Semantic SPARQL Similarity Search Over RDF Knowledge Graphs

Weiguo Zheng[1], Lei Zou[1], Wei Peng[1], Xifeng Yan[2], Shaoxu Song[3], Dongyan Zhao[1]

[1] *Peking University, Beijing, China, 100080;*
[2] *University of California at Santa Barbara, California, USA, 93106;*
[3] *Tsinghua University, Beijing, China, 100084.*
{zhengweiguo,zoulei,pengw,zhaody}@pku.edu.cn, xyan@cs.ucsb.edu, sxsong@tsinghua.edu.cn

ABSTRACT

RDF knowledge graphs have attracted increasing attentions these years. However, due to the schema-free nature of RDF data, it is very difficult for users to have full knowledge of the underlying schema. Furthermore, the same kind of information can be represented in diverse graph fragments. Hence, it is a huge challenge to formulate complex SPARQL expressions by taking the union of all possible structures.

In this paper, we propose an effective framework to access the RDF repository even if users have no full knowledge of the underlying schema. Specifically, given a SPARQL query, the system could return as more answers that match the query based on the semantic similarity as possible. Interestingly, we propose a systematic method to mine diverse semantically equivalent structure patterns. More importantly, incorporating both structural and semantic similarities we are the first to propose a novel similarity measure, *semantic graph edit distance*. In order to improve the efficiency performance, we apply the semantic summary graph to summarize the knowledge graph, which supports both high-level pruning and drill-down pruning. We also devise an effective lower bound based on the TA-style access to each of the candidate sets. Extensive experiments over real datasets confirm the effectiveness and efficiency of our approach.

1. INTRODUCTION

An RDF repository, which consists of a set of triples ⟨subject, predicate, object⟩, can be modeled as an RDF graph, where the vertices represent subjects and objects, and the labeled edges correspond to predicates. The rapidly growing RDF knowledge repositories, such as DBpedia, Yago and Freebase, increase the demand for managing graph data effectively and efficiently.

SPARQL, a structural query language proposed by W3C, is designed for querying RDF data. Since SPARQL queries can be represented as query graphs [28], a SPARQL query can be answered by performing the graph pattern matching over RDF graphs [1].

Due to the "schema-free" nature of RDF data, different data contributors may adopt different schemas to describe the same real-world fact. Thus, if we want to find more answers to a question, complex SPARQL queries that contain multiple UNION operators are required. Clearly, it is very difficult for users (even the professional users) to conceive the complicated SPARQL queries that not only conform to the syntax but also consider the flexible underlying schemas. We illustrate the challenges by the following motivating example.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org.
Proceedings of the VLDB Endowment, Vol. 9, No. 11
Copyright 2016 VLDB Endowment 2150-8097/16/07.

1.1 Motivating Example

Fig. 1 presents a piece of RDF graph extracted from DBpedia. Assume that we want to find the cars that are produced in Germany. There are at least three different German car brands, such as Porsche Cayenne, Mercedes Benz and BMWX6, which are stored in three different schemas in Fig. 1. In order to enable the query, we should issue the following SPARQL query, which is composed of three subqueries corresponding to the query graphs q_1, q_2, and q_3 in Fig. 2(a).

```
SELECT ?x WHERE {
{?x <type> Automobile. ?x <production> Germany.}
UNION
{?x <type> Automobile. ?x <assembly> Germany.}
UNION
{?x <type> Automobile. ?x <manufacturer> ?y.
 ?y <location> Germany.}}
```

Since different structural patterns may express the same semantic meanings, formulating a SPARQL query that considers all possible structures is not a trivial task. Although we can conceive the complete SPARQL query for "the cars produced in Germany", we cannot always use the same query pattern to answer other questions. For instance, in order to find all cars produced in Australia, we should add another SPARQL subquery "$?x < type > AutomobileOfAustralia$" to capture the complete answers.

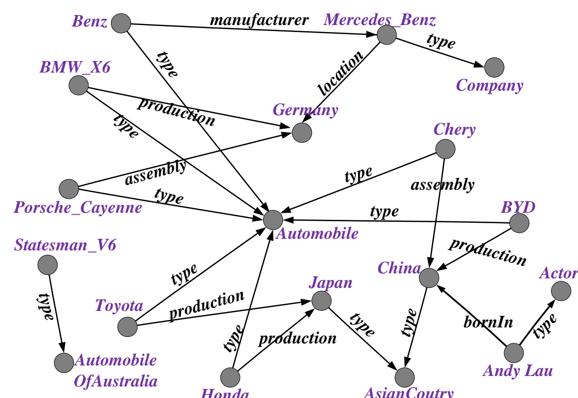
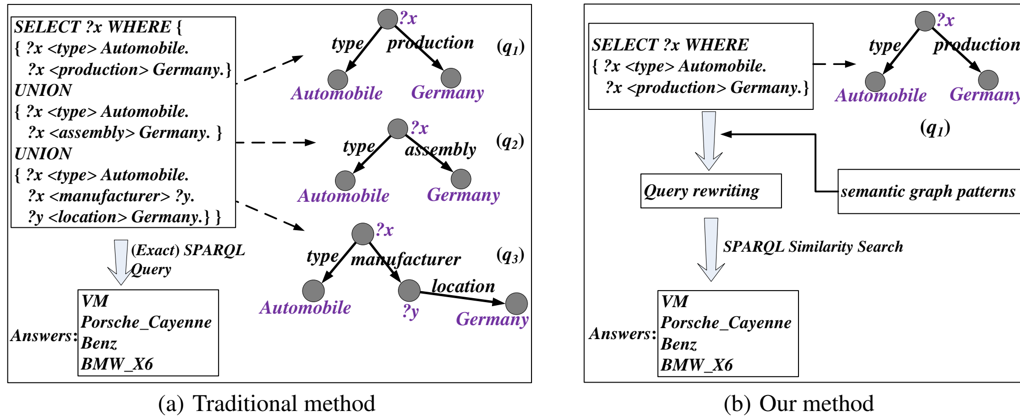


Figure 1: An RDF knowledge graph.



(a) Traditional method (b) Our method

Figure 2: Traditional method vs. our method

To obtain more correct answers, the traditional method (as shown in Fig. 2(a)) demands users to have the full knowledge about the schema of an RDF graph. In other words, it requires that users should not only know all the predicates in the knowledge base clearly, but also be aware of different structural expressions for identical semantic facts. It will be more difficult for open-domain knowledge graphs, such as DBpedia. Every coin has two sides. The "schema-free" nature of RDF facilitates the dataset construction, but it inevitably leads to the inherent difficulty of querying the knowledge base.

The goal of this paper is to provide an effective way to access the RDF repository even if one has no full knowledge of the underlying schema. To this end, we provide an effective query model. Given an RDF graph G, a user just needs to write a SPARQL query following one possible schema to express his/her query intention. The system should return as many answers that semantically match the query as possible. Fig. 2(b) illustrates our framework of SPARQL similarity search.

For example, given the SPARQL query in Fig. 2(b) (it corresponds to q_1), our system can find all cars that are produced in Germany. Graphs g_1, g_2, and g_3 in Figs. 3(a), 3(b), and 3(c) are three of the matches based on the semantic similarity.

1.2 Limitations of Existing Approaches

Although lots of efforts have been devoted to the graph similarity search [24, 8, 2, 7, 22, 23], they suffer from various drawbacks.

