IEEE Transactions on Knowledge and Data Engineering (Volume:28 , Issue: 6)18 January 2016

Quality-Aware Subgraph Matching Over Inconsistent Probabilistic Graph Databases

Xiang Lian, Lei Chen, *Member, IEEE*, and Guoren Wang

Abstract—Resource Description Framework (RDF) has been widely used in the Semantic Web to describe resources and their relationships. The RDF graph is one of the most commonly used representations for RDF data. However, in many real applications such as the data extraction/integration, RDF graphs integrated from different data sources may often contain uncertain and inconsistent information (e.g., uncertain labels or that violate facts/rules), due to the unreliability of data sources. In this paper, we formalize the RDF data by *inconsistent probabilistic RDF graphs*, which contain both inconsistencies and uncertainty. With such a probabilistic graph model, we focus on an important problem, *quality-aware subgraph matching over inconsistent probabilistic RDF graphs* (QA-gMatch), which retrieves subgraphs from inconsistent probabilistic RDF graphs that are isomorphic to a given query graph and with high quality scores (considering both consistency and uncertainty). In order to efficiently answer QA-gMatch queries, we provide two effective pruning methods, namely *adaptive label pruning* and *quality score pruning*, which can greatly filter out false alarms of subgraphs. We also design an effective index to facilitate our proposed pruning methods, and propose an efficient approach for processing QA-gMatch queries. Finally, we demonstrate the efficiency and effectiveness of our proposed approaches through extensive experiments.

Index Terms—Quality-Aware subgraph matching, inconsistent probabilistic graph databases, adaptive label pruning, quality score pruning.

1 Introduction

RDF (Resource Description Framework) is a W3C standard to describe resources on the Web and their relationships in the Semantic Web [1]. Specifically, RDF data can be represented by either triples in the form of (*subject*, *predicate*, *object*), or an equivalent graph representation.

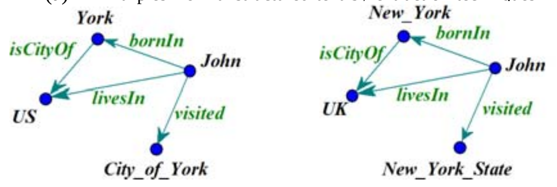
Figure 1(a) shows an example of RDF triples extracted from unstructured text, by using two different data extraction methods. Specifically, the left column depicts 4 RDF triples by using extraction technique A, whereas the right column shows another 4 RDF triples obtained from extraction technique B. For example, on the left column of Figure 1(a), the first triple, (⟨John⟩, ⟨bornIn⟩, ⟨York⟩), has subject ⟨John⟩, predicate ⟨bornIn⟩, and object ⟨York⟩, which indicates that John was born in the city of York.

Equivalently, 4 RDF triples on the left column of Figure 1(a) can be transformed to a graph, G_A, as shown in Figure 1(b). For example, Triple (1): (⟨John⟩, ⟨bornIn⟩, ⟨York⟩) can be converted into a directed edge (with label "bornIn") from vertex ⟨John⟩ to vertex ⟨York⟩. Similarly, we can also obtain graph G_B (in Figure 1(c)) from 4 RDF triples on the right column of Figure 1(a).

Due to the unreliability of data sources [15], [17] (e.g., the data expiration or the inaccuracy of data extraction techniques [27]), RDF graphs from different sources might contain imprecise or inconsistent information. In the example of Figure 1, by applying inaccurate extraction techniques A and B to some unstructured text (e.g., Wikipedia data), we may obtain two distinct RDF graphs, G_A and G_B, in Figures 1(b) and 1(c), respectively. In particular, graphs G_A and G_B may have conflicting vertex labels (e.g., John's birthplace is confusing, either "York" or "New_York"). In the applications such as data extraction/integration [10], [11], in order to resolve such conflicting labels, we can merge multiple versions of RDF graphs into a single probabilistic RDF graph, where each vertex is associated with its possible labels and their confidences to be true in reality (inferred from the extraction accuracy or reliability statistics of data sources over historical data).

Extraction Technique A:	**Extraction Technique B:**
(1)(⟨John⟩, ⟨bornIn⟩, ⟨York⟩);	(1') (⟨John⟩, ⟨bornIn⟩, ⟨New_York⟩);
(2) (⟨John⟩, ⟨livesIn⟩, "US");	(2') (⟨John⟩, ⟨livesIn⟩, "UK");
(3) (⟨York⟩, ⟨isCityOf⟩, "US");	(3') (⟨New_York⟩, ⟨isCityOf⟩, "UK");
(4) (⟨John⟩, ⟨visited⟩, "City_of_York").	(4') (⟨John⟩, ⟨visited⟩, "New_York_State").

(a) RDF triples from unstructured text by extraction techniques A and B



(b) RDF graph G_A (c) RDF graph G_B

Fig. 1. An example of RDF triples and graphs by Extraction Techniques A and B.

As an example, Figure 2 illustrates a probabilistic RDF graph G, which is integrated from two (inconsistent) graphs G_A and G_B in Figures 1(b) and 1(c), respectively. Each vertex in G is associated with uncertain labels and their existence probabilities. For example, the vertex at the bottom of G in Figure 2 has two possible labels, "New_York_State" and "City_of_York". That is,

- X. Lian is with the Department of Computer Science, University of Texas Rio Grande Valley, Edinburg, TX 78539, USA, Email: xiang.lian@utrgv.edu.
- L. Chen is with the Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Kowloon, Hong Kong, China, Email: leichen@cse.ust.hk.
- G. Wang is with the School of Information Science and Engineering, Northeastern University, Shenyang, Liaoning 110819, China. E-mail: wanggr@mail.neu.edu.cn.

information.

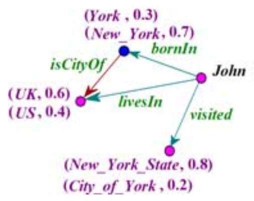


Fig. 2. A probabilistic RDF graph G integrated from graphs G_A and G_B in Figures 1(b) and 1(c), respectively.

John visited "New_York_State" with probability 0.8, or the "City_of_York" with probability 0.2, where probabilities can be inferred from the accuracy of information extraction (IE) methods [27]. Similarly, the country (vertex) John lives in is either "UK" with probability 0.6 or "US" with probability 0.4; the city in which John was born is either "York" with probability 0.3, or "New_York" with probability 0.7. This way, 2 graphs from unreliable data sources can be integrated into one probabilistic graph, which incorporates inconsistent vertex labels by a probabilistic model.

Following the literature of probabilistic databases [9], we consider the *possible worlds* semantics overs probabilistic RDF graphs, where each possible world is a materialized instance of the graph with (certain) vertex labels appearing in the real world. In Figure 2, we can obtain a possible world when each vertex is assigned with a deterministic (certain) label, for example, the birthplace (vertex) of John takes label "New_York", the country vertex takes label "UK", and the vertex of the visited place has label "New_York_State".

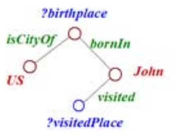
Although the probabilistic RDF graph can somewhat incorporate conflicting labels in vertices, inconsistencies may still exist in edges for some possible worlds of the probabilistic graph, violating facts/rules. For example, in probabilistic graph G of Fig. 2, one possible world of G may contain vertex labeled by "New_York" connecting with "UK" through edge "isCityOf", which is however violating the fact that New York is a city in US (rather than UK). Therefore, we call such inconsistent RDF data (violating facts/rules) an *inconsistent probabilistic graph*.

To resolve the inconsistencies and guarantee the data quality in possible worlds, we adopt the *X-repair semantics* [7], which delete edges in the graph such that the remaining graph has consistent labels, obeying facts/rules. Intuitively, some edges (RDF triples) in the graph are not reliable, and should not exist in reality. Thus, X-repair semantics consider removing such edges from the graph in order to improve the data quality. In the example of Figure 2, edge "isCityOf" can be deleted in a possible world, if it has two ending vertices with inconsistent labels "New_York" and "UK" (or "York" and "US").

In this paper, we propose the *quality-aware subgraph matching* problem (namely, QA-gMatch) in a novel context of inconsistent probabilistic graphs G with quality guarantees. Specifically, given a query graph q, a QA-gMatch query retrieves subgraphs g of probabilistic graph G that match with q and have high *quality scores* (defined later in Section 2.3). Note that, a single repaired graph via edge deletions may have corrupted graph structure, and fail to return matching subgraphs (e.g., by deleting edge

select	$?visitedPlace$ $?birthplace$
where	$\{\langle$John$\rangle$ $\langle$visited$\rangle$ $?visitedPlace.$
	$\langle$John$\rangle$ $\langle$bornIn$\rangle$ $?birthplace.$
	$?birthplace$ $\langle$isCityOf$\rangle$ "US". $\}$

(a)



(b)

Fig. 3. A query graph q transformed from a SPARQL query.

TABLE 1
Symbols and descriptions.

Symbol	Description
G	a probabilistic RDF graph
$pw(G)$	a possible world of a probabilistic RDF graph G
$pw^R(G)$	a repaired possible world of $pw(G)$
v_i	a vertex in probabilistic RDF graph G
$l(v_i).p$	the existence probability of a possible label $l(v_i)$ of vertex v_i
e_{ij}	a directed edge $\overrightarrow{v_i v_j}$
$e_{ij}.rp$	the repair confidence that edge e_{ij} should be deleted

"isCityOf" in Figure 2, no answer will be returned for any query graph q that contains edge "isCityOf"). Thus, instead, our QA-gMatch problem will consider subgraph answers over all possible repairs in possible worlds of G (i.e., all-possible-repair semantics [22]), and then return those subgraph answers with good quality scores.

The QA-gMatch problem has many practical applications such as the Semantic Web. For example, we can answer standard queries, SPARQL queries, over inconsistent probabilistic RDF graphs by issuing QA-gMatch queries. Figure 3(a) shows an example of a SPARQL query, which obtains the place visited by John, as well as John's birth place. Equivalently, we can transform the SPARQL query to a query graph q (as given in Figure 3(b)). Then, within inconsistent probabilistic RDF graph G (given by Figure 2), we can conduct a QA-gMatch query to find those subgraphs $g \subseteq G$ that are isomorphic to q with high quality scores, where quality scores indicate the confidences that subgraphs appear in the repaired probabilistic graphs of G.

One straightforward method to solve the QA-gMatch problem is to offline enumerate all possible worlds of probabilistic RDF graph G, repair these possible worlds (via edge deletions), and obtain subgraphs with high quality scores (QA-gMatch query answers) from the repaired possible worlds. However, since there are an exponential number of repaired possible worlds, this method is very inefficient, or even infeasible, to directly repair/store/query on the materialized possible worlds, in terms of time and space costs. Therefore, it is challenging to efficiently process the QA-gMatch query. In this paper, we will propose effective pruning methods, namely *adaptive label pruning* (based on a cost model) and *quality score pruning*, to reduce the QA-gMatch search space and improve the query efficiency.

We make the following contributions in this paper.

1) We propose the QA-gMatch problem in inconsistent probabilistic graphs, which, to our best knowledge, no prior work has studied.
2) We carefully design effective pruning methods, adaptive label and quality score pruning, specific for inconsistent and probabilistic features of RDF graphs.
3) We build a tree index over pre-computed data of inconsistent probabilistic graphs, and illustrate efficient QA-gMatch query procedure by traversing the index.

4) We demonstrate through extensive experiments the query performance of our QA-gMatch approaches.

Section 7 reviews prior works on inconsistent, graph, and probabilistic databases. Section 8 concludes this paper.

2 Problem Definition

Table 1 depicts the commonly used symbols in this paper.

2.1 Data Model for Probabilistic Graphs

Probabilistic graphs: In this subsection, we first give the data model for probabilistic RDF graph.

