Considering the limitations, we propose an efficient heuristic method that combines the above two steps into an integrated phase, which can significantly improve the online processing performance, and also avoid the tedious threhold-setting procedure.

3.2 HWspan: Heuristic Weighted Subgraph Pattern Mining

HWspan is a holistic method that combines pattern generation and pattern selection together. We always select some promising patterns to MATCH answers in $\mathcal{R}$ during the pattern generation process. Once selected patterns can MATCH the whole $\mathcal{R}$, the algorithm stops. As we know, a pattern P (*Child* Pattern) is always generated by extending one of its sub-pattern (called *Parent* Pattern) P' (Definition 6). Traditional frequent pattern mining algorithms always follow DFS or BFS search strategy, but we give priority to "promising" patterns which have small *weighted coverage* (i.e., Equation 1). To enable that, we design a novel *lower bound* (see Theorem 2) to predict the *weighted coverage* of a pattern and its descendant patterns. Obviously, the less *lower bound*, the more priority. More interestingly, the *lower bound* is monotonically increasing during expanding patterns (see Theorem 4), which means that if a pattern P is not "good" enough to be selected, it is not necessary to generate its descendant patterns. Therefore, a best-first search strategy (Algorithm 1) is designed to avoid exploring unnecessary searching space. Furthermore, our HWspan does not require frequency threshold *minSup* as input. Searching stops when the selected patterns can MATCH all answers in $\mathcal{R}$, which does not depend on *minSup*.

Definition 6. ***(Parent Pattern, Child Pattern)*** *Let $P'(V', E')$ and $P(V, E)$ be two patterns. P' is called a* Parent Pattern *of P (or P is a* Child Pattern *of P') if and only if (1) $(E = E' \cup \{e\}) \wedge (e \notin E')$ and (2) $(V = V') \vee (V = V' \cup \{u\})$, where e is a new edge in P whose both endpoints are both in V or one of its endpoints is a new node u in V but not in V'.*

An example of parent and child pattern is given in Figure 4. During pattern expansion, a pattern P is always generated by extending its parent P' by introducing one extra edge e. Note that all primary neighborhood graphs in $G_{P_{\mathcal{R}}}$ are subgraphs of RDF graph D. Each vertex in the primary neighborhood graph has a unique vertex label, which equals to vertex ID in RDF graph D. Based on this feature, we design a method in Section 3.2.3 to avoid generating isomorphic patterns repeatedly. As mentioned in Section 3.1.1, each primary neighborhood graph has a center vertex, whose label is assigned a distinct label "?x". In our problem, we only keep the patterns containing *center vertex*. Therefore, the pattern always grows from a single edge, one of whose endpoints is the center vertex "?x".

3.2.1 ***Intuition of HWspan.*** Recall pattern selection of the two-step method. We always select a pattern P to minimize the *weighted coverage* (Equation 1). While in HWspan, to avoid generating all patterns, we hope our algorithm has the following *monotone* property: if a pattern P' is not selected at some step, it is not necessary to consider any of its descendant patterns P at this step.

Unfortunately, the *weighted coverage* $wc(P_i)$ (see Equation 1) does not hold for the above *monotonicity*. Given a pattern P'_i and one of its child patterns P_i, $wc(P'_i)$ may be larger than $wc(P_i)$. In this case, according to the pattern selection method in Section 3.1.2, we need to select P_i (child pattern) rather than P'_i (parent pattern), which violates the monotonicity. Let us recall the *weighted coverage* in Equation 1, distinctly, the denominator of Equation 1 of P' is not smaller than that of P if P' is a parent pattern of P. Therefore, the reason that *weighted coverage* violates the monotonicity is the *numerator* of Equation 1, i.e., the graph edit distance $D(P_i, Q)$. For example, in Figure 4, $P' \subset P$ but $D(P', Q) > D(P, Q)$.

Essentially, HWspan is a A*-style algorithm, where we design a *lower bound* for $wc(P'_i)$ (Equation 1) that satisfies the monotonicity. The key is to find a monotonic *lower bound* for $D(P_i, Q)$, the *numerator* of Equation 1. Although there are lots of proposals for *lower bound* of graph edit distance [27, 29], they fail to satisfy the monotonicity. To achieve the goal, we propose the following SGED-based *lower bound*. An example is given in Figure 4.

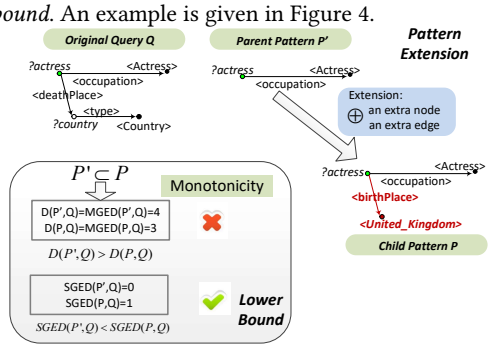

Figure 4: Pattern Extension and Lower Bound.

3.2.2 ***Lower Bound.*** In this subsection, we propose a *lower bound* for pattern P's *weighted coverage*.

Definition 7. ***Sub-Graph Edit Distance (SGED).*** *Given two graphs P and Q, the* sub-graph edit distance *between two graphs P and Q, written as $SGED(P, Q)$, denotes the minimum operations needed to transform from P to any of Q's subgraph Q_j, which can be defined as,*

$$SGED(P, Q) = \min_j \left(D(P, Q_j) \right), Q_j \subseteq Q$$

where $D(P, Q_j)$ denotes the MGED between P and Q_j (Definition 5).