Resorting to Structure Similarity. Several approaches are proposed for the approximate subgraph query, but most of them focus on the structure similarity, such as SAPPER [24], *kGPM* [2] and Ness [8]. SAPPER [24] investigates the problem of approximate subgraph search allowing some edges unmatched. It does not support the vertex/edge label substitution. *kGPM* [2] proposes a graph pattern query, which allows a path to match an edge. However, it restricts that the vertex/edge labels specified in the query graph q should be exactly matched. Exploiting the neighborhood-based similarity measure, Ness [8] and NeMa [7] try to identify the top-k approximate matches of a query graph q.

Generally speaking, most of these methods concentrate on the graph structure similarity without considering the semantic similarity. However, in RDF graphs, two graph patterns may have large structural dissimilarity, such as Figs. 3(b) and 3(c), but they describe the identical semantic meaning.

Using Concept-level Similarity. In order to enable semantic queries over a knowledge graph, several recent approaches have studied the "semantic" similarity, but they only consider the "concept-level" similarity. For example, KMatch [22] introduces the ontology-based subgraph query, which computes the similarity between two vertex labels by a similarity function. However, it requires that q and its match must share the same graph structure including the edge label constraints. Recently, SLQ [23] presents a query engine that integrates a set of transformation functions, such as "synonym" and "distance". Although it can plug in the "distance" transformation (i.e., transforming an edge to a shortest path), more complicated structures (e.g., graphs) are hard to deal with. Furthermore, structural transformation (corresponding to "distance") and semantic transformation (corresponding to "ontology") are taken into consideration separately.

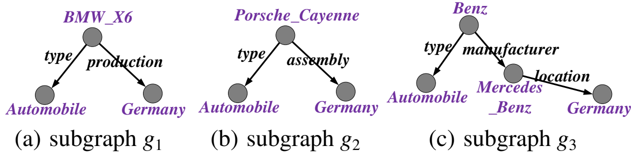
1.3 Challenges and Contributions

Challenge 1: Mining Diverse Structure Patterns with Equivalent Semantic Meanings. It is a common case that many subgraphs of a knowledge graph convey the same semantic meaning even if they do not share the identical structure. For example, graphs g_1, g_2, and g_3 in Fig. 3, are three different subgraphs extracted from the knowledge graph G in Fig. 1. Although they are different in terms of graph structures, they share the same semantic meaning, i.e., the automobiles that are produced in Germany. Note that the task of mining diverse structural patterns with equivalent semantic meanings is different from schema mapping [13] and ontology alignment [16]. (1) Different inputs: Schema mapping and ontology alignment take two schema/ontologies as inputs; but our input is a single knowledge graph. (2) Different outputs: Our task is to find sets of graph patterns that describe the same semantic meanings. However, schema mapping and ontology alignment aim to find the mapping between two elements from two schemas or two concepts from two ontologies.

In this paper, we propose an instance-driven approach to mine these semantically equivalent patterns. According to the mining results, we define three representative graph patterns of semantic equivalence, i.e., concept generation, edge redirection, and inductive inference, over the RDF knowledge graph.

Challenge 2: Measuring Semantic Similarity in a Uniform Manner. The traditional graph similarity metrics, such as graph edit distance [26] and neighborhood information [8], only measure the similarity on graph structures without considering semantic meanings. They are not satisfactory for querying RDF knowledge graphs. Some recent efforts try to take semantic meanings into consideration, but they mainly concentrate on the single concept level (i.e., only considering the similarity of two ontology concepts) [23].

This paper integrates the traditional graph structure similarity, concept-level similarity and diverse semantically equivalent structure patterns into a uniform measure, called *semantic graph edit distance* (Definition 3.2), which overcomes the limitation of the existing graph similarity metrics.



(a) subgraph g_1 (b) subgraph g_2 (c) subgraph g_3

Figure 3: Diverse semantic graph structures

Challenge 3: Improving Query Performance. Subgraph similarity search suffers from high time complexity. Thus, the time efficiency is a crucial issue especially in the scenario of RDF knowledge graphs that could be very large. Instead of enumerating all possible candidates, we design an efficient framework to deliver the top-k matches. Furthermore, in order to reduce the search space, we derive a lower bound for the semantic graph edit distance.

In summary, we make the following contributions in this paper.

- We propose an instance-driven approach to automatically discover the diverse structure patterns conveying equivalent semantic meanings from a large RDF graph.
- We formalize the problem of **S**emantic **S**PARQL **S**imilarity **S**earch (denoted by S^4) over RDF knowledge graphs. We propose a novel metric, *semantic graph edit distance*, to measure the similarity between RDF graphs, which is the first to consider the graph structure similarity, concept-level similarity and diverse semantic-equivalent structure patterns in a uniform manner.
- In order to improve the efficiency, we devise a novel index, *summary semantic graph*, which facilitates the query processing. We also derive a lower bound for the semantic graph edit distance to reduce the search space.
- Extensive experiments over real RDF repositories have demonstrated the effectiveness and efficiency of our method.

2. SEMANTIC GRAPH PATTERN

In this section, we first propose an instance-driven framework to explore the semantic graph patterns. Then we define three representative patterns according to the mining results.

DEFINITION 2.1. ***(RDF Knowledge Graph)****. A knowledge graph is a directed graph $G = (V, E, L)$, where V denotes a set of vertices (including entities, concepts, and literals); E denotes the set of edges, each of which is assigned with a label $l \in L$.*

In an RDF knowledge graph, each entity is associated with a type[1]. To evaluate the semantic relatedness, we resort to the ontology that is widely used in knowledge graphs.

DEFINITION 2.2. ***(Type/Predicate Ontology Graph)****. A type-/predicate ontology, denoted by O_C/O_P, is a directed acyclic graph describing the relations among types/predicates, where each vertex is a type/predicate, and each edge labeled with subTypeOf/subPredicateOf connects two types/predicates.*

DEFINITION 2.3. ***(Semantic Graph Pattern.)*** *In an RDF knowledge graph, a semantic graph pattern $P = \{s_1, s_2, \ldots, s_m\}$ is a set of structures that convey equivalent semantic meanings.*

Fig. 4 presents some semantic graph patterns, where the structures in the same row convey the identical semantic meaning. For instance, the two structures in the first row both describe the lakes of Denmark .

[1] If the type of an entity is unknown, we can employ the existing techniques, such as [10], to discover it.

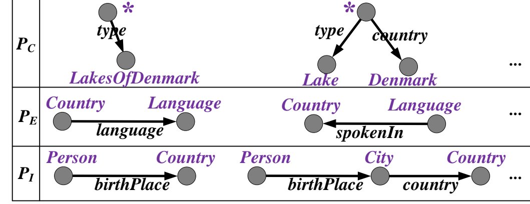


Figure 4: Semantic graph patterns.

Table 1: Dictionary of Semantic Instances

Semantic Meanings	Supporting Instances
"Lake of Denmark"	$\langle Lake_Esrum\rangle, \langle Lake_Madum\rangle \ldots$
"Automobiles of Germany"	$\langle Porsche_Cayenne\rangle, \langle Benz\rangle, \langle BMWX6\rangle, \langle VM\rangle \ldots$
"language spoken in a country"	$\langle Turkmenistan, Turkmen\rangle, \langle Estonia, Finnish\rangle \ldots$
"Person who was born in Austria"	$\langle Schrödinger, Austria\rangle, \langle Boltzmann, Austria\rangle \ldots$
...	...

2.1 Instance-driven Mining of Semantic Graph Patterns

To enable the mining of semantic graph patterns, we build a *dictionary of semantic instances D*, which records the semantically equivalent instances.