Definition 2.1: (Probabilistic Graph) A probabilistic graph G is a triple $\langle V(G), E(G), \Theta(G)\rangle$ such that:

- $V(G)$ is a finite set of vertices, each of which, v_i, is associated with a label set $L(v_i)$ containing possible labels, $l(v_i)$, with probability $l(v_i).p$;
- $E(G)$ is a finite set of directed edges, each of which, e_{ij}, is associated with a label $l(e_{ij})$; and
- $\Theta(G)$ is a mapping from $V(G) \times V(G)$ to $E(G)$, which contains $(v_i, v_j) \rightarrow e_{ij}$, indicating that edge e_{ij} connects vertices from v_i to v_j in graph G. ■

In Definition 2.1, we define a probabilistic RDF graph as a graph structure that has uncertain vertex labels. That is, each vertex v_i has one or multiple *mutually exclusive* labels $l(v_i)$ with *existence probabilities* $l(v_i).p \in (0, 1]$, where $\sum_{\forall l(v_i)} l(v_i).p = 1$. In this paper, we assume that there are no NULL values for vertex labels. Nonetheless, our graph model can be easily extended by allowing label probabilities $\sum_{\forall l(v_i)} l(v_i).p \leq 1$ (i.e., the vertex has the NULL value with probability $1 - \sum_{\forall l(v_i)} l(v_i).p$).

To our best knowledge, this work is the first effort to study the quality-aware subgraph matching in inconsistent probabilistic graphs. We simply assume that each vertex stores possible labels, which are independent of labels in other vertices. The case of correlated labels can be extended by considering joint probabilities among labels in vertices.

In real applications such as the Semantic Web, the probabilistic graph (mentioned in Definition 2.1) can be obtained by the data extraction/integration as follows. Assume that we have the RDF schema (e.g., ontology), which contains meta information in application domains, for example, $(\langle cityName\rangle, \langle isCityOf\rangle, \langle countryName\rangle)$. We can extract labels of vertices (entities) from unstructured text data, by assigning each word (token) in the unstructured text with a label (tag), using either *Hidden Markov Model* (HMM) [24] or *Conditional Random Field* (CRF) model [18]. For example, tokens "$York$" and "UK" can be tagged by labels $\langle cityName\rangle$ and $\langle countryName\rangle$, respectively. In addition, we can also infer the tagging accuracy of subjects or objects by HMM or CRF. In other words, vertex labels are associated with probabilities that tags (labels) can truly describe entities (i.e., subjects or objects). Moreover, during the semantic parsing, the unstructured texts are annotated with predicate-argument structures, called *rolesets* [12], such as **isCityOf(cityName, countryName)** which takes two arguments (entities) "cityName" and "countryName". Such rolesets can be used as a dictionary, which uniquely maps the entities (and the predicate as well) onto the ontology (i.e., the RDF schema of the knowledge base) via the ontology mapping [12]. As a result, we can obtain edges with deterministic labels (e.g., "isCityOf"), connecting entities, which form probabilistic RDF graphs (with uncertain vertex labels and deterministic edge labels).

Nonetheless, our proposed techniques (e.g., encoding signatures and pruning methods) in this paper can be easily extended to probabilistic graphs with both vertex and edge label uncertainties, which will be discussed in Section 5.

Possible worlds of a probabilistic graph: Similar to probabilistic databases [9], we consider the *possible worlds* semantics of a probabilistic graph. Specifically, each possible world is a materialized instance of the probabilistic graph, which corresponds to a certain graph with one label assignment in vertices that may appear in the real world. We formally define possible worlds in a probabilistic graph.

Definition 2.2: (Possible Worlds of a Probabilistic Graph) Given a probabilistic graph G, a *possible world*, $pw(G)$, of G is a materialized graph where each vertex $v_i \in V(G)$ is assigned with a certain label $l(v_i)$.

Then, the appearance probability, $Pr\{pw(G)\}$, of a possible world $pw(G)$ is given by:

$$Pr\{pw(G)\} = Pr\left\{\bigwedge_{\forall i}(X_i = l(v_i))\right\} = \prod_{\forall v_i \in V(G)} l(v_i).p. \qquad (1)$$

where X_i is a variable of possible label $l(v_i)$ that vertex v_i may be assigned with. ■

In Definition 2.2, each possible world corresponds to one vertex label assignment to the probabilistic graph. Since each vertex v_i can have more than one possible label, that is, $|L(v_i)|$ (≥ 1), the total number of possible worlds (or label combinations) is usually exponential, $\prod_{\forall v_i \in V(G)} |L(v_i)|$, where $|S|$ is the number of elements in a set S. The appearance probability of the possible world is given by multiplying existence probabilities of labels in vertices (due to the independence of labels).

2.2 Inconsistencies in Probabilistic Graphs

As mentioned in Section 1, probabilistic graphs are often obtained from real-world applications such as the data extraction/integration in the Semantic Web. Due to the unreliability of data sources or inaccurate extraction/integration techniques, probabilistic graph data often contain inconsistencies, violating some rules or facts. Here, rules or facts can be specified by knowledge base or inferred by data mining techniques, which is out of the scope of this paper.

As an example in Figure 2, an edge e_{ij} (associated with label $\langle isCityOf\rangle$) has two ending vertices v_i and v_j, where vertex v_i has two possible labels "$York$" and "New_York", and vertex v_j has two uncertain labels "UK" and "US". Here, a well-known fact is that York should be a city of UK, and New York is a city of US. Thus, in a possible world that contains an edge e_{ij} with ending vertices labeled by "$York$" and "US" (or "New_York" and "UK"), labels in vertices v_i and v_j are said to be inconsistent, violating the fact.

Thus, two ending vertices of an edge are inconsistent, if in some possible worlds their labels violate facts/rules. We call a probabilistic graph that contains such inconsistent vertex labels the *inconsistent probabilistic graph*.

2.3 The QA-gMatch Problem

Repair in inconsistent probabilistic graphs: To resolve inconsistencies and improve the data quality in possible worlds of inconsistent probabilistic graphs, we adopt an idea similar to the X-repair semantics [7] which repair data by deleting tuples in relational tables. In particular, we can repair inconsistent probabilistic graphs by deleting edges in the graph such that the remaining graphs become consistent (i.e., following facts/rules).

In Figure 2, consider a possible world of graph G that contains an $\langle isCityOf\rangle$ edge e_{ij} with inconsistent labels $l(v_i) =$"$York$" and $l(v_j) =$"US". To repair this possible world (i.e., an instance of graph G), we can delete their in-between edge e_{ij}, and only retain two disconnected vertices v_i and v_j, such that there is no inconsistency in the repaired graph. Here, the repaired graph indicates that John was born in York and lives in the US, but there is no relationship between birth and living places.

Note that, in real applications such as data integration, for inconsistent labels, we cannot directly repair (delete) edges for each data source in a pre-processing step before the data integration, since this may lead to the corruption of the integrated probabilistic graph. For example, in Figures 1(b) and 1(c), if we delete edges $\langle isCityOf\rangle$ between the city name and country name, then the integrated graph in Figure 2 would permanently lose this edge, and we may lose important query results, due to the missing edge.

We give the definition of the repair in possible worlds of inconsistent probabilistic graphs below.

Definition 2.3: (Repair in Possible Worlds of an Inconsistent Probabilistic Graph) Given a fact table, FT, indicating the relationships between vertex labels, and a possible world $pw(G)$ of a probabilistic graph G, a repair of $pw(G)$ is to delete a set of edges such that the resulting *repaired possible world*, $pw^R(G)$, becomes consistent. ■

In Definition 2.3, the fact table FT contains tuples of facts that indicate the true relationships between subjects and objects. Here, facts in FT can be obtained from some knowledge base or learnt from rules carefully confirmed by domain experts. They can be used to determine whether or not an edge is inconsistent in the probabilistic graph.

As an example, one possible fact triple in FT is (⟨York⟩, ⟨isCityOf⟩, "UK"), which states that York is a city of UK. In the case of an edge corresponding to triple (⟨York⟩, ⟨isCityOf⟩, "US"), we say that this edge in the graph is inconsistent, since it violates the fact triple in FT.

A repaired possible world $pw^R(G)$ is a subgraph of the original possible world $pw(G)$, by deleting those edges in $pw(G)$, such that $pw^R(G)$ is consistent (i.e., with edges that do not violate facts in FT).

Note that, due to unreliable data sources or inaccurate extraction/integration techniques, some edge connections in a probabilistic graph may be erroneous, that is, sometimes edges may not exist at all in reality. Thus, the repair in probabilistic graphs should not only remove all the inconsistent edges that violate the fact table FT, but also delete some potentially inconsistent edges that are not specified in FT. A repaired possible world, $pw^R(G)$, of the graph is consistent, if there are no inconsistent edges (violating FT) after edge deletions.

To evaluate the confidence of our repair (i.e., edge deletions), we can collect statistics about the reliability of edges in the probabilistic graph. Specifically, we assume that each edge e_{ij} is associated with a *repair confidence* $e_{ij}.rp \in [0, 1]$, which reflects the confidence that edge e_{ij} might not exist and should be repaired (deleted). In real-world applications, repair confidences can be obtained from the accuracy of extraction/integration methods [14], [29].

Thus, for a repaired possible world $pw^R(G)$, its repair weight, $W(pw^R(G))$, is given by:

$$W(pw^R(G)) = \prod_{\forall e_{ij}\in E(G)} \begin{cases} (1-e_{ij}.rp) & \text{if } e_{ij}\in E(pw^R(G)); \\ e_{ij}.rp & \text{otherwise.} \end{cases} \quad (2)$$

Intuitively, a higher repair weight implies higher confidence that we repair over possible world $pw(G)$.

Challenges of repair in probabilistic graphs: The repair in inconsistent probabilistic graphs needs to deal with exponential numbers of possible worlds $pw(G)$ (w.r.t. $|V(G)|$) and repaired possible worlds $pw^R(G)$ (w.r.t. $|E(G)|$). For almost any query, the query answering over so many (repaired) possible worlds is not efficient or even not feasible, in terms of time and space costs.

Thus, in this paper, rather than materializing and repairing all possible worlds directly, we will propose a metric, that is, quality score, to quantify the quality of the returned QA-gMatch subgraphs, by considering *consistent query answering* (CQA) [4], [21] over inconsistent probabilistic graphs. Specifically, to compute the quality score, we condense combinations of all possible worlds and possible repairs (i.e., the *all-possible-repair* semantics [22]), without materializing possible worlds or modifying the original graph data. To improve the query efficiency, we will also design effective pruning/indexing mechanisms and QA-gMatch algorithms on inconsistent probabilistic graphs.

QA-gMatch in an inconsistent probabilistic graph: In this paper, we aim to tackle the quality-aware subgraph matching problem over inconsistent probabilistic graphs G, under the *all-possible-repair* semantics [22] (i.e., exploring all possible repairs on possible worlds of probabilistic graphs). Specifically, our problem retrieves subgraphs in G that match with a given query graph and have their quality scores higher than a threshold.