Lemma 1. *Given two graphs P and Q, the following equation holds:*

$$SGED(P, Q) \leq D(P, Q)$$

Proof. Assume that $SGED(P, Q) > D(P, Q)$. According to MGED definition, P needs $D(P, Q)$ operations to transform from graph P to graph Q. Obviously, Q is also a subgraph of Q, i.e., $Q \subseteq Q$. According to SGED definition, pattern P only needs $D(P, Q)$ operations to

a special subgraph of Q (i.e., Q itself). This contradicts to the assumption $SGED(P, Q) > D(P, Q)$. Therefore, the lemma holds. □

Theorem 2. *Given a pattern P_i, the following equation defines a* lower bound *for P_i's* weighted coverage *$wc(P_i)$.*

$$lb(P_i) = \frac{SGED(P_i, Q)}{|\{v \in \mathcal{R} | MATCH(P_i, v)\} \cap \mathcal{R}'|} \leq wc(P_i) \quad (2)$$

Proof. It is easy to be proved based on Lemma 1. □

Lemma 3. *Given two pattern graphs P, P' and a query graph Q, where P is a* child pattern *of P', the following equation holds:*

$$P' \subset P \Rightarrow SGED(P', Q) \leq SGED(P, Q) \quad (3)$$

Proof. According to definition of SGED, let $SGED(P', Q) = D(P', Q_t) = \delta$, and $SGED(P, Q) = D(P, Q_k) = \theta$, where $Q_t \subseteq Q$ and $Q_k \subseteq Q$. Since $P'(V', E')$ and $P(V, E)$ satisfy: $(V = V') \vee (V = V' \cup \{u\})$, $(E = E' \cup \{e\}) \wedge (e \notin E')$. So if we remove edge e and its matching edge in P and Q_k, respectively, we transform P into P' and find a matching between P' and Q'_k, where $Q'_k \subset Q_k$. Assume that in this match P' can turn into Q'_k by Δ operations, that is $\Delta \geq D(P', Q'_k)$. It is easy to derive that $\theta \geq \Delta$, since the the matching between P' with Q'_k is the same with the matching between P with Q_k except for the removed edge e. Since $SGED(P', Q)$ indicates the MGED between P' and any subgraph of Q, that is, $\delta \leq D(P', Q'_k)$. Therefore $\theta \geq \delta$ holds. □

Theorem 4. ***Monotonicity.*** *Given two patterns P and P', where P is a* descendant pattern *of P' (i.e., $P' \subset P$), the following equation holds:*

$$P' \subset P \Rightarrow lb(P') \leq lb(P) \quad (4)$$

where $P' \subset P$ and $lb(P)$ is defined in Equation 2.

Proof. According to lemma 3, we have $SGED(P, Q) \geq SGED(P', Q)$. P' can MATCH no less answers in $\mathcal{R}$ than P because P' has less restriction than P. So P have larger numerator and smaller denominator than ancestor pattern P', so the theorem holds. □

Based on the monotonicity of the *lower bound*, it is easy to deduce the following pruning rule, which will be used in our best-first search algorithm in Section 3.2.4.

Prune 1. *At some moment of pattern selection, there are two patterns P_1 and P_2 to be selected, if $wc(P_2) < lb(P_1)$, it is not necessary to compute the* weighted coverage *of P_1 as well as its descendant patterns at this moment. Furthermore, it is also not necessary to expand P_1's child patterns at this moment.*

3.2.3 ***Pattern Extension.*** We generate patterns in a growing mode. Initially, all single edges with one endpoint labeled "?x" (i.e., the center vertex) in primary neighborhood graphs of $G_{P_\mathcal{R}}$ are ancestor patterns. We always extend the current pattern to its child P by introducing one more edge. To avoid duplicate pattern generation, considering the properties of primary neighborhood graphs in $G_{P_\mathcal{R}}$, we propose a canonical labeling code for each pattern graph (Definition 9). As mentioned earlier, the vertex label of primary neighborhood graphs and pattern graphs are their corresponding vertex IDs in RDF graph D. Thus, each vertex in primary neighborhood graphs and pattern graphs has a unique vertex label. We first define the order between two edges in a pattern graph as follows.

Definition 8. *Given two edges[6] $e_1 = (v_{i_1}, v_{j_1})$, $e_2 = (v_{i_2}, v_{j_2})$ in a pattern graph P, then $e_1 \succ e_2$, if and only if one of the following holds:*

(1) $L(v_{i_1}) > L(v_{i_2})$; *or*
(2) $L(v_{i_1}) = L(v_{i_2}) \wedge L(v_{j_1}) > L(v_{j_2})$; *or*
(3) $L(v_{i_1}) = L(v_{i_2}) \wedge L(v_{j_1}) = L(v_{j_2}) \wedge L(e_1) > L(e_2)$

where $L(\cdot)$ denotes the vertex or edge label.

Since each vertex has a unique label in primary neighborhood graphs and pattern graphs, thus, it is easy to define its canonical labeling code based on "$\succ$" edge order.