DEFINITION 2.4. ***(Dictionary of Semantic Instances.)*** *A dictionary of semantic instances is a table, where each row is a set of meaning-equivalent entities or entity pairs.*

Table 1 shows a dictionary of semantic instances, where Benz, Porsche_Cayenne, and BMWX6 are all automobiles of Germany.

Actually, lots of NLP literatures [3, 11, 21] can be adopted to extract the meaning-equivalent instances. For example, Patty [11] uses dependency relations between words to find the instance pairs that express the same relation phrase. Regarding the meaning-equivalent entities, we can resort to the existing knowledge base, such as Probase [21]. Probase contains 4.5 million *isA* pairs harvested automatically from web documents. In this work, we assume that the meaning-equivalent instances (including entities and entity pairs) are given.

Our goal is to identify semantic graph patterns (as shown in Fig. 4) according to the meaning-equivalent instances. Since there are two kinds of instances, i.e., single entities and entity pairs, we devise two algorithms to deal with these two cases.

2.1.1 Single Entity Based Semantic Graph Patterns

Let T_i denote a set of meaning-equivalent instances $\{v_i^1, \ldots, v_i^m\}$, where each instance is a single entity as shown in the first two rows of Table 1. Given a family of sets T, denoted by $D_S = \{T_1, \ldots, T_n\}$, our task is to mine the corresponding semantic graph patterns based on D_S. Algorithm 1 gives a formal description of the process.

We find the type t_j for each entity $v_i^j \in T_i$ $(1 \le j \le m)$ in the RDF graph G. Assume t_k is the type whose depth (i.e., the shortest path distance from the root type "Thing" to t_k) is the largest in the type ontology (lines 2-3 in Algorithm 1). Then the pattern $(*, type, t_k)$ is added into the semantic graph pattern P_i corresponding to T_i, where "*" represents a wildcard vertex (line 4 in Algorithm 1). For each entity v_i^j of type t_j $(\neq t_k)$, we check the neighbors L of v_i^j in the RDF graph G. If the similarity between $(L + t_j)$ and t_k is no less than a threshold, i.e., $sim(L + t_j, t_k) \ge \theta$, we can add $\{(*, r_1, l_1), \ldots, (*, r_x, l_x), (*, type, t_j)\}$ into P_i, where $L + t_j$ represents the concatenation of strings L and t_j, and $sim(\cdot, \cdot)$ can be computed based on the string edit distance (lines 5-12 in Algorithm 1). Since t_k is not too long, we can try all possible combinations of x neighbors to form L. Note that the $*$ in $\{(*, r_1, l_1), \ldots, (*, r_x, l_x), (*, type, t_j)\}$ refers to the same entity.

Algorithm 1 GSP_SingleEntity(G, D_S, O_C)

Input: The RDF knowledge graph G, the dictionary D_S consisting of meaning-equivalent single entities, O_C;
Output: The semantic graph patterns P based on D_S.

1: **for** Each set $T_i = \{v_i^1, \ldots, v_i^m\} \in D_S$ **do**
2: Find the type t_j for each vertex v_i^j
3: Let t_k is the type whose depth in O_C is the largest
4: $P_i \leftarrow (*, type, t_k)$
5: **for** Each entity $v_i^j \in T_i$ **do**
6: **if** $t_j \neq t_k$ **then**
7: **for** $1 \leq x \leq |N(v_i^j)|$ **do**
8: **for** Each x neighbors of v_i^j **do**
9: **for** Each possible combination L **do**
10: **if** $sim(L + t_j, t_k) \geq \theta$ **then**
11: $P_i \leftarrow \{(*, r_1, l_1), \ldots, (*, r_x, l_x), (*, type, t_j)\} \cup P_i$
12: $P \leftarrow P \cup P_i$
13: **return** P

EXAMPLE 1. *Let us consider the supporting instances for "Lake of Denmark" in Table 1. The types of "Lake_Esrum" and "Lake_Madum" are Lake and LakesOfDenmark, respectively. We find their structures in the RDF graph. Since the neighborhood "Denmark" together with "Lake" is similar to "LakeOfDenmark", we can generate the semantic graph patterns as shown in the first row of Fig. 4.*

2.1.2 Entity Pairs Based Semantic Graph Patterns

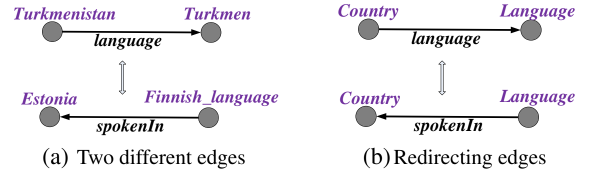
Let T_i denote a set of meaning-equivalent instances $\{\langle u_i^1, v_i^1\rangle, \ldots, \langle u_i^m, v_i^m\rangle\}$, where each instance is a pair of entities as shown in the third and fourth rows of Table 1. Given a family of sets T, denoted by $D_P = \{T_1, \ldots, T_n\}$, the goal is to mine semantic graph patterns based on D_P. The mining process is presented in Algorithm 2.

As shown in Algorithm 2, we first enumerate all simple paths $PS(u_i^j, v_i^j)$ between u_i^j and v_i^j in the RDF graph G. For efficiency considerations, we only find simple paths of length less than a threshold. Then we replace each entity in $path(u_i^j, v_i^j) \in PS(u_i^j, v_i^j)$ with its corresponding type to obtain $path^T(t_u, t_v)$, where t_u and t_v are the types of u_i^j and v_i^j, respectively. For each $path^T(t_u, t_v)$, we compute the number of entity pairs in the supporting instances that can be linked by such a path pattern $path^T(t_u, t_v)$, denoted as $Sup(path^T(t_u, t_v))$. The type path $path^T(t_u, t_v)$ with the largest $Sup(path^T(t_u, t_v))$ is added into the semantic graph pattern P_i.

Algorithm 2 GSP_EntityPairs(G, D_P, O_C)

Input: The RDF knowledge graph G, the dictionary D_P consisting of meaning-equivalent entity pairs, O_C;
Output: The semantic graph patterns P based on D_P.

1: **for** Each set $T_i = \{\langle u_i^1, v_i^1\rangle, \ldots, \langle u_i^m, v_i^m\rangle\} \in D_P$ **do**
2: **for** Each entity pair $\langle u_i^j, v_i^j\rangle \in T_i$ **do**
3: $PS(u_i^j, v_i^j) \leftarrow$ Find all simple paths (with length less than a predefined threshold) between u_i^j and v_i^j
4: **for** Each path $path(u_i^j, v_i^j) \in PS(u_i^j, v_i^j)$ **do**
5: $path^T(t_u, t_v) \leftarrow$ Replace each entity in $path(u_j^i, v_j^i)$ with its corresponding type
6: Compute $Sup(path^T(t_u, t_v))$
7: Add $path^T(t_u, t_v)$ with the largest $Sup(path^T(t_u, t_v))$ into P_i
8: $P \leftarrow P \cup P_i$
9: **return** P



(a) Two different edges (b) Redirecting edges

Figure 5: Redirecting edges.