Definition 2.4: (Quality-Aware Subgraph Matching in Inconsistent Probabilistic Graphs, QA-gMatch) Given an inconsistent probabilistic graph G, a fact table FT, a query graph q, and a threshold α_g, a *quality-aware subgraph matching in inconsistent probabilistic graph* (QA-gMatch) retrieves subgraphs $g \subseteq G$ such that:

- g is isomorphic to q (denoted as $g \equiv q$); and
- the quality score $score(g) > \alpha_g$.

where $score(g)$ is defined as:

$$score(g) = \sum_{\forall pw(G)} Pr\{pw(G)\} \cdot \chi(\exists pw^R(G), g \subseteq pw^R(G) \wedge g \equiv q) \cdot W(g). \quad (3)$$

Here, if z is *true*, $\chi(z) = 1$; otherwise, $\chi(z) = 0$. Moreover, $g \equiv q$ is *true*, if g and q are isomorphic. ■

In Definition 2.4, the query graph q contains information such as the graph structure and query labels $l(q_i)$ in vertices $q_i \in V(q)$. Subgraph g is the QA-gMatch answer, if g and q are isomorphic (in terms of both graph structure and label matching) and their quality scores are above threshold α_g. Note that, here we do not require specifying query labels for all vertices in query graph q (equivalently transformed from the SPARQL query for RDF data). When the label of some vertex q_i is not specified, we will consider it as a wildcard (i.e., being able to match with any label) during the isomorphism checking ($g \equiv q$).

The QA-gMatch problem (given in Definition 2.4) is very useful in real-world applications. As mentioned in Section 1, in the Semantic Web, we can use QA-gMatch to answer SPARQL queries over an inconsistent probabilistic RDF graph G. In particular, a SPARQL query can be translated into a query graph q (see an example in Figure 3), in which labels of vertices are obtained from conditions in the where clause. Then, the answering of the SPARQL query on inconsistent probabilistic RDF graph G corresponds to the process of solving the QA-gMatch problem, that is, searching subgraphs g in G that are isomorphic to q with high quality scores (i.e., $score(g) > \alpha_g$).

Due to probabilistic and inconsistent features of inconsistent probabilistic graphs, efficient QA-gMatch processing is quite challenging. One straightforward method is to consider every subgraph $g \subseteq G$ that is isomorphic to query graph q, and compute its quality score $score(g)$ by enumerating an exponential number of the repaired possible worlds. However, this method is quite costly. That is, the isomorphism checking between graphs g and q is NP-hard; what is worse, for each subgraph g ($\subseteq G$), the time complexity of computing the score $score(g)$ is given by $O(2^{|V(G)|+|E(G)|})$, which is very high and unacceptable. Inspired by this, in the sequel, we will propose effective pruning methods to reduce the QA-gMatch search space, and design indexing mechanisms to facilitate the pruning during the QA-gMatch processing.

3 Problem Reduction

As mentioned in Definition 2.4, to obtain QA-gMatch answers (i.e., subgraphs g), we need to compute the quality score $score(g)$ (given in Eq. (3)), which involves an exponential number of repaired possible worlds. The direct computation of $score(g)$ over repaired possible worlds is rather costly. Below, we will derive the formula of $score(g)$ so that this score can be computed over edges of graph g.

Lemma 3.1: (The Computation of quality score, $score(g)$) The quality score, $score(g)$, given in Eq. (3) can be written as:

$$score(g) = \left(\prod_{\forall e_{ij} \in E(g)} (1 - e_{ij}.rp) \right) \cdot \left(\prod_{\forall v_i \in V(g)} l(v_i).p \right) \quad (4)$$

Proof: Please refer to Appendix A in supplementary materials. □

Lemma 3.1 reduces our QA-gMatch problem over an exponential number of (repaired) possible worlds in probabilistic RDF graph G to the one on vertices/edges in subgraphs g ($\subseteq G$). After the problem reduction, the time complexity to compute the quality score becomes $O(|E(g)| + |V(g)|)$.

4 Pruning Strategies

4.1 Adaptive Label Pruning

In this subsection, we design an adaptive label pruning method specific for probabilistic graphs, which adaptively encodes label/structural information in signatures and filters out false alarms of QA-gMatch candidates via signatures.

Here, the design of signatures takes into account a special feature of probabilistic RDF graphs, that is, some vertices in graphs may incur high degrees. Thus, our basic idea of the signature design is to adaptively allocate more space for encoding vertices with high degrees (less space for those with low degrees), and maximize the pruning power. We also design a cost model to guide the signature generation.

4.1.1 Design of Label Signatures

We first present the details of our label signatures for probabilistic graphs, which can be used for our adaptive label pruning. Specifically, for each vertex $v_i \in V(G)$, we will produce its signatures $sig(v_i)$ for labels of its surrounding (k-hop) vertices/edges. Without loss of generality, we first discuss label signatures for vertices below.

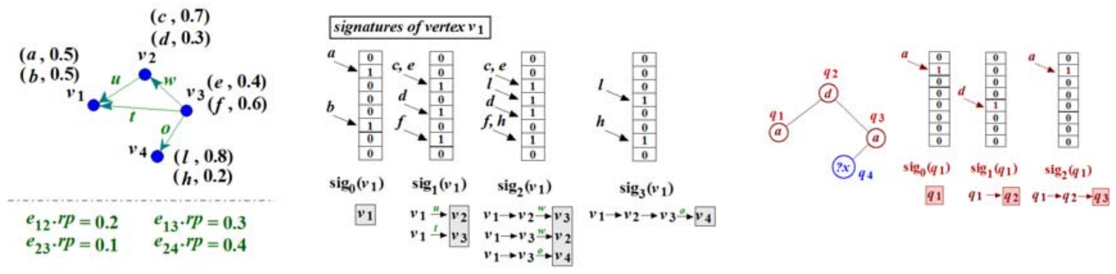
Label signature for vertices: We will generate the label signature $sig(v_i)$ for each vertex $v_i \in V(G)$, which contains the label information of vertex v_i and its surrounding neighbors in the graph. Intuitively, we can use such a label signature to check the existence of a vertex label, and thus enable the pruning for the QA-gMatch search (which will be discussed later in Section 4.1.2).

Particularly, to build label signature $sig(v_i)$, we start from vertex v_i, and traverse the graph to enumerate all paths with lengths equal to k, where $0 \le k \le k_{max}$ and k_{max} is the maximum possible path length in a query graph q.

Then, we can obtain k_{max} vertex sets, $S_k(v_i)$ ($0 \le k \le k_{max}$), which contain all vertices v_j that have path lengths to v_i equal to k. For each vertex set $S_k(v_i)$, we maintain a label signature, $sig_k(v_i)$, which is a bit vector of size B and encodes all possible labels in vertex set $S_k(v_i)$. Initially, all elements, $sig_k(v_i)[z]$ ($0 \le z < B$), in the label signature $sig_k(v_i)$ are zeros. Then, for each possible label $l(v_j)$ of vertex $v_j \in S_k(v_i)$, we map it to a random position in $sig_k(v_i)$ by using a hashing function $H(x)$ (i.e., $H(l(v_j))$-th position), and set $sig_k(v_i)[H(l(v_j))]$ to 1.

In this paper, we use a hashing function $H(x) = (a \cdot x + b) mod B$ to produce random numbers within $[0, B)$, where a and b are random numbers. To reduce the chance of label conflicts (i.e., distinct labels mapped to the same positions), we can utilize multiple hashing functions $H(x)$ (with distinct (a, b)-pairs) on different bit vectors. Nonetheless, for the sake of clear illustration, in this paper, we will simply use one hash function on each signature $sig_k(v_i)$.

Example of vertex label signatures: Figure 4 illustrates an example of encoding labels of vertices via signatures. In particular, Figure 4(a) shows a small probabilistic graph G with 4 vertices v_1, v_2, v_3, and v_4. Moreover, each vertex (e.g., v_1) in G is associated with its possible labels and



(a) inconsistent probabilistic graph G (b) label signatures of vertex v_1 ($k_{max} = 3$) (c) query graph q

Fig. 4. Illustration of label signatures and adaptive label pruning.

appearance probabilities (e.g., label a with probability 0.5 and b with probability 0.5).

In Figure 4(b), we show how to build 4 label signatures, $sig_0(v_1)$, $sig_1(v_1)$, $sig_2(v_1)$, and $sig_3(v_1)$, for vertex v_1, where each signature is a bit vector of size 8 ($= B$).

For $sig_0(v_1)$, since vertex v_1 has 0 distance to itself, v_1 is the only vertex in set $S_0(v_1)$, and we thus hash v_1's labels a and b into the signature.

Then, for signature $sig_1(v_1)$, there are two paths $v_1 \rightarrow v_2$ and $v_1 \rightarrow v_3$ of length 1. Thus, we can obtain vertex set $S_1(v_1) = \{v_2, v_3\}$, which contains vertices that are 1-hop away from v_1. Next, we map possible labels of v_2 and v_3 into the signature $sig_1(v_1)$. Here, labels c and e have conflicts mapping to the same position in the bit vector.

As illustrated in Figure 4(b), starting from the 2 paths with length 1 (for $sig_1(v_1)$), we can continue to expand them by including vertex v_2, v_3, or v_4. As a result, for signature $sig_2(v_1)$, we can have 3 paths of length 3, and obtain vertex set $S_2(v_1) = \{v_2, v_3, v_4\}$. Similarly, we can hash their vertex labels into bit vector, and obtain $sig_2(v_1)$.

Finally, for signature $sig_3(v_1)$, we can only expand path $v_1 \rightarrow v_2 \rightarrow v_3$ (of length 2 for $sig_2(v_1)$) by including vertex v_4. We cannot expand the other two paths of length 2, since all connecting vertices have been visited in those 2 paths (e.g., for path $v_1 \rightarrow v_3 \rightarrow v_2$, both vertices v_1 and v_3 that are adjacent to v_2 have been accessed in the path). Thus, we have vertex set $S_3(v_1) = \{v_4\}$, and labels l and h are hashed into $sig_3(v_1)$. ■

Label signature for edges: The case of edge label signatures is quite similar to label signatures for vertices. We also start from each vertex $v_i \in V(G)$, and traverse graph G to generate edge label signatures, $sig^e_k(v_i)$, for vertices with path lengths to v_i equal to k, where $1 \le k \le k_{max}$.

Specifically, for each path from v_i to v_j with length k, we first obtain the label of the last edge $e_{.j}$ on this path, and then hash the label into the $H(l(e_{.j}))$-th position in $sig^e_k(v_i)$. This way, we can construct edge label signatures for labels of the k-th edges on paths from v_i.

Example of edge label signatures: In the previous example of Figure 4(a), edge signature $sig^e_1(v_1)$ encodes edge labels $\{u, t\}$, $sig^e_2(v_1)$ contains labels $\{w, o\}$, and $sig^e_3(v_1)$ represents label $\{o\}$. ■

For differences of our synopses from that in prior works, please refer to Appendix B.

4.1.2 Pruning with Label Signatures

Next, we illustrate how to enable the pruning with label signatures discussed above, by using an example in Figure 4. Assume that we have a query graph q, as shown in Figure 4(c). Similar to the construction of label signature in graph G, we start from vertex $q_1 \in V(q)$, and generate signatures $sig_k(q_1)$, where $0 \le k \le 2$.

Suppose graph G is a candidate graph that may match with query graph q. We want to utilize the signatures, $sig_k(q_1)$ and $sig_k(v_1)$, to enable the pruning. As an example, if q_1 matches with v_1, then label a specified by q_1 should also appear in vertex v_1. In other words, it must hold that: $sig_0(q_1)[1] = sig_0(v_1)[1] = 1$.

The lemma below enables label pruning via signatures.

Lemma 4.1: Given a query graph q and a subgraph g in a probabilistic RDF graph G, if subgraph g is matching with q, then for all corresponding vertex pairs (q_i, v_i), and for any $0 \le k \le k_{max}$, we have:

$$sig_k(q_i) \bigwedge sig_k(v_i) = sig_k(q_i) \tag{5}$$

where $\bigwedge$ is the bit-AND operator that is performed over two bit vectors of size B.

From the matching property with signatures in Lemma 4.1, we can immediately obtain the pruning with label signatures in the theorem below.