Definition 9. ***Canonical Labeling Code.*** *Given a pattern graph P, assuming that P has n edges, its* canonical labeling code *is defined as $(e_1, e_2, ..., e_n)$, where $e_i \succ e_{i+1}$, $i = 1, ..., n-1$.*

Definition 10. ***Frontier Edge.*** *Given a pattern graph $P = (V_P, E_P)$, an edge $e_t \in E_P$ is called the* frontier edge *of P if and only if $\forall e_i \in E_P$, $i \neq t$, such that $e_t \succ e_i$.*

During the pattern extension, when extending one pattern P' to its child pattern P by introducing one edge e, if $e \succ e'$, where e' is the frontier edge of P', we accept this extension and update e as the frontier edge of P, otherwise, we reject that because an isomorphic pattern has been detected yet (i.e. Prune 2).

Prune 2. *When extending a pattern P' with a frontier edge e', if there exists an edge e that can be attached to P' but $e' \succ e$, then e is left out of consideration for extending P'.*

3.2.4 ***Best-First Searching.*** To find the most "promising" patterns with small *weighted coverage*, we design a best-first search strategy (see Algorithm 1). We maintain three priority queues: *span_queue*, *wait_select_queue*, and *select_queue*. Among them, *span_queue* stores all graph patterns that have been mined, which are ranked in the increasing order of $lb(P)$ (see Equation 2), the *lower bound* for pattern P's *weighted coverage*. Initially, all single-edge patterns containing center vertex ?x are pushed into *span_queue* (Line 2 in Algorithm 1). In each step, we always pop the head pattern P' of *span_queue* and expand P' to its child patterns (see Section 3.2.3) and push them back to *span_queue* again (Lines 5-6). Meanwhile, if P' is a *qualified pattern* (see Definition 4), we push P' into another priority queue *wait_select_queue* (Lines 7-8), in which all patterns are ranked in the increasing order of $wc(P')$ (see Equation 1), the *weighted coverage* of pattern P'. Now, let P_1 and P_2 be head patterns in priority queues *span_queue* and *wait_select_queue*, respectively. At some moment, if $lb(P_1) > wc(P_2)$, it means that all patterns in both *span_queue* and *wait_select_queue* (not including P_2) have larger *weighted coverage* than P_2. Furthermore, due to the monotonicity (Theorem 4), all unexplored patterns' *weighted coverage* cannot be lower than $wc(P_2)$. In other words, at this moment, P_2 has the minimized *weighted coverage*. Thus, we pop P_2 from priority queue *wait_select_queue* and push it into *select_queue* (Lines 10-11), where *select_queue* stores all patterns that have been selected to MATCH answers in $\mathcal{R}$. After selecting P_2, we need to update $\mathcal{R}'$ (Line 12), which denotes all answers (in $\mathcal{R}$) that are not yet MATCHed by chosen patterns. Note that updating $\mathcal{R}'$ leads to update the orders of patterns in both *span_queue* and *wait_select_queue* (Lines

[6] Considering the context of RDF and SPARQL, the graphs discussed in this paper are always directed graphs.

13). Finally, when answers in $\mathcal{R}$ are MATCH-ed by all selected patterns (i.e, $\mathcal{R}' = \phi$), these selected patterns form the refined query $Q' = \{Q'_1 \cup ... \cup Q'_s\}$ (Line 14).

The above HWSpan algorithm assumes that there is only a single variable (corresponding to *center* in the pattern graph) in the generated SPARQL. In fact, HWspan algorithm can be generalized to process query graphs with multiple variables. Specifically, when expanding an existing pattern with an edge, we can consider an edge with a node marked by wildcard character (i.e., a variable vertex in SPARQL), which can match any node while judging whether the new pattern can MATCH an answer in $\mathcal{R}$. Others are similar to the above discussion. We omit the details due to space limit.

Algorithm 1: Best-First Search Algorithm

Input: Primary neighborhood graph sets $G_{P_{\mathcal{R}}}$ and $G_{P_{Q^-(D)}}$
Output: The refined query $Q' = \{Q'_1 \cup ... \cup Q'_s\}$

```
1  Let priority queues span_queue, wait_select_queue,
     select_queue be φ.
2  Push all single-edge patterns containing center vertex ?x into priority
     queue span_queue, ranked by the lower bound.
3  Let R' = R
4  while R' ≠ φ do
5      Pop head pattern P' of span_queue
6      Push all child patterns of P' to span_queue (see Section 3.2.3)
7      if P' is a qualified pattern then
8          Push P' into wait_select_queue, ranked by the weighted
             coverage.
9      Let P1 and P2 be head patterns of span_queue and
         wait_select_queue, respectively.
10     if lb(P1) > wc(P2) then
11         Pop P2 from wait_select_queue and push it into
             select_queue.
12         Update R' = R' − {v ∈ R|MATCH(P2, v)}.
13         Update the orders of patterns in both span_queue and
             wait_select_queue.
14 All patterns in select_queue form the refined query
     Q' = {Q'1∪, ..., ∪Q's}.
15 return Q'
```

4 EXPERIMENTS

4.1 Setup

Datasets. We perform following experiments on two real RDF graphs: DBpedia[7] and Freebase[8]. DBpedia is a structured knowledge dataset extracted from Wikipedia. We use DBpedia 2014, which consists of 110 million triples with 5.4 million entities and 9708 predicates. Freebase is a collaboratively edited knowledge base. We use Freebase of version 2013, which has 596 million assertions with 41 million entities and 19456 predicates.