EXAMPLE 2. *Let us consider the supporting instances for "Person who was born in Austria" in Table 1. The entity pair $\langle$chrödinger, Austria$\rangle$ is the first to be dealt with. We enumerate all simple paths between chrödinger and Austria, and generate the corresponding type paths. Since the type path $\langle Person, birthPlace, Country\rangle$ has the largest $Sup(\cdot)$, it is added into P_i. Similarly, we find the type path $\langle Person, birthPlace, City, country, Country\rangle$ for the second entity pair $\langle Boltzmann, Austria\rangle$. This type path is also added into P_i as shown in the third row of Fig. 4.*

2.2 Representative Semantic Graph Patterns

We categorize the semantic graph patterns mined in the previous subsection into three representative semantic graph patterns.

2.2.1 Concept Generalization

Considering the two graph patterns in the first row of Fig. 4, we find that the entity in the right graph is a lake with the constraint of Country (Denmark). In comparison, the entity in the left graph is a "LakesOfDenmark", which is a subconcept of "Lake". Actually, this case can be called "concept generalization". That is, a concept statement with some constraints is semantically equivalent to one of its subconcept statement. The set of patterns that satisfy the concept generalization constraint is denoted by P_C.

2.2.2 Edge Redirection

In most languages, it is very common that a fact can be described using both active voice and passive voice. This phenomenon also exists in knowledge graphs. Specifically, there are two different edges sharing the same vertex types (i.e., concepts) but having inverse directions. Let P_E denote the set of semantic graph patterns that satisfy the edge redirection constraint.

For example, Fig. 5(a) presents two different edges. Although they are different from each other, they express the same semantic meaning (corresponding to the second row of Fig. 4).

2.2.3 Inductive Inference

Another interesting phenomenon on knowledge graphs is inductive inference. Consider the semantic graph patterns in the third row of Fig. 4. Though the two graphs have different structures, they convey the equivalent semantic meaning that "the person who was born in Austria". Hence, we can find that two entities can be linked by means of an directed edge or a semantically equivalent path. The set of semantic graph patterns satisfying the inductive inference is denoted by P_I.

3. PROBLEM FORMALIZATION

Based on the semantic graph patterns, we propose a novel definition, called *semantic graph edit distance* (Sections 3.1 and 3.2), and formalize the problem studied in this paper (Section 3.3).

3.1 Semantic Graph Edit Operation

In the context of knowledge graphs, semantics should be considered together with the structure similarity. Thus, we define nine primitive semantic graph edit operations.

Operation 1. (**Semantic Vertex Insertion**). Inserting a vertex of type t at some cost $c_{vi}(t)$, where $c_{vi}(t)$ can be computed by $dist(t, t_0)$ (defined by Equation 1), i.e., the semantic distance between t and the root type t_0 in the ontology.

For any two ontological types t_1 and t_2, their semantic distance can be computed by the upward cotopic distance [15]. The intuition is that two types are similar to each other if they have more common supertypes. Formally, it is defined as:

$$dist(t_1, t_2) = 1 - \frac{|S(t_1, O) \bigcap S(t_2, O)|}{|S(t_1, O) \bigcup S(t_2, O)|} \quad (1)$$

where $S(t_i, O)$ is the set of supertypes of t_i in O.

Analogously, we can also define the semantic distance between two predicates r_1 and r_2, denoted as $dist(r_1, r_2)$.

Operation 2. (**Semantic Vertex Deletion**). Deleting a vertex of type t at some cost $c_{vd}(t)$, where $c_{vd}(t)$ can be computed by $dist(t, t_0)$ (defined by Equation 1).

Operation 3. (**Semantic Vertex Substitution**). Replacing a vertex type t with another type t' at some cost $c_{vs}(t, t')$, where $c_{vs}(t, t')$ can be computed by $dist(t, t')$.

Operation 4. (**Semantic Edge Insertion**). Inserting an edge of label (predicate) r at some cost $c_{ei}(r)$, where $c_{ei}(r)$ can be computed by $dist(r, r_0)$, i.e., the semantic distance between r and the root predicate r_0 in the predicate ontology.

Operation 5. (**Semantic Edge Deletion**). Deleting an edge of label (predicate) r at some cost $c_{ed}(r)$, where $c_{ed}(r)$ can be computed by $dist(r, r_0)$, i.e., the semantic distance between r and the root predicate r_0 in the predicate ontology.

Operation 6. (**Semantic Edge Substitution**). Replacing an edge label (predicate) r with label (predicate) r' at some cost $c_{es}(r, r')$, where $c_{es}(r, r')$ can be computed by $dist(r, r')$.

Note that the costs of traditional graph edit operations are identical (e.g., $c_{vi} = c_{vd} = c_{vs} = c_{ei} = c_{ed} = c_{es} = 1$) in the literature [25, 26, 27]. Different from that, the semantic graph edit operations are associated with semantic costs, and these costs may be different from each other.

Above, we only consider the simple vertex/edge insertion/deletion/substitution. As analyzed in Section 2, there are many diverse structures conveying the same semantic meaning. To cover these three graph patterns, we introduce three short-cut operations below.

Operation 7. (**Semantic Edge Redirection**.) Change the direction of an edge, and substitute the edge label r with r' following the patterns mined in P_E, i.e., $(v_1, r, v_2) \longleftrightarrow (v_1, r', v_2)$.

For instance, the edge $(person_1, influenced, person_2)$ is equivalent to $(person_2, influencedBy, person_1)$.

DEFINITION 3.1. ***(Star.)*** *A star rooted at vertex v, denoted as s_v, consists of vertex v and some adjacent vertices and edges of v.*

Operation 8. (**Semantic Star Substitution**.) Replacing a star s_v by an edge $(v, type, t)$ following the patterns mined in P_C.

EXAMPLE 3. *By applying the semantic star substitution, we can substitute the left graph in Fig. 6 with the edge (Statesman_V6, type, AutomobileOfAustralia) in a semantically equivalent way.*

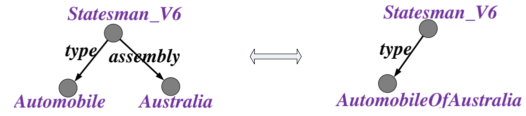


Figure 6: Semantic star substitution.

Operation 9. (**Semantic Path Substitution**). Replacing a path p by an edge e following the previously mined patterns in P_I.

EXAMPLE 4. *As shown in Fig. 3(c), since the path starting from the vertex "Benz" to the vertex "Germany" in g_3 can be substituted by an edge $(?x, production, Germany)$ in the query graph q_1 of Fig. 2(b), g_3 is a good match for q_1.*

The structure patterns P_E, P_C and P_I constitute a dictionary of short-cut patterns, denoted as P_D. Since the short-cut operations are introduced to bridge the gap between semantics and structures, their semantic costs are 0.

3.2 Semantic Graph Edit Distance

Based on the semantic graph edit operations above, we can define the semantic graph edit distance as follows.

DEFINITION 3.2. *(**Semantic Graph Edit Distance**.) Given two graphs g_1 and g_2, their semantic graph edit distance, denoted by $sged(g_1, g_2)$, is the minimum cost required to transform g_1 to g_2 by applying semantic graph edit operations*[2].

THEOREM 3.1. *Given two graphs g_1 and g_2, computing the semantic graph edit distance between g_1 and g_2 is NP-hard.*

PROOF. The proof is achieved by reducing the traditional graph edit distance (GED) problem to the semantic graph edit distance problem. Note that the three short-cut operations, i.e., semantic edge redirection, semantic star substitution, and semantic path substitution, are allowed only if the dictionary P_D contains the corresponding graph patterns. For any two graphs g_1 and g_2, we can always construct a dictionary, in which any graph patterns cannot be used for g_1 and g_2. It indicates that the short-cut operations do not apply in the computation of $sged(g_1, g_2)$. Hence, computing $ged(g_1, g_2)$ equals computing $sged(g_1, g_2)$ in this case. Since the reduction is polynomial and computing GED is a well-known NP-hard problem, the problem of computing the semantic graph edit distance is NP-hard as well. □

Remark. Since "semantics" is not a well-defined notion, it is hard to exhaustively collect all possible shortcuts. In this paper, we try to reduce the gap between semantics and graph structure by the proposed semantic graph edit operations.