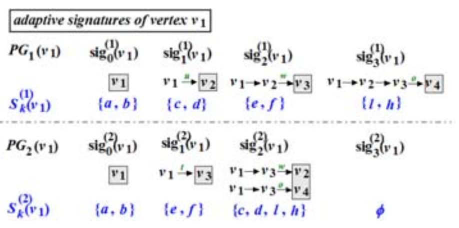
Theorem 4.1: (Pruning with Label Signatures) Given a query graph q and a subgraph g in a probabilistic RDF graph G (with corresponding vertex pairs (q_i, v_i)), the subgraph g is not a QA-gMatch answer, if Eq. (5) does not hold for some vertex pair (q_i, v_i) and some $k \in [0, k_{max}]$.

In the example of Figure 4(c), we can check the condition in Eq. (5) for all signature pairs $(sig_k(q_i), sig_k(v_i))$, where k varies from 0 to 2. When $k = 0$ or 1, the condition holds. However, for $k = 2$, we have $sig_2(q_i) \bigwedge sig_2(v_i) = 0 \neq sig_2(q_i)$, indicating that label a exists in query vertex q_3 (with distance 2 to q_1), but it does not exist in vertex v_3, v_2, or v_4 (with distance 2 to v_1). Thus, we know that q_1 cannot match with v_1, and we can safely prune all the subgraph candidates that have vertex v_1 matching with q_1.

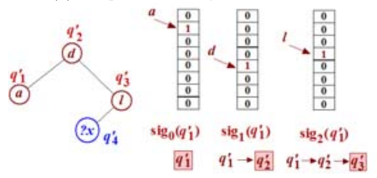
4.1.3 Adaptive Label Signatures

Up to now, we have discussed the data structure for storing vertex/edge label signatures. In the sequel, we will design *adaptive label signatures*, which can optimize the pruning power of label signatures, adaptive to specific features of vertices in probabilistic RDF graphs (e.g., label distributions, vertex degrees, and structural information).

Adaptive vertex/edge label signatures: We observe that, in a probabilistic RDF graph, there are many vertices with high degrees. As a consequence, if we construct signatures $sig(v_i)$ for a vertex v_i of high degree $deg(v_i)$ (e.g., $\gg B$),


adaptive signatures of vertex v_1
$PG_1(v_1)$ $sig_0^{(1)}(v_1)$ $sig_1^{(1)}(v_1)$ $sig_2^{(1)}(v_1)$ $sig_3^{(1)}(v_1)$
v_1 $v_1 \to v_2$ $v_1 \to v_2 \to v_3$ $v_1 \to v_2 \to v_3 \to v_4$
$S_k^{(1)}(v_1)$ $\{a, b\}$ $\{c, d\}$ $\{e, f\}$ $\{l, h\}$
$PG_2(v_1)$ $sig_0^{(2)}(v_1)$ $sig_1^{(2)}(v_1)$ $sig_2^{(2)}(v_1)$ $sig_3^{(2)}(v_1)$
v_1 $v_1 \to v_3$ $v_1 \to v_3 \to v_2$ $v_1 \to v_3 \to v_4$
$S_k^{(2)}(v_1)$ $\{a, b\}$ $\{e, f\}$ $\{c, d, l, h\}$ ϕ

(a) adaptive label signatures ($m = 2$)


q'_1 q'_2 q'_3 q'_4
a d l ?x
$sig_0(q'_1)$ $sig_1(q'_1)$ $sig_2(q'_1)$
q'_1 $q'_1 \to q'_2$ $q'_1 \to q'_2 \to q'_3$

(b) query graph q'

Fig. 5. Illustration of adaptive label pruning.

then most bits in signature $sig_1(v_i)$ are very likely to be set to 1. This indicates that there are many conflicting labels mapping into the same positions in the signature, which would incur low or even no pruning power.

Inspired by the special feature of high degrees in probabilistic RDF graph, we propose to adaptively divide all paths (starting with vertex v_i) into m groups, $PG_1(v_i)$, $PG_2(v_i)$, ..., $PG_m(v_i)$, where m is an adaptive parameter within $[1, m_{max}]$, and m_{max} is constrained by the space to store signatures of a vertex in index nodes (which will be discussed later).

Intuitively, we adaptively create different numbers of signatures specific for each vertex $v_i \in V(G)$, such that the conflict rate of this vertex can be reduced.

Clearly, when $m = 1$, it exactly corresponds to the case in Section 4.1.1, where we build signatures on all paths (as a single group), which may lead to high conflict rates (in turn, low pruning power). On the other hand, if we consider each path as one individual group, then we can achieve the lowest conflict rate in signatures (i.e., the highest pruning power). However, in this case, the number of groups (paths) is also large, which requires higher computation cost to check more signatures. Thus, there is a trade-off between conflict rate (pruning power) and computation cost. We will delay the discussion of obtaining groups of paths, based on a cost model, to Section 4.1.4.

After we obtain path groups, for each group $PG_{m'}(v_i)$, we can construct signatures, denoted as $sig_k^{(m')}(v_i)$, by encoding labels of last vertices in paths (similar to the example in Figure 4(b)), where $1 \le m' \le m$.

<u>Example of adaptive label signatures:</u> *We illustrate adaptive label signatures by using the example of a probabilistic RDF graph in Figure 4(a), where $m = 2$. Instead of encoding labels for all paths surrounding vertex v_1 (as shown in Figure 4(b)), Figure 5(a) divides all (longest) paths into 2 partitions,*

$$PG_1(v_1) = \{v_1 \to v_2 \to v_3 \to v_4\}, \text{ and}$$

$$PG_2(v_1) = \{v_1 \to v_3 \to v_2, v_1 \to v_3 \to v_4\}.$$

Then, for each group $PG_{m'}(v_1)$, we can construct signatures $sig_k^{(m')}(v_1)$ by hashing uncertain labels of k-hop vertices from v_1 in set $S_k^{(m')}(v_1)$, where $m' \in \{1, 2\}$, and $0 \le k \le 3$.

For example, in path group $PG_2(v_1)$, to obtain signature $sig_1^{(2)}(v_1)$, we first obtain vertex set $S_1^{(2)}(v_1)$ which contains vertex v_3. Thus, we can hash labels, e and f, of v_3 into signature $sig_1^{(2)}(v_1)$. ■

Pruning with adaptive vertex/edge label signatures: Given adaptive vertex/edge label signatures, we are now ready to utilize them to apply the adaptive label pruning method to reduce the QA-gMatch search space, which is described in the following theorem.

Theorem 4.2: (Adaptive Label Pruning) Given a query graph q and a subgraph g in a probabilistic RDF graph G (with corresponding vertex pairs (q_i, v_i)), the subgraph g is not a QA-gMatch answer, if for some vertex pair (q_i, v_i) and for all path groups $PG_{m'}(v_i)$, Eq. (5) does not hold with some $k \in [0, k_{max}]$.

With adaptive label signatures, we can achieve higher pruning power, as illustrated in the example below.

<u>Example of adaptive label pruning:</u> *Figure 5(b) shows a query graph q', where query nodes q'_1, q'_2, and q'_3 have labels a, b, and l, respectively. Our goal is to check whether labels of q' is matching with that of a subgraph g of probabilistic RDF graph G given in Figure 4(a) (i.e., the graph G without edge e_{13}).*

If we still apply the pruning with label signatures ($m = 1$) in Figure 4(b), subgraph g cannot be pruned (since the condition in Eq. (5) holds for all $k = 0 \sim 2$).

On the other hand, if we use adaptive label signature ($m = 2$) in Figure 5(a), we can safely prune subgraph g. This is because for path group $PG_1(v_1)$, we have label l (in q'_3) is not hashed into the same position as labels e and f (thus, $sig_2^{(1)}(v_1) \wedge sig_2^{(1)}(q'_1) \ne sig_2^{(1)}(q'_1)$ hold); for path group $PG_2(v_1)$, label d (in q'_2) is not hashed into the same location as labels e and f (i.e., $sig_1^{(2)}(v_1) \wedge sig_1^{(2)}(q'_1) \ne sig_1^{(2)}(q'_1)$ hold). Since subgraph g can be pruned in both groups, g cannot be the QA-gMatch answer. ■

4.1.4 Cost Model for Adaptive Label Signatures

We now discuss the remaining issue how to obtain m path groups $PG_1(v_i)$, $PG_2(v_i)$, ..., and $PG_m(v_i)$ from all paths starting with vertex $v_i \in V(G)$, which can be used to construct adaptive label signatures. As mentioned earlier, fine partitioning granularity of path groups can achieve low conflict rates (or high pruning power) but high cost of checking more signatures.

Thus, our goal is to design a cost model to guide the path group partitioning, such that the resulting label signatures can minimize the total computation cost of pruning.

Cost model for the computation cost: To achieve the goal above, we propose a cost model to formalize the total computation cost of signature operations, which is a criterion for the partitioning. Without loss of generality, assume that the unit cost of an operation (e.g., bit-AND or comparison) on two signatures of size B is denoted as C.

For each path group $PG_{m'}(v_i)$, we check the pruning condition in Eq. (5), for k hop values 0, 1, ..., k_{max} in order.

Once the condition does not hold for some $k' \in [0, k_{max}]$, we can stop checking for the remaining k values. Thus, in this case, the computation cost is given by $(k'+1) \cdot C$.

Below, we give the expected computation cost, $Cost_{m'}(v_i)$, of checking pruning conditions via label signatures in a group $PG_{m'}(v_i)$.

$$\begin{aligned} Cost_{m'}(v_i) &= \left(\sum_{k'=0}^{k_{max}} Pr\{PG_{m'}(v_i) \text{ can be pruned at hop } k'\} \cdot (k'+1) \cdot C\right) \\ &\quad + Pr\{PG_{m'}(v_i) \text{ cannot be pruned at hop } k_{max}\} \cdot (k_{max}+1) \cdot C \\ &= \left(\sum_{k'=0}^{k_{max}} P_1(k') \cdot (k'+1) \cdot C\right) + P_2(k_{max}) \cdot (k_{max}+1) \cdot C \end{aligned} \quad (6)$$

In Eq. (6), the first term corresponds to the computation cost when path group $PG_{m'}(v_i)$ can be pruned at some hop $k' \in [0, k_{max}]$, which is given by the probability that the group is pruned at hop k' (i.e., $P_1(k')$ given in Eq. (6)) times the filtering cost (i.e., $(k'+1) \cdot C$). Moreover, the second term is the expected cost that this group cannot be pruned by all the k_{max} signatures, which is given by the probability that the group cannot be pruned (i.e., k_{max}) times the filtering cost $(k_{max}+1) \cdot C$.

Computation of $P_1(k')$: We compute the probability $P_1(k')$ that the group is pruned at hop k', by considering conflict rates in label signatures of size B. Specifically, the probability that a label is mapped to a random position in the signature is given by $1/B$. Thus, with U labels hashed in the signature, the probability that a position is empty (i.e., "0") is given by $(1-\frac{1}{B})^U$; similarly, the probability that a position is set to "1" is given by: $1-(1-\frac{1}{B})^U$.

Therefore, we can collect statistics of label information in label signatures. For example, for each group $PG_{m'}(v_i)$, we can obtain the number, $U_{m'k'}$, of labels that are hashed into signature $sig_{k'}(v_i)$.

Then, we rewrite $P_1(k')$ as:

$$P_1(k') = \left(\prod_{j=0}^{k'-1}\left(1-\left(1-\frac{1}{B}\right)^{U_{m'j}}\right)\right) \cdot \left(1-\frac{1}{B}\right)^{U_{m'k'}}$$

Computation of $P_2(k_{max})$: Similar to $P_1(k')$, we can rewrite $P_2(k_{max})$ as:

$$P_2(k_{max}) = \prod_{j=0}^{k_{max}}\left(1-\left(1-\frac{1}{B}\right)^{U_{m'j}}\right)$$

The total computation cost, $Cost_{total}(v_i)$: By substituting both $P_1(k')$ and $P_2(k_{max})$ into Eq. (6), we can calculate the computation cost, $Cost_{m'}(v_i)$, for the m'-th group.