Benchmarks. We use two benchmarks: QALD-6[9] for DBpedia and WebQuestionsSP[26] (WebQSP for short) for Freebase. QALD is a series of open-domain question answering competitions over linked data based on DBpedia. QALD-6 contains 350 training and 100 testing questions, both with gold standard SPARQL query and answers. WebQuestionsSP contains the evaluated value of gathering SPARQL queries and answers for a set of original questions from WebQuestions[6], containing 5810 question-answer pairs with 65% as training data (3778) and 35% as test data (2032).

Entity Linking Tools. We use the method mentioned in [23] to detect entities in natural language questions based on Freebase and use DBpedia Lookup[10] to search for DBpedia entities by related keywords in the natural language questions.

Measure Metrics. For DBpedia, we adopt three metrics: *precision*, *recall*, and *F1 of average precision and recall*. For Freebase, we measure *average F1 of all questions*[11]. We report the number of questions whose answers are the same with gold standard (*see "#Perfect" in Table 1*), others are regarded as "faulty". We measure the average edit distance between Q and Q' to quantize the modification.

QA systems. We reproduced two semantic parsing-based and dictionaries-dependent QA systems: *gAnswer*[12, 30] and *Aqqu*[5], where *gAnswer* is used on DBpedia and *Aqqu* is used on Freebase. Among all 2032 test questions in WebQSP, 750 questions return exact gold answers by *Aqqu*, while other 1282 questions cannot be answered perfectly, i.e, the F1 score of these questions is not 100%, denoted as "faulty". For *gAnswer*, 66 queries of QALD-6 receive perfect answers, while other 34 queries are faulty. We report statistic results on above metrics in Table 1 (*see row "Before"*).

Feedback Collection. We asked 300 college students to join our experiment, and each natural language question N is shown to at least 5 users. If a user is not satisfied with the original answer set $Q(D)$, she/he can give feedbacks, including marking positive answers $\mathcal{R}$ and crossing out negative answers $Q^-(D)$. The user is encouraged to give 1-5 feedbacks considering knowledge limitation. (In Section 4.3.3 we discuss how the number of "hints" affects the effectiveness of IMPROVE-QA.) Online, for each question with each user, IMPROVE-QA returns a better Q' based on Q and her/his feedback to meet her/his expectation. And offline, we adopt a voting machanism to analyze the feedback logs. If at least 3 users give the same feedback on an answer for a specific question, then we regard it as "good feedback". We only summarize the dictionaries (see Section 4.3.5) and statistics based on good feedbacks.

4.2 Results of IMPROVE-QA

Experiment Design. To extensively illustrate the effect of our IMPROVE-QA, we conduct following two experiments: (1) For each QA system, IMPROVE-QA can be regarded as a plug-in, where basic QA system is viewed as a black box performing original whole QA tasks, and the user benefits from the results after applying IMPROVE-QA. To verify this process, Section 4.2.1 simulates to give feedback directly on QALD-6 and WebQSP **test** questions. (2) IMPROVE-QA can also be integrated with the existing QA system into a self-promotion platform, where original QA system acts as a fundamental QA module and IMPROVE-QA essentially improves the kernel supporting dictionaries. To certify that IMPROVE-QA radically augments the QA system a continuous-learning capability, Section 4.2.2 simulates to give feedback on QALD-6 and WebQSP **training** questions, and use the updated dictionaries after learning by IMPROVE-QA to answer **test** questions **without** feedback.

4.2.1 ***Apply IMPROVE-QA Directly on QA.*** In this experiment, we **only** use QALD-6 and WebQSP **test** questions. First, 100

[7] http://blog.dbpedia.org/
[8] https://developers.google.com/freebase/
[9] https://qald.aksw.org/index.php?x=home&q=6
[10] http://wiki.dbpedia.org/projects/dbpedia-lookup
[11] Since most RDF QA systems based on Freebase [5, 6] only report *average F1*, for easy comparision, we use *average F1* for Freebase.

Metrics	*gAnswer*: QALD-6+DBpedia (Total #:100)								*Aqqu*: WebQSP+Freebase (Total #:2032)			
	#Perfect	Recall		Precision		F1		Edits	#Perfect	Average F1		Edits
		Partial	**All**	Partial	**All**	Partial	**All**			Partial	**All**	
Before	66	11.8%	**70.0%**	67.6%	**89.0%**	20.0%	**78.4%**	-	750	19.8%	**49.4%**	-
IMPROVE-QA-ALL	80	66.5%	**93.3%**	76.7%	**95.3%**	71.2%	**94.3%**	1.76(test)	1380	36.7%	**79.7%**	1.43(test)
IMPROVE-QA-DIC	73	45.3%	**85.2%**	72.8%	**92.7%**	55.8%	**88.8%**	2.12(train)	1028	31.8%	**66.3%**	2.23(train)

Table 1: Original and Refined Results of Faulty Queries on WebQSP by *Aqqu* and on QALD by *gAnswer*.

and 2032 test questions are fed into QA systems. For each question, we receive original query Q and corresponding answer set $Q(D)$. Then we imitate the user to give feedback on original answer set $Q(D)$. IMPROVE-QA collects the feedback, finds a refined query Q', and gives new answer set $Q'(D)$.