3.3 Semantic Similarity Search

In this paper, we investigate semantic queries over RDF knowledge graphs using the semantic graph edit distance.

Problem Statement 1. *Given a knowledge graph G and a query graph q*[3]*, our goal is to deliver k subgraphs of G, denoted by $A = \{g_1, g_2, \ldots, g_k\}$, such that for $\forall g \in G \wedge g \notin A$, it holds that $sged(q, g) \geq sged(q, g_i)$, where $1 \leq i \leq k$.*

A straightforward method is to enumerate all subgraphs g in G, and then compute the semantic edit distance between g and q. Finally, we sort the candidates according to $sged(q, g)$ and return the top-k subgraphs with the smallest $sged(q, g)$.

[2]The short-cut operations can be only applicable to the structures in g_1 that satisfy the patterns in P_D.

[3]We only deal with the basic graph patterns of SPARQL queries.

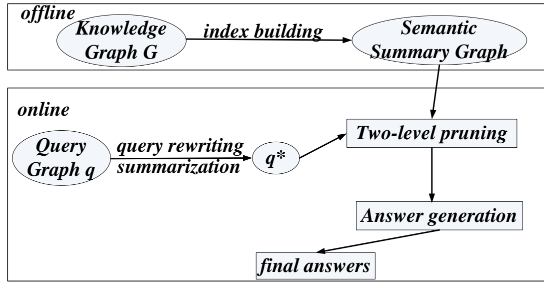


Figure 7: **Framework of Our Approach.**

Clearly, the naive approach above is time-consuming since the search space is too large. Hence, it is desired to devise an efficient solution. As depicted in Fig. 7, our approach contains two phases.

Offline Phase. In the offline phase, we build an index, i.e., the semantic summary graph, for the knowledge graph G (in Section 4). Actually, the semantic summary graph is a hierarchical clustering of the semantic facts (please refer to Definition 4.1) in G.

Online Phase. Regarding the query graph q, we first exploit the same method to summarize and rewrite q to obtain q^*. Then we perform a two-level pruning over the semantic summary graph (Section 5). Finally, we generate answers according to the candidates (Section 6).

4. SEMANTIC SUMMARY GRAPH

In this section, we propose an effective index to reduce the space cost and facilitate the query processing.

4.1 Semantic Fact and Semantic Graph

DEFINITION 4.1. *(**Semantic Fact**.) Consider an edge (v_1, r, v_2) from vertex v_1 to vertex v_2 in G, where r is the predicate. If v_1 (resp. v_2) has a type t_1 (resp. t_2) in the type ontology, we use t_1 and t_2 to represent vertices v_1 and v_2. Thus, we can obtain the corresponding semantic fact, $f = (t_1, r, t_2)$.*

Based on the semantic facts derived from the knowledge graph G, we can construct a semantic graph.

DEFINITION 4.2. *(**Semantic Graph**.) The semantic graph of a knowledge graph G, denoted by SG, consists of all semantic facts derived from G, where vertices and edges correspond to the types of entities and the semantic facts, respectively.*

EXAMPLE 5. *As shown in Fig. 8, G^2 represents a semantic graph, where each edge is a semantic fact. For example, (Company, location, City) is a semantic fact derived from graph G^1, where G^1 is actually the knowledge graph.*

4.2 Semantic Summary Graph

Relying on the type and predicate ontologies, we can reduce the space cost further.

DEFINITION 4.3. *(**Abstract Semantic Fact**.) We say (t'_1, r', t'_2) is the abstract semantic fact of (t_1, r, t_2) in a semantic graph SG if types t'_1 and t'_2 are the parents of types t_1 and t_2 in O_C, respectively, and the predicate r' is a parent of r in O_P.*

If e' is a (abstract) semantic fact of edge e, e is a **precedent** of e'. We also say that e is **covered** by e'.

EXAMPLE 6. *Consider (Politician, birthPlace, AsianCountry) in Layer2 of Fig. 8. Replacing "Politician" and "AsianCountry" with "People" and "Country", we obtain the abstract semantic fact (People, birthPlace, Country).*

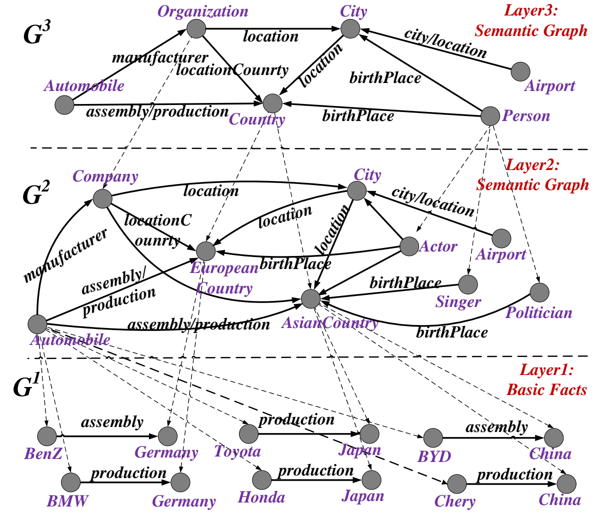


Figure 8: **Semantic Summary Graph.**

DEFINITION 4.4. *(**Abstract Semantic Graph**.) The vertices and edges of the abstract semantic graph, correspond to the parent types and abstract semantic facts of the vertices and semantic facts in a semantic graph SG[4], respectively.*

We can recursively define the abstract semantic fact for (t'_1, r', t'_2) if only t'_1, t'_2, or r' has a parent in the ontology. Therefore, we can build more semantic graphs (e.g., Layer2 and Layer3 in Fig. 8) with these abstract semantic facts.

DEFINITION 4.5. *(**Semantic Summary Graph**.) A semantic summary graph, denoted by G^S, is a multi-layer graph, where*

- *the 1st layer, G^1, consists of basic facts, i.e., edges in the knowledge graph G (G^1 is identical to G);*
- *the m-th ($m \geq 2$) layer, G^m, is a semantic graph summarized from G^{m-1} in the $(m-1)$-th layer by enumerating all (abstract) semantic facts in G^{m-1}.*

Fig. 8 depicts a semantic summary graph. Let θ denote summarization ratio (defined as $\theta = \frac{|G^{m+1}|}{|G^m|}$, where $|G|$ is the sum of vertex and edge numbers). We can control the summarizing process using θ. For example, if $\theta > 80\%$, it indicates that most vertices and edges do not change any more. So the summarizing process can stop. We will study the effect of θ empirically in Section 7.

Actually, we only need to maintain the semantic summary graph (excluding the 1st layer knowledge graph) as our index. Hence, it saves much space, which is confirmed by our experiments. More importantly, the semantic summary graph facilitates the query processing in the online phase.

Remark. Although some similar efforts can be found for keyword search [20, 9], our proposed semantic summary graph differs from them: (1) Le et al., split the RDF graph into smaller partitions and then identify a set of templates serving as summary of these partitions [9]. (2) The graph schema index in [20] does not consider multi-level summarization.