Thus, for all the m groups, the total computation cost, $Cost_{total}(v_i)$, is given by:

$$Cost_{total}(v_i) = \sum_{m'=1}^{m} Cost_{m'}(v_i) \quad (7)$$

Therefore, essentially, we want to find a good partitioning strategy over paths (starting from v_i), which minimizes the total computation cost, $Cost_{total}(v_i)$, given in Eq. (7).

Path partitioning based on cost model: Our basic idea is to partition paths into m groups, for different m values from 1 to m_{max}, and choose an m value, as well as its partitioning strategy, that achieves the minimum computation cost $Cost_{total}(v_i)$ in Eq. (7).

Due to exponential number of possible partitioning methods, we propose an approximation approach to find a good partitioning strategy for a particular m value. Specifically, we start from vertex v_i to traverse the graph, and each time we obtain a set, $Path_k(v_i)$, of paths with length k.

```
Procedure Vertex_Label_Signature_Generator {
  Input: a probabilistic RDF graph $G$, a starting vertex $v_i$, and maximum hop $k_{max}$
  Output: vertex label signature $sig(v_i)$.
  (1)   $S_0(v_i) = \{v_i\}$;
  (2)   $Path_0(v_i) = \{v_i\}$;
  (3)   encode labels $l(v_i)$ of vertex $v_i$ in signature $sig_0(v_i)$
  (4)   $k = 0$;
  (5)   while $k < k_{max}$        // for generating signature $sig_k(v_i)$
  (6)       for each path $PA \in Path_k(v_i)$
  (7)           expand path $PA$ to $PA_{new}$ by including a vertex $v_j$
  (8)           add the expanded paths $PA_{new}$ to $Path_{k+1}(v_i)$
  (9)           add vertex $v_j$ to $S_{k+1}(v_i)$
  (10)          hash labels $l(v_j)$ of vertex $v_j \in S_{k+1}(v_i)$ into signature $sig_{k+1}(v_i)$
  (10')         // adaptively hash labels into signature of the corresponding path group
  (11)      $k = k + 1$;
  (12) return $sig_0(v_i)$, $sig_1(v_i)$, ..., and $sig_{k_{max}}(v_i)$
}
```

Fig. 6. The construction of vertex label signature $sig(v_i)$.

If $|Path_k(v_i)| \le m$, we continue to expand the length of these paths. Otherwise (i.e., $|Path_k(v_i)| > m$), we will divide these paths into m groups. Initially, we first randomly select m pivot paths with their individual k-hop signatures, and then assign the remaining paths to pivots with the closest signatures under the *Hamming distance*. After the initialization, we evaluate the computation cost of such partitions based on our proposed cost model in Eq. (7). In order to obtain good results, we perform random swapping of paths among path groups, in order to obtain a better partitioning strategy with lower computation cost.

Construction of vertex label signatures: Figure 6 illustrates the pseudo-code of generating vertex label signature, namely procedure Vertex_Label_Signature_Generator, which produces label signatures for a vertex v_i. In brief, the procedure starts with vertex v_i, and maintains two sets, $Path_k(v_i)$ and $S_k(v_i)$, which are the set of paths with lengths k and the set of vertices k hops from vertex v_i (via some path), respectively. In each iteration (lines 5-11), we compute set $Path_{k+1}(v_i)$ by expanding paths in $Path_k(v_i)$, and update $S_{k+1}(v_i)$ with newly included vertices correspondingly (lines 7-9). Then, we can obtain label signature $sig_{k+1}(v_i)$ with labels from set $S_{k+1}(v_i)$ (line 10). In the case of constructing adaptive label signatures, we hash labels of vertices to the signature $sig_{k+1}^{(m')}(v_i)$ of the corresponding path group (line 10'). Finally, we return all the signatures $sig_k(v_i)$, for $0 \le k \le k_{max}$ (line 12).

Construction of edge label signatures: The construction of edge label signatures is similar to that of vertex label signatures. The algorithm of constructing edge label signature $sig(e_{rj})$ is the same as procedure Vertex_Label_Signature_Generator in Fig. 6, except that in line 10 we hash the edge label $l(e_{rj})$ of the newly included edge $e_{rj} \in PA_{new}$ into signature $sig_{k+1}(e_{rj})$, which will be returned by line 12.

4.2 Quality Score Pruning

While the adaptive label pruning method filters out those subgraphs whose labels do not match with the query graph, we next present a *quality score pruning* method, which prunes subgraph candidates g with quality scores $score(g)$ (given in Eq. (4)) less than or equal to threshold α_g.

Our basic idea of the quality score pruning is as follows. Assume that given any subgraph g, we can quickly obtain the upper bound, $ub\_score(g)$, of the quality score $score(g)$, with low cost. Then, as long as it holds that $ub\_score(g) \leq \alpha_g$, we can safely prune g. We summarize the quality score pruning in the lemma below.

Lemma 4.2: (Quality Score Pruning) For a subgraph g, let $ub\_score(g)$ be an upper bound of its quality score $score(g)$. Then, given a quality score threshold α_g, if $ub\_score(g) \leq \alpha_g$ holds, subgraph g can be safely pruned.

Proof: Derived from the inequality transition, we have $score(g) \leq ub\_score(g) \leq \alpha_g$. Thus, subgraph g cannot be the QA-gMatch result (based on Definition 2.4). □

4.2.1 Derivation of Score Upper Bound

We now derive a score upper bound $ub\_score(g)$ for a subgraph g, based on the formula of $score(g)$ in Eq. (4).

Score upper bounds of subgraphs g, $ub\_score(g)$: Given a subgraph g that is isomorphic to query graph q, we can overestimate existence probabilities, $l(v_i).p$, of its vertex labels, and compute the score upper bound $ub\_score(g)$ in the lemma below.

Lemma 4.3: (Online Upper Bound of quality score, $ub\_score(g)$) Assume that a subgraph g is structurally isomorphic to a query graph q. Then, we can obtain the score upper bound:

$$ub\_score(g) = \left(\prod_{\forall e_{ij} \in E(g)} (1 - e_{ij}.rp)\right) \cdot \left(\prod_{\forall v_i \in V(g)} l(v_i).p_{max}\right) \tag{8}$$

where $l(v_i).p_{max} = \max_{\forall l(v_i)}\{l(v_i).p\}$.

In Lemma 4.3, the score upper bound $ub\_score(g)$ uses $l(v_i).p_{max}$ to overestimate existence probabilities of possible labels. Note, however, that this bound assumes that subgraph g is isomorphic to query graph q. Since q is usually online given in practice, it is not efficient (e.g., NP-hard) to perform the isomorphism checking, before we use Lemma 4.3 to compute online score upper bound.

Therefore, we will propose an efficient pre-computation approach below to enable fast calculation of score bounds via offline pre-computed data.

Score upper bounds in candidate graphs g', $ub\_score(g)$: In particular, instead of directly computing score upper bounds for subgraphs g ($\equiv q$), we will consider some candidate graphs g' in G, and offline estimate score upper bounds for any subgraph g (with $|V(q)|$ vertices and $|E(q)|$ edges) within candidates g'.

To obtain such candidate graphs g', we can start from each vertex $v_i \in V(G)$, and traverse probabilistic RDF graphs G in a breadth-first manner. This way, we can retrieve subgraphs g', whose vertices v_j are at most k_{max} hops away from vertex v_i (via some paths). Then, we can derive an upper bound, $ub\_score(g)$, of quality score $score(g)$ below, for any subgraph $g \subseteq g'$.

Specifically, for all vertices $v_i \in V(g')$, we denote their maximum existence probabilities of labels (i.e., $l(v_i).p_{max}$ given in Lemma 4.3) as vp_1, vp_2, ..., and $vp_{|V(g')|}$. Moreover, all edges $e_{ij} \in E(g')$ are associated with repair confidences erp_1, erp_2, ..., and $erp_{|E(g')|}$.

Without loss of generality, we assume that $vp_1 \geq vp_2 \geq ... \geq vp_{|V(g')|}$, and $erp_1 \leq erp_2 \leq ... \leq erp_{|E(g')|}$. Let $|V(q)|$ and $|E(q)|$ be the numbers of vertices and edges in query graph q, respectively. We have the following lemma to derive the score upper bound $ub\_score(g)$.

Lemma 4.4: (Offline Upper Bound of quality score, $ub\_score(g)$) Given a query graph q and a candidate graph g', the upper bound, $ub\_score(g)$, of quality score for any subgraph $g \subseteq g'$ (which may match with q) is:

$$ub\_score(g) = \left(\prod_{i=1}^{|E(q)|} (1 - erp_i)\right) \cdot \left(\prod_{j=1}^{|V(q)|} vp_j\right). \tag{9}$$

In Lemma 4.4, erp_i and vp_j can be offline pre-computed within candidate graphs g', where $1 \leq i \leq |E(g')|$ and $1 \leq j \leq |V(g')|$. Then, specific for query graph q, we only use the first $|E(q)|$ smallest erp_i's and $|V(q)|$ largest vp_j's to derive the score upper bound $ub\_score(g)$ in Eq. (9).

<u>Example of deriving score upper bound:</u> *In the example of Figure 4(a), maximum existence probabilities of vertex labels are given by:* $l(v_1).p_{max} = 0.5$, $l(v_2).p_{max} = 0.7$, $l(v_3).p_{max} = 0.6$, *and* $l(v_4).p_{max} = 0.8$. *Thus, these maximum existence probabilities in decreasing order are* $vp_1 = 0.8$, $vp_2 = 0.7$, $vp_3 = 0.6$, *and* $vp_4 = 0.5$. *Further, in graph G, we can sort repair confidences in ascending order:* $erp_1 = 0.1$, $erp_2 = 0.2$, $erp_3 = 0.3$, *and* $erp_4 = 0.4$. *Thus, for query graph q in Figure 4(c) with 4 vertices and 3 edges, score upper bound $ub\_score(g)$ of any subgraph $g \subseteq G$ is:*

$$\begin{aligned} ub\_score(g) &= \prod_{i=1}^{3}(1 - erp_i) \cdot \prod_{j=1}^{4} vp_j \\ &= (1-0.1) \times (1-0.2) \times (1-0.3) \times 0.8 \times 0.7 \times 0.6 \times 0.5. \end{aligned}$$

Therefore, $ub\_score(g)$ above is the score upper bound for any subgraph $g \subseteq G$ that may match with q. ■

4.2.2 Linear Approximation of Score Upper Bounds

According to Lemma 4.4, we need to offline pre-compute and store repair confidences erp_i and existence probabilities vp_j, which requires $O(|E(g')| + |V(g')|)$ space cost. Similarly, the time complexity to online compute $ub\_score(g)$ in Eq. (9) is also $O(|E(g')| + |V(g')|)$ in the worst case. In order to reduce the space and online computation costs, we propose a linear approximation approach, which approximates (upper bounds) terms in Eq. (9) and only requires $O(1)$ space/time costs.