In Table 1 (*see row "IMPROVE-QA-ALL"*), for 34 and 1282 faulty questions answered by *gAnswer* and *Aqqu* respectively, after applying IMPROVE-QA, we report the results by comparing $Q'(D)$ with gold standard $Q^*(D)$ (*see column "Partial"*). We also do statistics on the whole QALD-6 and WebQSP benchmark (*see column "All"*) to expediently compare with state-of-the-art results. The number of faulty queries decreases and residual faulty queries receive better answers after IMPROVE-QA on both datasets. Overall, IMPROVE-QA can improve F1 from 78.4% to 94.3% for QALD-6 and average F1 from 49.4% to 79.7% for WebQSP. We received refined dictionaries (*see "test" in Table 5*), and details are given in Section 4.3.5.

4.2.2 ***Apply Updated Dictionaries on QA.*** In this experiment, 350 and 3778 **training** questions of QALD-6 and WebQSP can receive feedback and be applied to IMPROVE-QA. We received updated entity/class-linking dictionary, relation-paraphrasing dictionary, and sentence-structure dictionary learning from training data for both QA systems . Then three updated dictionaries are fed into *gAnswer* and *Aqqu* respectively. **Without** additional feedback, *gAnswer* and *Aqqu* give answers on QALD-6 and WebQSP **test** questions respectively based on refined dictionaries.

The last row (*"IMPROVE-QA-DIC"*) in Table 1 gives the results on test questions. Overall, IMPROVE-QA can improve F1 from 78.4% to 88.8% for QALD-6 and average F1 from 49.4% to 66.3% for WebQSP benchmark using the trained dictionaries, which can certify the self-learning ability of IMPROVE-QA. In Section 4.3.5, Table 5 (*see "train"*) shows the quality of dictionaries.

Moreover, for DBpedia dataset, to enable the comparision with all QA systems in QALD-6[12], Table 2 shows their results in the QALD competition report format. For WebQSP benchmark, we show average F1 in Table 3 to compare with previous work. Obviously, both results show the superiority of IMPROVE-QA.

4.3 Detailed analysis

4.3.1 ***Sizes of (Primary) Neighborhood Graphs.*** Table 4(a) shows the average scale of vertices and edges in k-hop neighborhood graph and primary neighborhood graph when $k = 2$. Obviously, it is too large for neighborhood graph to be processed. Primary neighborhood graphs shrink the size greatly. Furthermore, we find that target entities in gold standard SPARQL queries are *not* missed by shrinking, i.e., primary neighborhood graph has the same "gold entities"(entities mentioned in gold standard SPARQL) coverage rate as neighborhood graph, (*see Table 4(a)*). A very small percentage of "gold entities" cannot be covered in both kinds of graphs when k=2, which has negligible influence on the experiments. Such confirms the superiority and effectiveness of our proposed compression technique and chosen parameter for $k(=2)$-*hop*.

System	Recall	Precision	F1
CANaLI	89.0%	89.0%	89.0%
UTQA	69.0%	82.0%	75.0%
KWGAnswer	59.0%	85.0%	70.0%
NbFramework	85.0%	87.0%	54.0%
SemGraphQA	25.0%	70.0%	37.0%
gAnswer	70.0%	89.0%	78.4%
gAnswer-IMPROVE-QA-ALL	93.3%	95.3%	94.3%
gAnswer-IMPROVE-QA-DIC	85.2%	92.7%	88.8%

Table 2: Results for QALD-6 Test Questions.

System	Average F1
Yavuz et al. [24]	52.6%
Sempre [6]	35.7%
STAGG [25]	52.5%
Aqqu [5]	49.4%
Aqqu-IMPROVE-QA-ALL	79.7%
Aqqu-IMPROVE-QA-DIC	66.3%

Table 3: Results for WebQSP Test Questions.

4.3.2 ***Performance.*** We calculate the online time of *two-step method* and *HWspan* in our experiments. Let $minSup = 50\% * \theta_r$, and generally, HWspan is faster than two-step method by around 10 times. For example, the average time for HWspan is 1.67 seconds but two-step method spends 17.38 seconds in DBpedia and similarly in Freebase, as reported in Table 4(b). HWspan generates 15 candidate patterns on average while two-step method generates 56. For the worst case of HWspan with 14.95s execution time, two-step method consumes more than 10 minutes. This is because separate pattern generation and pattern selection phases waste unnecessary time, and existing frequent pattern mining algorithms cause unnecessary cost. Below, we give four specific cases, and Figure 5 shows the online time of two-step method and HWspan, which illustrates that both pattern generation and selection in the two-step method consume much more time than HWspan.

Case1: *Which country was Bill Gates born in?*
Case2: *Which rivers flow into the North Sea?*
Case3: *What kind of music did Lou_Reed play?*
Case4: *Show me all books in Asimov's Foundation series.*

Moreover, the two-step method can not neccessarily return better query and answer set with a predefined *minSup*. When $minSup = 50\%*\theta_r$[13], for QALD-6 test benchmark, we get a lower "precision" after using the two-step method ($recall = 81.3\%$, **precision=80.8%**, $F1 = 81.0\%$). This is because that a lower *minSup* allows more patterns, which brings in extra answers. With the increase of *minSup*, precision will increase and recall will decrease.

[12]https://qald.aksw.org/6/documents/qald-6_results.pdf

[13]Our experiments suggest, $minSup = 50\% * \theta_r$ is the best considering both performance and refined answer set.