5. QUERY REWRITING AND PRUNING

Before generating final answers, we first perform the query rewriting and candidate generation. To reduce the search space, we propose a two-level pruning strategy.

[4]Since an abstract semantic graph is also a semantic graph in actual, we do not distinguish them.

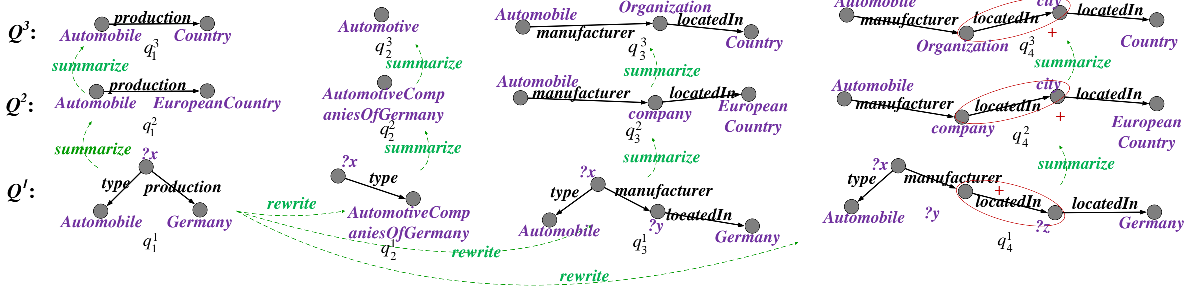


Figure 9: Query rewriting and summarization.

5.1 Query Rewriting

The objective is to obtain a set of semantically equivalent queries for the given query graph q. Hence, we rewrite a query graph using the semantic star substitution and semantic path substitution.

Star Rewriting. We first detect the stars in the query graph q that could be substituted, and then conduct the semantic star substitution to obtain a set of rewritten queries, denoted by Q.

EXAMPLE 7. *As shown in Fig. 9, graph q_1^1 is actually the query q. Graph q_2^1 is the star rewriting result for q.*

Path Rewriting. For each rewritten query graph $q_i^1 \in Q$. We check whether there exists a path that can be substituted by a shortcut edge, i.e., the semantic path substitution.

We introduce a special edge, called wildcard edge, which can match an edge or a path. After the star and path rewritings, we obtain a set of rewritten queries Q^1.

EXAMPLE 8. *Graphs q_3^1 and q_4^1 in Fig. 9 are path rewriting results. The edge circled by an ellipse in q_4^1 is a wildcard edge that represents an edge or a path. Note that the newly introduced variables $?y$ and $?z$ are wildcard vertices.*

Query Summarization. Similar to the construction of the semantic summary graph for the knowledge graph G, we can also summarize the rewritten queries "Q" using the same method. Thus, we can get the summarized query graphs, denoted by $Q^S = \{Q^1, ..., Q^h\}$, where h is the height of the semantic summary graph for query graphs. Each query graph in the m-th layer is denoted as q_i^m

EXAMPLE 9. *Fig. 9 presents the query rewriting and summarization. After the query rewriting, graphs q_1^1, q_2^1, q_3^1, and q_4^1 constitute the rewritten set Q^1. Correspondingly, graphs q_1^i, q_2^i, q_3^i, and q_4^i form the summarized rewritten queries Q^i, where $2 \le i \le h$.*

5.2 Two-level Pruning

In order to retrieve the final answers, we need to obtain candidates for each edge. In this subsection, we propose a two-level pruning strategy to compute candidates.

High-level Pruning. We first search the summarized query graphs Q^S over the semantic summary graph G^S. The principle is that if the subgraph g^s in G^S is not similar to $q^i \in Q^S$, the subgraphs g in G that are covered by g^s are probably not similar to the query q.

Considering a summarized query $q^i \in Q^S$, we try to obtain the candidates for edges in q^i. Assume u_1u_2 is an edge in q^i. We first compute the candidates for u_1 and u_2.

Let $C(u)$ denote the candidates of vertex u in q^i. We can compute candidates for the edge $e = u_1u_2 \in q^i$ using $C(u_1)$ and $C(u_2)$. Regarding each vertex pair (v_1, v_2), where $v_1 \in C(u_1)$ and $v_2 \in C(u_2)$, we check whether there exists an edge v_1v_2 or v_2v_1 that is semantically equivalent to e. Then we can obtain a set of candidate edges for each edge $e \in q^i$, denoted by $C(e)$.

EXAMPLE 10. *Let us consider the semantic graph G^3 in Fig. 8 and the summarized query graph q_1^3 in Fig. 9. The edge (Automobile, production, Country) in q_1^3 matches the edge (Automobile, assembly/production, Country) in G^3.*

Drill-down Pruning. Since the rewritten queries are summarized upward layer by layer, the summarized graphs in the lower layer convey more specific semantic meanings. Therefore, we can refine candidates by going down along with the multi-layer summarized query graphs. Algorithm 3 describes the key steps.

We first construct a set of temporary candidates $T(e_i)$ based on $C(e_i)$ (lines 3-5), where $cov(e)/cov(q)$ denotes the set of edges/queries covered by e/q. Then we consider the query graphs $cov(q^k)$ that are covered by q^k (lines 6-9). For each edge $e_i' \in q^{k-1}$, we compute its candidates based on $T(e_i)$. Note that we do not need to materialize all candidates for an edge. Therefore, we select k edges that are most similar to e_i' as candidates.

Algorithm 3 Drill-downPruning(G^S, Q^S, q^k, C, k)

Input: The semantic summary graph G^S for G, the summarized rewritten queries Q^S, the query graph q^k in the k-th layer of Q^S, the candidate edges $C = \{C(e_1), ..., C(e_m)\}$ for the edges $\{e_1, ..., e_m\}$ in q^k;

Output: The candidates for edges in each query graph q that are covered by q^k.

1: **if** $k = 1$ **then**
2:    **return** C
3: $T(e_i), ..., T(e_m) \leftarrow \phi$
4: **for** each edge $e \in C(e_i)$ **do**
5:    $T(e_i) \leftarrow T(e_i) \cup \{e' | e' \in G^{k-1} \wedge e' \in cov(e)\}$
6: **for** each $q^{k-1} \in cov(q^k)$ **do**
7:    **for** each edge $e_i' \in q^{k-1}$ **do**
8:       $C(e_i') \leftarrow$ the first k edges that are similar to e_i'
9:    Drill-downPruning(G^S, Q^S, q^{k-1}, C, $k-1$)

EXAMPLE 11. *Let us consider the edge e = (Organization, locatedIn, Country) $\in q_3^3$ in Fig. 9. Its candidate set is $C(e)$ = {(Organization, locationCountry, Country)}. When we go over the summarized query q_3^2, the precedent edge of e is e' = (Company, locatedIn, EuropeanCountry). The temporary candidate is $T(e) = \{e_1, e_2\}$, where e_1 = (Company, locationCountry, EuropeanCountry), and e_2 = (Company, locationCountry, AsianCountry). Since e_1 is more similar to e' than e_2 in terms of semantic meanings. Hence, the candidate of e' is $C(e') = \{e_1\}$.*

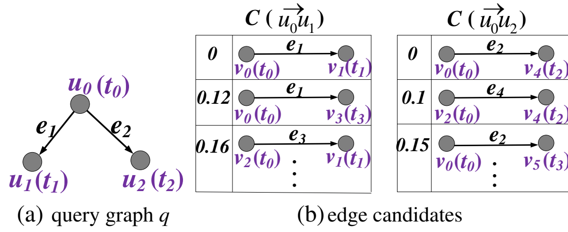


Figure 10: A query graph and its edge candidates .