Rewriting score upper bounds $ub\_score(g)$ in Eq. (9): We first rewrite Eq. (9) by taking logarithms on both hand sides, and obtain:

$$ln(ub\_score(g)) = \left(\sum_{i=1}^{|E(q)|} ln(1 - erp_i)\right) \cdot \left(\sum_{j=1}^{|V(q)|} ln(vp_j)\right). \tag{10}$$

Next, we introduce two functions $UB_1(x)$ and $UB_2(x)$, that is:

$$UB_1(x) = \sum_{i=1}^{x} ln(1 - erp_i) \tag{11}$$

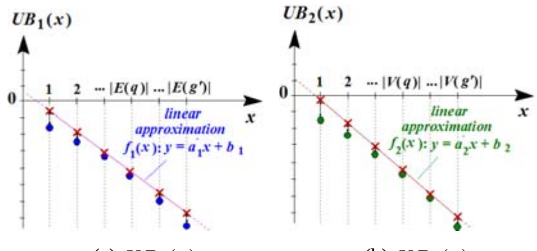
$$UB_2(x) = \sum_{j=1}^{x} ln(vp_j) \tag{12}$$

where the range of input x in $UB_1(x)$ is $[1, |E(g')|]$, and input x in $UB_2(x)$ falls into range $[1, |V(g')|]$.

Thus, by substituting $UB_1(x)$ and $UB_2(x)$ (in Eqs. (11) and (12), respectively) into Eq. (10), we have:

$$ub\_score(g) = exp(UB_1(|E(q)|) + UB_2(|V(q)|)) \tag{13}$$

where $exp(z) = e^z$.



(a) $UB_1(x)$ (b) $UB_2(x)$

Fig. 7. Linear approximations of score upper bounds.

Linear approximation of $UB_1(x)$ and $UB_2(x)$: As illustrated in Figure 7, for a candidate graph g', we plot its functions $UB_1(x)$ and $UB_2(x)$ (denoted as circular points) for all possible inputs x in 2D spaces.

As mentioned earlier, the space cost to store these circular points is high. Thus, we propose to use line functions $f_1(x) = a_1x + b_1$ and $f_2(x) = a_2x + b_2$ to approximate upper bounds of $UB_1(x)$ and $UB_2(x)$, respectively. In other words, we have $f_1(x) \geq UB_1(x)$ for all $1 \leq x \leq |E(g')|$, and $f_2(x) \geq UB_2(x)$ for all $1 \leq x \leq |V(g')|$.

Therefore, we can further rewrite Eq. (14), and obtain a score upper bound $ub\_score_{linear}(g)$ below, which is efficient to compute (i.e., $O(1)$ time complexity):

$$ub\_score_{linear}(g) = exp(f_1(|E(q)|) + f_2(|V(q)|)) \quad (14)$$

Optimal linear approximation: The only remaining issue is how to (offline) pre-compute optimal linear approximations $f_1(x)$ and $f_2(x)$ over points for $UB_1(x)$ and $UB_2(x)$, respectively. Since $UB_1(x)$ and $UB_2(x)$ are similar, we focus on $UB_1(x)$ in the discussion below.

To obtain an optimal approximation $f_1(x)$, we have two optimization goals:

- $f_1(x) \geq UB_1(x)$, and
- minimize the sum of squared distances between $f_1(x)$, and $UB_1(x)$, that is:

$$T = \sum_{x=1}^{|E(g')|} (f_1(x) - UB_1(x))^2 = \sum_{x=1}^{|E(g')|} (a_1x + b_1 - UB_1(x))^2.$$

Note that, $f_1(x)$ should be an upper bound of $UB_1(x)$ (in order to compute score upper bounds), and as close to $UB_1(x)$ as possible. As proved in [2], such an optimal linear approximation must pass through at least one anchor point A. Without loss of generality, we denote this anchor point as $A(x_A, y_A)$ for $y_A = UB_1(x_A)$ in the 2D space (e.g., Figure 7(a)). Thus, we have $f_1(x) = a_1x - a_1x_A + y_A$.

To minimize T in our goal, we let:

$$\frac{\partial T}{\partial a_1} = \frac{\partial}{\partial a_1}\left(\sum_{x=1}^{|E(g')|}(a_1x - a_1x_A + y_A - UB_1(x))^2\right) = 0,$$

from which we obtain:

$$a_1 = \frac{\sum_{x=1}^{|E(g')|}(UB_1(x) - y_A)\cdot(x - x_A)}{\sum_{x=1}^{|E(g')|}(x - x_A)^2}. \quad (15)$$

Thus, we have:

$$b_1 = -a_1x_A + y_A \quad (16)$$

Therefore, we will set each point $(x,\ UB_1(x))$ in the 2D space (as shown in Figure 7(a)) as the anchor point A, and obtain its optimal linear function by computing (a_1, b_1)-pair via Eqs. (15) and (16). Then, we select one of linear functions as $f_1(x)$ that satisfies the condition $f_1(x) \geq UB_1(x)$ and minimizes T. Thus, for the resulting $f_1(x)$ function, we only need to store 2 parameters a_1 and b_1 with $O(1)$ space cost.

Similar to $f_1(x)$, we can also compute the optimal linear approximation function $f_2(x)$ for $UB_2(x)$. The details are similar, and thus omitted.

5 Query Processing Approach

5.1 Index Construction

Pre-processing of probabilistic RDF graphs: Recall that, the QA-gMatch problem retrieves those subgraphs g that are both isomorphic to query graph q and with quality score $score(g) > \alpha_g$. In order to utilize our adaptive label pruning and quality score pruning, we start from each vertex $v_i \in V(G)$ of graph G, and extract a candidate graph $g' \subseteq G$ whose vertices are within k_{max} hops from v_i (potentially containing subgraph g). Then, we can obtain their adaptive label signatures $sig(v_i)$ and $sig^e(v_i)$, as well as numbers of vertices/edges (denoted as $cnt(v_i) = |V(g')|$ and $cnt^e(v_i) = |E(g')|$). Moreover, to derive score upper bounds (for quality score pruning), we also compute linear approximation functions (e.g., $f_1(x)$ and $f_2(x)$) for candidate graph g'.

Index structure: Our tree index $\mathcal{I}$ is exactly constructed over those pre-computed data of candidate graphs g'. Specifically, in leaf nodes, each entry corresponds to a candidate graph g' and stores its signatures, vertex/edge counters, and linear functions.

On the other hand, in non-leaf nodes, each entry N_z aggregates the information (i.e., signatures, counters, and functions) in its children. In particular, since label signatures store information of label existence, we summarize signatures by performing bit-OR operations over signatures in children of node N_z.

Signature aggregation: Specifically, we merge two adaptive label signatures, $sig(v_i)$ and $sig(v_j)$ of sizes m_1 and m_2 (w.o.l.g. $m_1 \geq m_2 \geq 1$), respectively, as follows. For each group $sig^{(m')}(v_j)$ in $sig(v_j)$ ($1 \leq m' \leq m_2$), we perform bit-OR with one of groups in $sig(v_i)$ which minimizes the *Hamming distance* (intuitively, this can reduce the conflict rate and thus improve the pruning power). As a result, for entry N_z in non-leaf nodes, we denote:

$$sig(N_z) = \vee_{\forall v_i \in N_z} sig(v_i);$$
$$sig^e(N_z) = \vee_{\forall v_i \in N_z} sig^e(v_i).$$

Counter aggregation: For numbers of vertices/edges, we obtain:

$$cnt(N_z) = \max_{\forall v_i \in N_z} cnt(v_i);$$
$$cnt^e(N_z) = \max_{\forall v_i \in N_z} cnt^e(v_i).$$

Linear function aggregation: Finally, we illustrate how to find a linear approximation over approximation line segments (for deriving score upper bounds) of multiple subgraphs under entry N_z.

Figure 8 shows 2 linear approximations, $f_1'(x)$ and $f_1''(x)$, for function $UB_1(x)$ in 2 candidate graphs g' and g'', respectively. Without loss of generality, assume that $|E(g'')| < |E(g')|$, and an index entry N_z contains both candidate graphs g' and g''. Our goal is to compute a linear approximation function $f_1(x)$ for entry N_z, which can tightly upper-bound line segments of subgraphs in N_z.

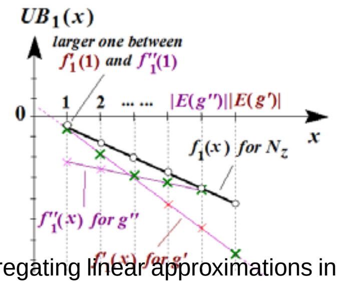


Fig. 8. Aggregating linear approximations in index $\mathcal{I}$.

To achieve this goal, as depicted in Figure 8, we first take the larger value between $f'_1(x)$ and $f''_1(x)$, for each $x \in [1, |E(g')|]$. For example, when $x = 1$, we have $f'_1(1) > f''_1(1)$, and thus take the point $(1, f'_1(1))$.

This way, we can get totally $|E(g')|$ points in the 2D space. Then, we can apply the approach in Section 4.2.2 to find optimal linear approximation function $f_1(N_z)$ over these points. Once we obtain linear function $f_1(N_z)$, we store its two coefficients $a_1(N_z)$ and $b_1(N_z)$ in entry N_z. The case of function $f_2(\cdot)$ is similar and thus omitted.

Index construction: We construct the index $\mathcal{I}$ in a bottom-up manner. Specifically, starting from leaf nodes (on level 0), we iteratively build non-leaf nodes on Level L by grouping nodes on Level $(L-1)$. We perform the grouping based on the criterion of minimizing the total summed pairwise Hamming distance among label signatures.

In particular, let F be the average fanout of the tree index. Thus, there are $M_0 = |V(G)|/F$ leaf nodes on Level 0. In a general case, on Level $(L-1)$, there are $M_{L-1} = |V(G)|/F^L$ nodes. To‘ construct Level L from Level $(L-1)$, we first randomly select $M_L = |V(G)|/F^{L+1}$ pivot nodes, and assign the remaining nodes to their closest pivots in terms of Hamming distances on their label signatures. This way, we can obtain M_L groups. After that, we randomly swap a pivot node with a non-pivot node, and evaluate the summation of pairwise Hamming distances in all M_L groups. If the swapping leads to lower Hamming distances (indicating lower conflict rate), we accept the swapping. This process repeats until a maximum number of swapping times (a system parameter) is reached. To avoid local optimum, we execute the process above multiple times with different sets of initial pivot nodes.

Pruning with index nodes: The pruning with index nodes is similar to that with subgraphs. Instead of checking pruning conditions for subgraphs, we can utilize label signatures and linear approximation functions in index entries N_z to enable the pruning. That is, for adaptive label pruning, we check the pruning conditions (given in Eq. (5)), by performing bit-OR operations over signatures $sig(q_i)$ and $sig(N_z)$ (or signatures for edges).

Similarly, for quality score pruning, we estimate the score upper bound for any subgraph under entry N_z, using functions $f_1(x)$ and $f_2(x)$ in entry N_z. If this upper bound is not greater than α_g, then we can safely prune all subgraphs under entry N_z (i.e., saving the computation cost of visiting children of this entry).