(a) Size and coverage rate of Primary Neighborhood Graph(PNG) and Neighborhood Graph(NG).

	NG	PNG
|V|	10^6	10^2
|E|	10^6	10^2
WebQSP	99.06%	99.06%
QALD-6	96.0%	96.0%

(b) Average runtime of two-step method(TS) ($minSup = 50\% * \theta_r$) and HWspan(HW) .

	TS	HW
Freebase	14.4s	1.33s
DBpedia	17.38s	1.67s

Table 4: Graph Size and Online Runtime.

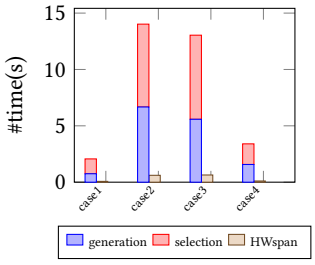

Figure 5: Runtime of Two-Step versus HWspan.($\theta_r = 2$)

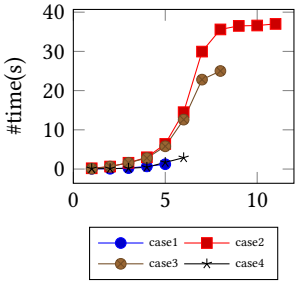

Figure 6: Runtime Rises as θ_r Increases.

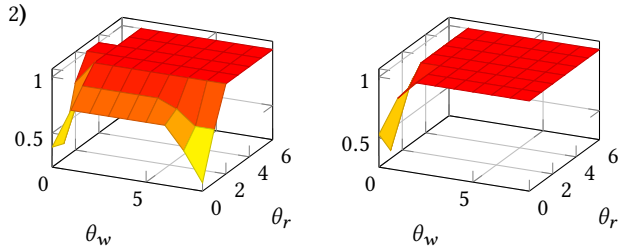

Figure 7: Recall(left) and Precision(right) versus θ_w and θ_r.

4.3.3 ***Cost of Feedback.*** Intuitively, if the user can give more "hints", IMPROVE-QA can achieve a better self-correction ability, but we prove that IMPROVE-QA can learn perfectly when it receives *enough* information instead of *all* evaluation of the answers. To validate our intuition, we make the following experiments. Figure 6 shows that, among all above 4 cases, processing time of HWspan increases sharply as θ_r (number of positive answers) rises (θ_w has few effect on runtime). Take a typical question as an example, *"Which rivers flow into the North Sea?"*, which has 24 answers in gold answer set and the QA system returns 16 answers (7 wrong and 9 right). Experiments show that the precision, recall and F1 are generally first increasing then keeping as the number of hints grows (i.e., increasing θ_w and θ_r), as shown in Figure 7. There are also some intriguing findings: (1)θ_r has a major impact on recall and θ_w mainly impacts on precision. Specifically, when θ_r grows from 1 to 2, recall rises sharply and when θ_w grows from 0 to 1, precision increases obviously. We regard 2 as θ_r's *safe value*, and 1 as θ_w's *safe value*. (2)When θ_r is small and $\theta_w = 0$, *precision* changes instability and vice versa. (3)When θ_r is much larger than θ_w, and θ_w is no larger than its safe value, the result may be overfitting, and vice versa. (4)However, once surpassing the safe value, IMPROVE-QA returns excellent results generally.

Above all the discussion about performance and number of hints, when $\theta_r = 2$ and $\theta_w = 1$, $F1$ value approaches 100%, and online runtime is acceptable for interaction, which verifies that IMPROVE-QA is friendly and efficient for users, only a few hints can improve the answers' quality significantly.

4.3.4 ***SPARQL Assessment.*** We compute graph edit distance between refined query graph Q' and original query Q in Table 1. The average distances are 1.76 and 1.43 for test queries in QALD-6 and WebQSP respectively, 2.12 and 2.23 for training queries respectively, which indicates that IMPROVE-QA not only improves the precision/recall of the returned answer set but also returns a "better" SPARQL query by tiny modifications. An accurate refined SPARQL query can deduce a more accurate entity/relation-mappings.

Dictionaries		QALD-6+DBpedia		WebQSP+Freebase	
		Pairs	Reliability	Pairs	Reliablity
Entity-Mapping	test	26	73.1%	530	68.5%
	train	52	90.4%	678	61.8%
Relation-Paraphrase	test	18	83.3%	614	50.8%
	train	86	95.3%	729	60.1%
Sentence-Structure	test	8	100%	19	94.7%
	train	12	100%	21	90.5%

Table 5: Refined Dictionaries.

4.3.5 ***Refined Dictionaries.*** For DBpedia, based on original queries by *gAnswer* and refined queries by IMPROVE-QA, we extract 26 entity mappings, 18 relation mappings and 8 sentence-structure templates from 100 test questions; 52 entity mappings, 86 relation mappings and 12 sentence-structure templates from 350 training questions, and evaluate their reliability manually. As shown in Table 5, most of them make sense and are accepted into dictionaries. We also do such extraction and evaluation on Freebase. Table 6 gives some typical examples of three dictionaries. All of these extracted mappings/templates can be used into answering subsequent questions, which enables QA systems to continuously learn from users. Experiments in Section 4.2.2 confirm that.