6. ANSWER GENERATION

We can generate the top-k answers for q by employing a *TA-style* exploration over these candidates for each edge in q.

6.1 Bipartite-graph Based Lower Bound

Before presenting the process of answer generation, we devise a bipartite-graph based lower bound to reduce the search space.

Let $C(e)$ denote the candidates of edge e. We first sort these candidates in non-descending order according to their semantic distances to e. Given two edges $e_1 = u_1u_2$ and $e_2 = v_1v_2$, their semantic graph edit distance is computed by the following equation,

$$sged(e_1, e_2) = dist(L(e_1), L(e_2)) + \sum dist(L(u_i), L(v_i)) \quad (2)$$

where $L(e)$ denotes the predicate of edge e and $L(u)$ denotes the type of vertex u.

Let us consider a simple case, i.e., there are only two edges in q as shown in Fig. 10(a), where the characters beside vertices are vertex IDs and the characters in brackets are vertex types. Fig. 10(b) presents two sorted (in non-descending order) candidate lists, $C(u_0u_1)$ and $C(u_0u_2)$, for edges u_0u_1 and u_0u_2, respectively. The left column of each list presents the semantic distance between a query edge and the corresponding candidate edge.

High-level idea of *TA-style* exploration based pruning: Traversing each candidate list from top to bottom to construct a candidate graphs g^*. Instead of computing the exact $sged(q, g^*)$, we first compute a lower bound for $sged(q, g^*)$, based on which we determine whether the exploration continues or not. If the lower bound is larger than the minimum $sged(q, g)$, g^* can be screened out.

Given two graphs g_1 and g_2, a bipartite graph can be constructed using two edge sets, E_1 and E_2, which are obtained from g_1 and g_2, respectively. There is a weighted edge (given by $sged(e_1, e_2)$) between each $e_i \in E_1$ and $e_j \in E_2$.

We define a mapping distance between two graphs according to the mapping constructed above.

DEFINITION 6.1. (***Mapping distance***.) *Given two graphs g_1 and g_2, their mapping distance, denoted by $md(g_1, g_2)$, is defined as $md(g_1, g_2) = min \sum sged(e_i, \lambda(e_i))$, where $e_i \in g_1$, $\lambda(e_i) \in g_2$, and $\lambda(*)$ is a bijective mapping function.*

THEOREM 6.1. *Given two graphs g_1 and g_2, Equation 3 gives a lower bound of their semantic graph edit distance.*

$$sged(g_1, g_2) \geq \frac{md(g_1, g_2)}{d} \quad (3)$$

where d is the largest vertex degree of g_1 and g_2.

PROOF. Let $P = (p_1, p_2, ..., p_k)$ be an optimal alignment transforming from g_1 to g_2, i.e., $sged(g_1, g_2) = \sum cost(p_i)$, where $cost(p_i)$ is the cost of operation p_i. Accordingly, there is sequence of graph $g_1 = g_1^0 \rightarrow g_1^1 \rightarrow ... \rightarrow g_1^k = g_2$, where $g_1^i \rightarrow g_1^{i+1}$ indicates transforming g_1^i to g_1^{i+1} by operation p_i. Since the short-cut operations have been considered in the query rewriting, we can only consider the first six operations. Assume that there are k_1 edge insertion/deletion/relabeling operations, k_2 vertex insertion/deletion/relabeling operations in P.

Edge Operations (Insertion/Deletion/Relabeling): If an edge is inserted or deleted or relabeled over graph g_1^i, only one edge is affected. Thus, it holds that $md(g_1^i, g_1^{i+1}) \leq cost(p_i^e) \leq 1$ in the case of performing one edge operation over g_1^i.

Vertex Operations (Insertion/Deletion/Relabeling): Inserting, deleting or relabeling a vertex over g_1^i will affect d edges at most. Hence, $md(g_1^i, g_1^{i+1}) \leq d \cdot cost(p_i^v)$.

Above all, we have the following inequality:

$$\begin{aligned} md(g_1, g_2) &\leq 1 \cdot \textstyle\sum_1^{k_1} cost(p_i^e) + d \cdot \sum_1^{k_2} cost(p_i^v) \\ &\leq d \cdot (\textstyle\sum_1^{k_1} cost(p_i^e) + \sum_1^{k_2} cost(p_i^v)) \\ &\leq d \cdot sged(g_1, g_2) \end{aligned}$$

Thus, $sged(g_1, g_2) \geq \frac{md(g_1, g_2)}{d}$. □

Algorithm 4 TA_PMD(q, g)

Input: A query graph q, a candidate graph g with edges $\widehat{e_1}, \ldots, \widehat{e_m}$;
Output: The pseudo-mapping distance $pmd(q, g)$.
1: $pmd(q, g) \leftarrow 0$
2: **if** There do not exist identical edges in g **then**
3: $\quad pmd(q, g) \leftarrow \sum sged(e_i, \widehat{e_i})$
4: **else**
5: $\quad$ **for** Each non-repetitive edge $\widehat{e_j}$ **do**
6: $\quad\quad pmd(q, g) \leftarrow pmd(q, g) + sged(e_j, \widehat{e_j})$
7: $\quad$ **for** Each set of identical repetitive edges $\widehat{E}$ **do**
8: $\quad\quad$ Let $sged(e_i, \widehat{e_i})$ be the smallest one
9: $\quad\quad pmd(q, g) \leftarrow pmd(q, g) + sged(e_i, \widehat{e_i})$
10: $\quad\quad$ remove $\widehat{e_i}$ from $\widehat{E}$
11: $\quad$ **for** Each set of identical repetitive edges $\widehat{E}$ **do**
12: $\quad\quad$ **for** Each repetitive edge $\widehat{e_j} \in \widehat{E}$ **do**
13: $\quad\quad\quad$ Replace $\widehat{e_j}$ with a special empty edge e_ϕ
14: $\quad\quad\quad pmd(q, g) \leftarrow pmd(q, g) + sged(e_j, e_\phi)$
15: **return** $pmd(q, g)$

Computing the mapping distance above is equivalent to the maximum matching problem. A classical algorithm to solve the maximum matching problem is the hungarian method [6]. However, it suffers from the time complexity $O(m^3)$, where m is the larger edge number in graphs q and g.

To improve the efficiency, we propose a pseudo-mapping distance (Definition 6.2), which can be computed in linear time.

DEFINITION 6.2. (***Pseudo-mapping distance***.) *During the TA-style exploration of each set of candidate edges, any mapping between q and g is called a pseudo-mapping. The summation of the semantic distances between edges in the pseudo-matching is called the pseudo-mapping distance, denoted by $pmd(q, g)$.*

Algorithm 4 gives the details of how to compute $pmd(q, g)$, where we deal with two cases. Given a candidate graph g consisting of edges $\widehat{e_1}, \ldots, \widehat{e_m}$, if there do not exist identical edges, we can sum up the semantic edit distance between each mapping pair $\langle e_i, \widehat{e_i} \rangle$, i.e., $pmd(q, g) \leftarrow \sum sged(e_i, \widehat{e_i})$ (lines 2-3). Otherwise, we first sum up the non-repetitive edges (lines 5-6) and the smallest $sged(e_i, \widehat{e_i})$ in each set of repetitive edges (lines 7-9). Regarding the rest repetitive edges, they are replaced with special empty edges e_ϕ. Then we compute $sged(e_j, e_\phi)$, where $e_j \in q$ is the edge matching the replaced $\widehat{e_j}$. Finally, $pmd(q, g)$ is returned.