5.2 QA-gMatch Query Procedure

Figure 9 illustrates the pseudo code of the QA-gMatch processing algorithm, namely procedure QA-gMatch_Processing. We first start from the root, $root(\mathcal{I})$, of the index $\mathcal{I}$, and prune those entries N_z in the root that cannot contain candidate vertices matching with query vertex q_i. If an entry N_z cannot be pruned w.r.t. q_i, then we insert it into candidate set $cand(q_i)$ (lines 3-6).

```
Procedure QA-gMatch_Processing {
  Input: an inconsistent probabilistic RDF graph $G$, a tree index $\mathcal{I}$ over $\mathcal{G}$,
         a query graph $q$, and a score threshold $\alpha_g$
  Output: subgraphs $g$ that are QA-gMatch answers
  (1) pre-process $q$ to obtain label signatures and numbers of vertices/edges
  (2) initialize a candidate set $cand(q_i)$ for each vertex $q_i \in V(q)$
  (3) for each entry $N_z$ in $root(\mathcal{I})$
  (4)     for each $q_i$
  (5)         if $N_z$ cannot be pruned by $q_i$ via adaptive label and quality score pruning
  (6)             add $N_z$ to $cand(q_i)$
  (7) while $cand(\cdot)$ is not empty
  (8)     for each $N_z \in \bigcup_{\forall i} cand(q_i)$
  (9)         remove $N_z$ from $cand(q_i)$
  (10)        if $N_z$ is a leaf node
  (11)            for each subgraph $g'$ under $N_z$
  (12)                apply adaptive label pruning and quality score pruning to $g'$ w.r.t., $q_i$
  (13)                if $g'$ cannot be pruned
  (14)                    add $g'$ to $cand^{new}(q_i)$
  (15)        else // intermediate node
  (16)            for each child node $N_c$ under $N_z$
  (17)                apply adaptive label pruning and quality score pruning to $N_c$ w.r.t., $q_i$
  (18)                if $N_c$ cannot be pruned
  (19)                    add $N_c$ to $cand^{new}(q_i)$
  (20)    if leaf level is not encountered, then $cand(q_i) = cand^{new}(q_i)$ for all $q_i$
  (21) refine candidate graphs $g'$ by joining candidates in $cand^{new}(q_i)$ and return
       the QA-gMatch answers
}
```

Fig. 9. QA-gMatch over inconsistent probabilistic graphs.

Next, we will traverse the tree index in a breadth-first manner (lines 7-20). For any leaf or non-leaf node N_z w.r.t. q_i, we apply our proposed pruning methods to children (subgraphs g' or child nodes N_c) of N_z (lines 12 and 17). In the case where we cannot prune children, we will either add subgraphs g' (if N_z is a leaf node) to final QA-gMatch candidate set $cand^{new}(q_i)$, or insert child nodes N_c into $cand(q_i)$ for further filtering.

After the index traversal, we can obtain candidate sets, $cand^{new}(q_i)$, for each query vertex q_i. Specifically, starting from any two query points q_i and q_j that are connected in the query graph q, we can join their corresponding candidate sets, $cand^{new}(q_i)$ and $cand^{new}(q_j)$. That is, for any two vertices $v_i \in cand^{new}(q_i)$ and $v_j \in cand^{new}(q_j)$, if they are connected in graph G, then the pair (v_i, v_j) is in the join result. Each time we add a new query vertex, q_r, from q that is connected to query vertices we have scanned so far (e.g., v_i and v_j), and join its corresponding candidate set, $cand^{new}(q_r)$, with our previous (join) result. Finally, we can obtain complete subgraphs g (i.e., final joining result). After that, we will check the QA-gMatch condition (as given in Definition 2.4) to return the actual QA-gMatch answers (line 21).

Discussions on QA-gMatch over Probabilistic Graphs with Vertex/Edge Label Uncertainties. Our proposed QA-gMatch techniques can be easily extended to probabilistic graphs with vertex/edge label uncertainties. In particular, for probabilistic graphs with the edge label uncertainty, we can re-define the quality score $score(g)$ (as given in Eq. (3)), by considering existence probabilities of edge labels under possible worlds. That is, the appearance probability, $Pr\{pw(G)\}$, of each possible world (in Eq. (1)) is given by multiplying existence probabilities of vertex/edge labels.

Based on the variant of QA-gMatch problem that con-

TABLE 2
The experimental settings.

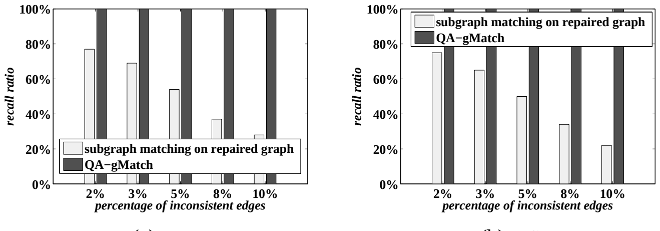
Parameters	Settings
α_g	$0.1, 0.2, \mathbf{0.5}, 0.8, 0.9$
$\lvert V(q)\rvert$	$2, 3, \mathbf{5}, 8, 10$
$N\ (= \lvert V(G)\rvert)$	$10K, 20K, \mathbf{30K}, 40K, 50K$
k_{max}	$1, 2, \mathbf{3}, 4$
m_{max}	$1, 2, \mathbf{3}, 4, 5$

siders edge label uncertainty, we can encode uncertain edge labels (rather than deterministic ones) into bit vectors (i.e., label signatures for edges). Since the adaptive label pruning method only checks the existence of vertex/edge labels via signatures, our encoding with uncertain edge labels can safely prune those false alarms (which do not contain vertex/edge labels in the query graph). Moreover, for the quality score pruning method, we can take into account edge existence probabilities during the derivation of score upper bound (i.e., Eq. (9)). Thus, our proposed pruning methods (w.r.t. vertex/edge label existence pruning, and pruning with score upper bounds by including edge label existence probabilities) would not introduce false dismissals. This way, our proposed QA-gMatch query processing algorithm can be extended to probabilistic graphs with both vertex and edge label uncertainties.

6 Experimental Study

In this section, we report the experimental results of our proposed QA-gMatch processing approaches over both real and synthetic RDF data. Specifically, for synthetic data, we generate an RDF graph G by randomly producing edges e_{ij} among vertices, such that each vertex v_i has its degree $deg(v_i)$ within the range $[1, deg_{max}(v_i)]$. Next, we generate L possible labels $l(v_i)$ ($\in [0, 300)$ by default) for each vertex v_i, and one label $l(e_{ij})$ ($\in [0, 100)$ by default) for each edge e_{ij}, where $L \in [1, 3]$ by default. Then, we also randomly assign existence probabilities $l(v_i).p \in (0, 1]$ to vertex labels $l(v_i)$, and repair confidences $e_{ij}.rp \in [0, 0.5]$ to edges e_{ij}, where $\sum_{\forall l(v_i)} l(v_i).p = 1$. We consider Uniform and Skew (with Zipf skewness 0.8) distributions for vertex labels (denoted as U and S, respectively), and Uniform and Gaussian (with mean 50 and variance 20) for edge labels (denoted as U and G, respectively). Thus, we obtain 4 types of RDF data sets: $vUeU$, $vUeG$, $vSeU$, and $vSeG$. For real data, we used "directed_CheckedFactExtractor" in YAGO [25], which contains RDF triples associated with confidences among facts (i.e., subjects/objects). For each triple (or edge e_{ij}), we let its repair confidence $e_{ij}.rp$ be $(1 - \text{confidence of edge})$. The resulting graph contains $51,875$ vertices, and each vertex v_i has two possible labels. One is specified in RDF data, and the other one is randomly selected from existing vertex labels in RDF data. We also randomly select $x\%$ of edges in G as inconsistent edges (assumed to violate a fact table).

To evaluate the QA-gMatch query performance, we extract 50 query graphs q from probabilistic RDF graph G, by starting with a random vertex $v_i \in V(G)$ and randomly walking through edges in the graph to include more $(|V(q)| - 1)$ vertices. Labels of each vertex/edge in query graphs q are also randomly chosen from the corresponding vertex/edge in G.



(a) $vUeU$ (b) $vSeU$

Fig. 10. Effectiveness evaluation between subgraph matching over X-repaired probabilistic RDF graph and QA-gMatch.

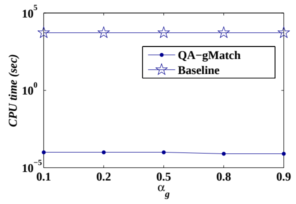
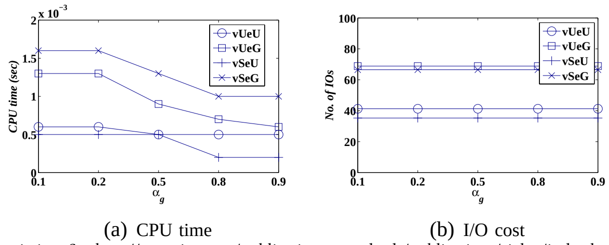


Fig. 11. QA-gMatch performance over YAGO, for different quality score thresholds α_g (QA-gMatch vs. Baseline).

We measure the QA-gMatch performance, in terms of the CPU time and the number of I/Os, where the CPU time is the time cost of filtering through the index, and the number of I/Os is the number of page accesses for QA-gMatch processing. Table 2 depicts our experimental settings, where the numbers in bold font are default values. For each set of our experiments, we will vary one parameter, while fixing others to their default values. All experiments were run on a Pentium IV 3.2GHz PC with 1G memory, and results are the average of 50 queries.

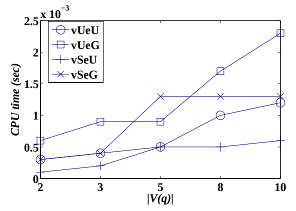
Effectiveness evaluation: We first evaluate the effectiveness of our proposed QA-gMatch approach, compared with that of conducting subgraph matching query on a repaired probabilistic RDF graph G^R (i.e., simply deleting edges that may contain inconsistencies) under X-repair semantics [7]. We vary different percentages, $x\%$, of inconsistent edges from 2% to 10%, where $\alpha_g = 0$, $|V(q)| = 10$, $k_{max} = 3$, and $N = 30K$. Figure 10 reports the *recall ratio* of both methods on $vUeU$ and $vSeU$ (results of other data sets are similar and omitted), where the recall ratio is defined as the number of actual matching subgraphs that are retrieved divided by the total number of actual subgraph answers. In the figure, due to the graph repair (via edge deletions), the subgraph matching on the repaired graph can only achieve around 23% $\sim$ 77% recall ratio, whereas QA-gMatch has 100% recall ratio (since QA-gMatch considers all possible repairs rather than a single repair), which confirms the effectiveness of our approach.

QA-gMatch performance vs. α_g: Figures 11 and 12 evaluate the performance of our QA-gMatch approaches over real (YAGO) and synthetic data, respectively, by varying α_g from 0.1 to 0.9, where other parameters are set to their default values. Specifically, on YAGO data, we compare our QA-gMatch approach with a baseline method which

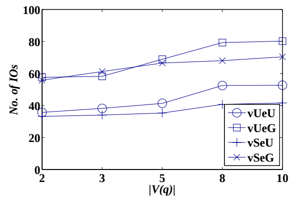


(a) CPU time (b) I/O cost

Fig. 12. Query performance vs. quality score threshold α_g.

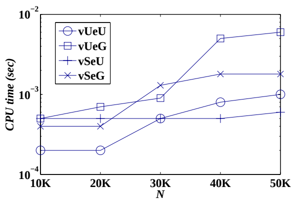


(a) CPU time

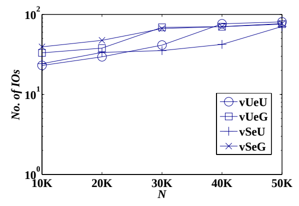


(b) I/O cost

Fig. 13. Query performance vs. query graph size $|V(q)|$.



(a) CPU time



(b) I/O cost

Fig. 14. Query performance vs. graph size N $(= |V(G)|)$.

enumerates all subgraphs g, conducts the isomorphism checking, calculates the quality score, $score(g)$ (given in Eq. (4)), and returns actual QA-gMatch answer. We can see that, in Figure 11, the CPU time of our approach outperforms the baseline method over YAGO, by about 7-8 orders of magnitude. To clearly illustrate the performance trends of our approach, in subsequent experiments, we will not report the results of the baseline method.

For both real and synthetic data, with larger α_g, the CPU time decreases. This is because, when the quality score threshold α_g increases, our quality score pruning is expected to filter out more false alarms with scores lower than α_g. Thus, the cost of accessing few candidates will incur lower CPU time. Moreover, with different α_g values, the number of I/Os remains low (i.e., 35-69), which indicates the effectiveness of our pruning methods, and the efficiency of QA-gMatch processing w.r.t. α_g.