5 RELATED WORK

QA Systems over RDF Knowledge Base. Question answering systems (QA) have attracted lots of attention in both academia and industry. In this paper, we focus on the semantic parsing-based method, which translates natural language questions into logical representation, including λ-DCS [6] and SPARQL[21]. In these methods, one of commonly used approaches is to employ paraphrasing dictionaries and map natural language phrases to corresponding segments in SPARQL queries [2, 30]. One of our goals in this paper is to enhance the paraphrasing dictionaries based on the user's feedback, as shown in Section 4.3.5 and Table 5, 6.

Why-not Problem. WHY-NOT was first proposed by [9] in the database community. When issuing a SQL query, the user may not satisfy the returned answers. In this case, there are three typical approaches to address WHY-NOT problems. (1)*Giving reasons on why/why-not,* [8, 10]. Recent work [1] gives a visilized interpretation of complete derivation sequence from the natural language utterance to the final answer; (2) *Modifying the underlying database,* [13]; and (3) *Modifying queries.* This one is quite related to our problem, but most of them consider relational database. For example, [18] proposes algorithms to generate refined query which can explain both select-project-join queries and select-project-join queries with aggregation. The recent work [14] considers the why-not problem in the context of graph database, but they consider

(a) **Example of refined entity/class-linking dictionary.**

Question	Who is the son of Sonny and Cher?
Original Query	**SELECT DISTINCT ?x WHERE** { ?x ⟨parent⟩ ⟨Sonny_Cher⟩. }
Refined Query	**SELECT DISTINCT ?x WHERE** { ?x ⟨parent⟩ ⟨Cher⟩. ?x ⟨parent⟩ ⟨Sonny_Bono⟩. }
Change of Dict	Add: "Sonny"→ ⟨Sonny_Bono⟩
Remark	Error in node recognition

(b) **Example of refined relation-paraphrasing dictionary.**

Question	Which rivers flow into the North Sea?
Original Query	**SELECT DISTINCT** ?uri **WHERE** { ?uri ⟨type⟩ ⟨River⟩ . ⟨North_Sea⟩ ⟨inflow⟩ ?uri . }
Refined Query	**SELECT DISTINCT** ?uri **WHERE** { ?uri ⟨type⟩ ⟨River⟩ . ?uri ⟨riverMouth⟩ ⟨North_Sea⟩ . }
Change of Dict	Add: "flow into"→⟨riverMouth⟩
Remark	Lack of target mapping

(c) **Example of refined sentence-structure dictionary.**

Question	In what city is the Heineken brewery ?
Original Query	**SELECT DISTINCT** ?x **WHERE** { ⟨Heineken⟩ ⟨manufacturer⟩ ?x . ?x ⟨type⟩ ⟨City⟩ . }
Refined Query	**SELECT DISTINCT** ?x **WHERE** { { ⟨Heineken⟩ ⟨manufacturer⟩ ?x . ?x ⟨type⟩ ⟨City⟩ .} **UNION** {⟨Heineken⟩ ⟨manufacturer⟩ ?uri . ?uri ⟨locationCity⟩ ?x . } }
Change of Dict	Add template pattern: "In ?place verb sth?"→ "(sth ⟨verb⟩ ?place1. ?place1 ⟨verb1⟩ ?place.)"
Remark	Lack of pattern leads to incomplete sentence structure.

Table 6: Examples of Refined Dictionaries.

why-not query over a collection of small data graphs, which is different from a single large RDF graph database.

Query by Examples. Querying by examples (QBE) lies in helping non-professional users access knowledge database by examples instead of SPARQL queries [4, 11, 15]. The major difference between QBE and our problem is that our goal is to maximize the similarity between the refined query Q' and Q, but QBE does not consider the original query graph Q. The only inputs to QBE are sample answers, which are different from our problem while we take both the original query graph Q as well as sample answers as inputs.

6 CONCLUSION

In this paper, we propose an **I**nteractive **M**echanism aiming for **PRO**motion **V**ia f**e**edback to **QA** systems **(IMPROVE-QA)**. To improve the performance, we design an efficient *HWspan* algorithm that combines the pattern generation and pattern selection together. The amendment of query is used to enhance QA system's translation dictionaries for subsequent queries which augments the QA system a continuous-learning capability from the user's feedback.

ACKNOWLEDGMENTS

This work was supported by The National Key Research and Development Program of China under grant 2018YFB1003504 and NSFC under grant 61932001, 61961130390, 61622201 and 61532010. Lei Zou is the corresponding author of this paper.