QA-gMatch performance vs. $|V(q)|$: Figure 13 shows the effect of query graph size $|V(q)|$ on the QA-gMatch performance, where $|V(q)|$ varies from 2 to 10, and other parameters are set to their default values. For large $|V(q)|$, since we need to retrieve vertex candidates for more query nodes, the time and I/O costs are expected to increase. In figures, the increasing curves are smooth, and CPU time and I/O cost remain low (i.e., 0.1-2.3 ms and 35-80 I/Os).

QA-gMatch performance vs. N: Figure 14 demonstrates the scalability of our QA-gMatch approaches with respect to the graph size N $(= |V(G)|)$, where N varies from $10K$ to $50K$, and other parameters are by default. When N increases, both CPU time and I/O cost smoothly increase. This is reasonable, since more candidates are expected to be examined in a larger data set. Similar to previous results, the CPU time and I/O cost of QA-gMatch remain low, which indicates the good scalability of our approach against different sizes of the RDF data graph.

We tested the index construction time, cost model verification, and data sets with parameters like k_{max} and m_{max}. Please refer to Appendix B in supplementary materials.

7 Related Work

Inconsistent databases: An inconsistent database contains those data that violate some integrity constraints (e.g., key constraints, functional dependencies, etc.), rules, or facts. Previous works often considered inconsistencies in relational databases [16], [4] or probabilistic databases where tuples are associated with probabilities [22]. In contrast, our QA-gMatch problem involves inconsistent vertex labels in probabilistic graphs (rather than tuples). Thus, previous techniques cannot be directly used in our problem.

To resolve inconsistencies, there are 3 repair models [13]: X-repair [7] that allows tuple deletions only, S-repair [4] that performs both tuple insertions and deletions, and U-repair [6], [28], [8] that considers tuple value modifications. Our QA-gMatch repair model is different, in that we delete graph edges (rather than tuples in relational tables).

Different from the repair that changes data in databases, previous works also studied the consistent query answering (CQA) [4], [22] over inconsistent data, which does not update the database, but returns the aggregated query answers over (minimal or all) repaired databases. The investigated query types include relational operations (e.g., selection, projection, and join) [4], [16], [28], [3], [5] and spatial operations (e.g., range query, spatial join, and top-k) [22]. Specific pruning methods are proposed for different CQA query types to reduce the search space. In contrast, our QA-gMatch problem considers a different query type (i.e., subgraph matching) and different data model (i.e., graph data rather than relational data), which thus cannot borrow existing techniques for querying tuples or spatial objects.

RDF graph databases: RDF data can have different formats, such as triple store, column store, property tables, or graphs. In literature, Tran et al. [26] studied the keyword search query over certain RDF graph, which retrieves subgraphs that contain keywords with high ranking scores. In contrast, we consider a different subgraph matching query (instead of keyword search) over a probabilistic graph model (rather than a certain one).

Different from certain general graphs [32], inconsistent probabilistic RDF graph in our QA-gMatch problem needs to consider inconsistent/probabilistic features, and has much more possible labels (to encode) or incurs high degrees in vertices, which are thus more challenging to tackle. Moreover, there are some existing works [17], [15], [21] that model probabilistic RDF data. However, they either focused on data modeling for probabilistic RDF data [15], or considered query types over consistent graphs, other than the quality-aware subgraph matching query over inconsistent probabilistic graphs.

Yuan et al. [31], [30] considered probabilistic consistent graphs with vertex/edge uncertainties, and studied the subgraph similarity search that obtains matching subgraphs with a given query graph with high probabilities. Moustafa et al. [23] proposed a model for probabilistic entity graphs (PEGs), which incorporates identity, node attribute, and edge existence uncertainties. This model also assumes that possible worlds of PEGs are consistent, and the subgraph pattern matching is conducted over such consistent PEGs to find the matching subgraphs with high confidence. In contrast, our QA-gMatch problem models the graph by an inconsistent probabilistic graph (with vertex label uncertainties and edge repair confidences), which allows inconsistent labels in possible worlds that violate rules/facts (instead of consistent labels). Moreover, when we answer QA-gMatch queries, we need to consider resolving inconsistencies, and retrieve subgraphs with high quality scores via repairs (rather than graph existence probabilities). Thus, QA-gMatch differs from prior works on consistent probabilistic graphs, in terms of data models and query types.

Probabilistic databases: A probabilistic database [9] consists of x-tuples, and each x-tuple contains one or multiple *mutually exclusive* alternatives, associated with existence probabilities. It can represent an exponential number of possible worlds, where each possible world is a materialized instance of the database that can appear in the real world. As a consequence, the query answering over probabilistic databases is equivalent to issuing queries over all possible worlds, and aggregating the returned query answers, which is quite inefficient. Many existing works study various queries such as top-k queries [20] to improve the query efficiency by avoiding enumerating possible worlds. In contrast, our QA-gMatch query is conducted on a probabilistic RDF graph (rather than probabilistic relational tables), and thus prior techniques in probabilistic databases cannot be directly applied. This inspires us to design specific pruning techniques for inconsistent probabilistic RDF graphs.

8 Conclusions

In this paper, we study an important QA-gMatch problem, which retrieves those consistently matching subgraphs from inconsistent probabilistic data graphs with the guarantee of high quality scores. To tackle the problem, we specifically design effective pruning methods, adaptive label pruning and quality score pruning, for reducing the search space. Further, we build an effective index to facilitate the QA-gMatch processing. We conducted extensive experiments to verify the efficiency and effectiveness of our approaches.

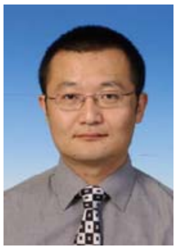
9 Acknowledgment

Lei Chen is supported in part by the Hong Kong RGC Project N HKUST637/13, National Grand Fundamental Research 973 Program of China under Grant 2014CB340303, and NSFC (Grant No. 61328202 and U1301253). Guoren Wang is supported by the NSFC (Grant No. 61332006, 61332014, 61328202 and U1401256).

References

[1] W3C: Resource description framework (RDF). In *http://www.w3.org/RDF/*.
[2] E. Achtert, C. Böhm, P. Kröger, P. Kunath, A. Pryakhin, and M. Renz. Efficient reverse k-nearest neighbor search in arbitrary metric spaces. In *SIGMOD*, 2006.
[3] P. Andritsos, A. Fuxman, and R. Miller. Clean answers over dirty databases: A probabilistic approach. In *ICDE*, 2006.
[4] M. Arenas, L. Bertossi, and J. Chomicki. Consistent query answers in inconsistent databases. In *PODS*, 1999.
[5] G. Beskales, M. A. Soliman, I. F. Ilyas, and S. Ben-David. Modeling and querying possible repairs in duplicate detection. *PVLDB*, 2(1), 2009.
[6] P. Bohannon, W. Fan, M. Flaster, and R. Rastogi. A cost-based model and effective heuristic for repairing constraints by value modification. In *SIGMOD*, 2005.
[7] J. Chomicki and J. Marcinkowski. Minimal-change integrity maintenance using tuple deletions. *Inf. Comput.*, 197(1/2), 2005.
[8] G. Cong, W. Fan, F. Geerts, X. Jia, and S. Ma. Improving data quality: consistency and accuracy. In *VLDB*, 2007.
[9] N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. *VLDB J.*, 16(4), 2007.
[10] X. L. Dong, L. Berti-Equille, and D. Srivastava. Integrating conflicting data: The role of source dependence. *PVLDB*, 2(1), 2009.
[11] X. L. Dong, A. Halevy, and C. Yu. Data integration with uncertainty. *The VLDB Journal*, 18(2), 2009.
[12] P. Exner and P. Nugues. Entity extraction: From unstructured text to dbpedia rdf triples. In *Proc. of the Web of Linked Entities Workshop*, 2012.
[13] W. Fan. Dependencies revisited for improving data quality. In *PODS*, 2008.
[14] I. Fellegi and D. Holt. A systematic approach to automatic edit and imputation. *J. American Statistical Association*, 71(353), 1976.
[15] Y. Fukushige. Representing probabilistic relations in RDF, 2005.
[16] A. Fuxman, E. Fazli, and R. Miller. ConQuer: efficient management of inconsistent databases. In *SIGMOD*, 2005.
[17] H. Huang and C. Liu. Query evaluation on probabilistic RDF databases. In *WISE*, 2009.
[18] J. Lafferty, A. McCallum, and F. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proc. of Int'l Conf. on Machine Learning (ICML)*, 2001.
[19] G. Li, B. C. Ooi, J. Feng, J. Wang, and L. Zhou. EASE: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data. In *SIGMOD*, 2008.
[20] J. Li, B. Saha, and A. Deshpande. A unified approach to ranking in probabilistic databases. *PVLDB*, 2(1), 2009.
[21] X. Lian and L. Chen. Efficient query answering in probabilistic RDF graphs. In *SIGMOD*, 2011.
[22] X. Lian, L. Chen, and S. Song. Consistent query answers in inconsistent probabilistic databases. In *SIGMOD*, 2010.
[23] W.E. Moustafa, A. Kimmig, A. Deshpande, and L. Getoor. Subgraph pattern matching over uncertain graphs with identity linkage uncertainty. In *ICDE*, 2014.
[24] L. R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. In *Proceedings of the IEEE*, volume 77, pages 257– 286, 1989.
[25] F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: A large ontology from wikipedia and wordnet. *Web Semant.*, 6(3), 2008.
[26] T. Tran, H. Wang, S. Rudolph, and P. Cimiano. Top-k exploration of query candidates for efficient keyword search on graph-shaped (RDF) data. In *ICDE*, 2009.
[27] D. Z. Wang, M. J. Franklin, M. Garofalakis, and J. M. Hellerstein. Querying probabilistic information extraction. *PVLDB*, 3(1), 2010.
[28] J. Wijsen. Database repairing using updates. *ACM Trans. Database Syst.*, 30(3), 2005.
[29] W. E. Winkler. Methods for evaluating and creating data quality. *Inf. Syst.*, 29(7), 2004.
[30] Y. Yuan, G. Wang, L. Chen, and H. Wang. Efficient subgraph similarity search on large probabilistic graph databases. In *PVLDB*, 2012.
[31] Y. Yuan, G. Wang, H. Wang, and L. Chen. Efficient subgraph search over large uncertain graphs. In *PVLDB*, 2011.
[32] P. Zhao and J. Han. On graph query optimization in large networks. *PVLDB*, 3(1), 2010.

Xiang Lian received the BS degree from the Department of Computer Science and Technology, Nanjing University, and the PhD degree in computer science from the Hong Kong University of Science and Technology. He is now an assistant professor in the Computer Science Department at the University of Texas Rio Grande Valley. His research interests include probabilistic/uncertain/inconsistent, uncertain/certain graph, time-series, and spatial databases.



Lei Chen received his BS degree in Computer Science and Engineering from Tianjin University, China, in 1994, the MA degree from Asian Institute of Technology, Thailand, in 1997, and the PhD degree in computer science from University of Waterloo, Canada, in 2005. He is now an associate professor in the Department of Computer Science and Engineering at Hong Kong University of Science and Technology. His research interests include crowdsourcing, uncertain and probabilistic databases, Web data management, multimedia and time series databases, and privacy. He is a member of the IEEE.

Guoren Wang received the BSc, MSc, and PhD degrees, in computer science, from Northeastern University, China, in 1988, 1991, and 1996, respectively. He is now a professor in the School of Information Science and Engineering, Northeastern University, China. His research interests include XML data management, query processing and optimization, high-dimensional indexing, parallel database systems, P2P data management, and uncertain data management.