REFERENCES

[1] Abdalghani Abujabal, Rishiraj Saha Roy, Mohamed Yahya, and Gerhard Weikum. 2017. QUINT: Interpretable Question Answering over Knowledge Bases. In *EMNLP Demo*. 61–66.
[2] Abdalghani Abujabal, Mohamed Yahya, Mirek Riedewald, and Gerhard Weikum. 2017. Automated Template Generation for Question Answering over Knowledge Graphs. In *WWW*. 1191–1200.
[3] M. Allahbakhsh, B. Benatallah, A. Ignjatovic, H. R. Motahari-Nezhad, E. Bertino, and S. Dustdar. 2013. Quality Control in Crowdsourcing Systems: Issues and Directions. *IEEE Internet Computing* 17, 2 (2013), 76–81.
[4] Marcelo Arenas, Gonzalo I. Diaz, and Egor V. Kostylev. 2016. Reverse Engineering SPARQL Queries. In *WWW*. 239–249.
[5] Hannah Bast and Elmar Haussmann. 2015. More Accurate Question Answering on Freebase. In *CIKM*. 1431–1440.
[6] Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. 2013. Semantic Parsing on Freebase from Question-Answer Pairs. In *EMNLP*. 1533–1544.
[7] Antoine Bordes, Sumit Chopra, and Jason Weston. 2014. Question Answering with Subgraph Embeddings. *Computer Science* (2014).
[8] Peter Buneman, Sanjeev Khanna, and Wang Chiew Tan. 2001. Why and Where: A Characterization of Data Provenance. In *ICDT*. 316–330.
[9] Adriane Chapman and H. V Jagadish. 2009. Why not? *SIGMOD* (2009), 523–534.
[10] Y. Cui and J. Widom. 2003. Lineage tracing for general data warehouse transformations. *The VLDB Journal* 12, 1 (2003), 41–58.
[11] Jialong Han, Kai Zheng, Aixin Sun, Shuo Shang, and Ji-Rong Wen. 2016. Discovering Neighborhood Pattern Queries by Sample Answers in Knowledge Base. In *ICDE*. 1014–1025.
[12] Sen Hu, Lei Zou, and Xinbo Zhang. 2018. A State-transition Framework to Answer Complex Questions over Knowledge Base. In *EMNLP*. 2098–2108.
[13] Jiansheng Huang, Ting Chen, AnHai Doan, and Jeffrey F. Naughton. 2008. On the provenance of non-answers to queries over extracted data. *PVLDB* 1, 1 (2008), 736–747.
[14] Md. Saiful Islam, Chengfei Liu, and Jianxin Li. 2015. Efficient Answering of Why-Not Questions in Similar Graph Matching. *IEEE TKDE* 27, 10 (2015), 2672–2686.
[15] Nandish Jayaram, Arijit Khan, Chengkai Li, Xifeng Yan, and Ramez Elmasri. 2015. Querying Knowledge Graphs by Example Entity Tuples. *IEEE TKDE* 27, 10 (2015), 2797–2811.
[16] Michihiro Kuramochi and George Karypis. 2001. Frequent Subgraph Discovery. In *ICDM*. 313–320.
[17] Matteo Lissandrini, Davide Mottin, Yannis Velegrakis, and Themis Palpanas. 2018. X 2 Q: your personal example-based graph explorer. *PVLDB* 11, 12 (2018), 2026–2029.
[18] Quoc Trung Tran and Chee Yong Chan. 2010. How to ConQueR why-not questions. *SIGMOD* (2010), 15–26.
[19] Christina Unger, Lorenz Bühmann, Jens Lehmann, Axel-Cyrille Ngonga Ngomo, Daniel Gerber, and Philipp Cimiano. 2012. Template-based question answering over RDF data. In *WWW*. 639–648.
[20] Kun Xu, Siva Reddy, Yansong Feng, Songfang Huang, and Dongyan Zhao. 2016. Question Answering on Freebase via Relation Extraction and Textual Evidence. In *ACL*.
[21] Mohamed Yahya, Klaus Berberich, Shady Elbassuoni, Maya Ramanath, Volker Tresp, and Gerhard Weikum. 2012. Natural language questions for the web of data. In *EMNLP*. 379–390.
[22] Xifeng Yan and Jiawei Han. 2002. gSpan: Graph-Based Substructure Pattern Mining. In *ICDM*. 721–724.
[23] Yi Yang and Ming-Wei Chang. 2016. S-MART: Novel Tree-based Structured Learning Algorithms Applied to Tweet Entity Linking. *CoRR* (2016).
[24] Semih Yavuz, Izzeddin Gur, Yu Su, Mudhakar Srivatsa, and Xifeng Yan. 2016. Improving Semantic Parsing via Answer Type Inference. In *EMNLP*. 149–159.
[25] Wen Tau Yih, Ming Wei Chang, Xiaodong He, and Jianfeng Gao. 2015. Semantic Parsing via Staged Query Graph Generation: Question Answering with Knowledge Base. In *ACL-IJCNLP*. 1321–1331.
[26] Wen-tau Yih, Matthew Richardson, Christopher Meek, Ming-Wei Chang, and Jina Suh. 2016. The Value of Semantic Parse Labeling for Knowledge Base Question Answering. In *ACL*.
[27] Zhiping Zeng, Anthony K. H. Tung, Jianyong Wang, Jianhua Feng, and Lizhu Zhou. 2009. Comparing Stars: On Approximating Graph Edit Distance. *PVLDB* 2, 1 (2009), 25–36.
[28] Wei Zhang, Jian Su, Chew Lim Tan, and WenTing Wang. 2010. Entity Linking Leveraging Automatically Generated Annotation. In *COLING*. 1290–1298.
[29] Weiguo Zheng, Lei Zou, Xiang Lian, Dong Wang, and Dongyan Zhao. 2015. Efficient Graph Similarity Search Over Large Graph Databases. *IEEE TKDE* 27, 4 (2015), 964–978.
[30] Lei Zou, Ruizhe Huang, Haixun Wang, Jeffrey Xu Yu, Wenqiang He, and Dongyan Zhao. 2014. Natural language question answering over RDF: a graph data driven approach. In *SIGMOD*. 313–324.