Time Complexity. The operation on line 3 of Algorithm 4 is just the summation of the semantic edit distance of each mapping

edge $(e_i, \widehat{e_i})$. Lines 5-14 deal with the case that there exist some identical edges in $\widehat{e_1}, \ldots, \widehat{e_m}$. The time cost of computing the minimum $sged(e_i, \widehat{e_i})$ (line 8) is $O(|\widehat{E}|)$. In the worst case, computing the minimum $sged(e_i, \widehat{e_i})$ for all sets of identical repetitive edges (lines 7-10) is $O(m)$. Since there are m edges at most, the time cost of lines 11-14 is $O(m)$. Hence, the overall time complexity of Algorithm 4 is $O(m)$.

Remark. We use the pseudo-mapping distance $pmd(g_1, g_2)$ to replace the optimal mapping distance $md(g_1, g_2)$ in Equation 3. Although it may not correspond to the optimal mapping, we can use it to facilitate the query processing without generating any false negatives. More discussions will be presented in Section 6.2.

6.2 Answer Generation

In this subsection, we first introduce an A^* algorithm to refine the candidate graphs, and then present a whole picture of the *TA-style* exploration based answer generation.

6.2.1 Verification

For the candidate graphs that are not pruned, we need to compute the exact semantic graph edit distance $sged(q, g)$. Since the last three operations, semantic edge redirection, star substitution, and path substitution, have been handled in the query rewriting and pruning phases, we can just consider the first six semantic graph edit operations.

Analogous to the computation of the traditional graph edit distance [14], we adopt A^* algorithm to explore the search space. During the exploration, we maintain a cost function $f(x)$ consisting of two parts, i.e., $f(x) = g(x) + h(x)$, where $g(x)$ is the distance caused by the current mapping part, and $h(x)$ is estimated according to some heuristics over the unmatched part. If $f(x)$ is larger than the current minimum cost, the searching branch can be pruned.

6.2.2 Put It All Together

Algorithm 5 gives the details of answer generation, which contains three steps as follows.

Step 1. We sort the candidate edges in each $C(e_i)$ in non-descending order according to $sged(e_i, e_j)$ (lines 1-2).

Step 2. We retrieve k graphs by accessing each $C(e_i)$ in parallel (lines 3-6), and insert graph g into the buffer B in increasing order based on $sged(q, g)$ (lines 7-8).

Step 3. We perform the *TA-style* access to each candidate set $C(e_i)$. Specifically, we retrieve the next candidate graph $\widehat{g_i}$ with the minimum $pmd(q, \widehat{g_i})$. If $\widehat{g_i}$ has not been discarded yet, we deduce the pseudo-lower bound t based on the pseudo-mapping distance that is computed by Algorithm 4 (lines 11-16). We can discard graph $\widehat{g_i}$ if t is not smaller than $sged(q, g_B)$, where g_B is the last graph in B. Otherwise, $sged(q, \widehat{g_i})$ is computed to determine whether the buffer B needs to be updated (lines 17-23). The exploring process terminates when it is impossible to produce any true answers, i.e., the current threshold $t \geq sged(q, g_B)$.

Note that we compute the lower bound based on the pseudo-mapping distance. Although it may not correspond to the optimal mapping, we use it to filter out the unpromising candidate graphs.

Correctness. We clarify that Algorithm 5 will not generate any false negatives. For ease of the presentation, let t denote the pseudo-mapping distance based lower bound, and o denote the optimal lower bound. It is straightforward that $o \leq t$. If $t < sged(q, g_B)$, the optimal lower bound o must be smaller than $sged(q, g_B)$. Otherwise, we just skip the verification of the candidate g temporarily instead of discarding it. If g is a true answer, it will be found in the subsequent searching. Thus the correctness of Algorithm 5 is guaranteed.

Algorithm 5 AnswerGeneration(q, C)

```
Input:  A query graph q, the candidates C = {C(e_1), ..., C(e_m)}
        for edges e_i ∈ q, user-specified threshold k;
Output: Top-k matches for q
 1: for Each candidate set C(e_i) do
 2:    Sort each candidate edge e_j ∈ C(e_i) in the non-descending
       order according to sged(e_i, e_j)
 3: Maintain a buffer B of bounded size k
 4: while B is not full do
 5:    Do sorted access in parallel to each C(e_i)
 6:    Retrieve the graph g consisting of edges e_1, ..., e_m
 7:    Compute sged(q, g)
 8:    Insert g into B in the increasing order based on sged(q, g)
 9: t ← 0
10: while t < sged(q, g_B) do
11:    Retrieve the next graph ĝ_i with the minimum pmd(q, ĝ_i)
12:    if ĝ_i has been discarded then
13:       Continue
14:    t ← TA_PMD(q, ĝ_i)
15:    d ← The maximum vertex degree in q and ĝ_i
16:    t ← t/d
17:    if t < sged(q, g_B) then
18:       Compute sged(q, ĝ_i)
19:       if sged(q, ĝ_i) < sged(q, g_B) then
20:          Pop the last graph g_B in B
21:          Insert ĝ_i into B and discard graph g_B
22:       else
23:          Discard graph ĝ_i
24:    return Graphs in B
```

7. EXPERIMENTAL STUDY

7.1 Experimental Setup

7.1.1 Datasets

Dataset1. DBpedia: DBpedia 3.9[5] is an open-domain knowledge base, which is constructed by using the structured information extracted from Wikipedia[6]. It contains 5,040,948 vertices and 61,481,483 edges.

DBpedia Ontology: Each entity in DBpedia has a type (if an entity has no type, it is assigned with a root type, i.e., "Thing"). Besides the type ontology, there is a predicate ontology describing the relations of predicates.

SPARQL Queries: We use QALD-4[7], a benchmark delivered in the fourth evaluation campaigns answering over linked data. It contains 236 SPARQL queries, each of which has the answers. Note that a SPARQL query may involve multiple UNION operators, which correspond to different structural patterns. In our experiments, we randomly select only one of these UNION operators as a query each time. It is desired that we can find the complete answers instead of using the complete UNION operators.

Dataset2. Yago: Yago [5] is an RDF (Resource Description Framework) knowledge base that is extracted from Wikipedia, WordNet, and GeoNames. It contains 10,538,013 entities and 183,041,129 edges. There are 22,137 types and 100 predicates in the type ontology and predicate ontology, respectively. Considering three shortcut operations, we derive ten queries based on those used in [12].

[5]http://blog.dbpedia.org/
[6]http://www.wikipedia.org/
[7]http://qald.sebastianwalter.org/index.php?x=challenge&q=4

Type	*Predicate*	*Neighborhood label*	*Subtype*
SportsLeague	*country*	*Australia*	*AustralianFootballLeague*
Lake	*country*	*Denmark*	*LakeOfDenmark*
Book	*series*	*The_Courtney_Novels*	*Novel*
Politician	*successor*	*William_Steuart*	*Mayor*
Comics	*publisher*	*Manga_Entertainment*	*Manga*

Figure 11: Patterns of concept generation.

Type1	*Type2*	*Redirecting predicates*
Settlement	*SoccerManager*	*manager / managerClub*
Person	*Person*	*influenced / infulencedBy*
Lake	*Lake*	*inflow / outflow*
Person	*Politician*	*successor / predecessor*
TelevisionShow	*TelevisionShow*	*previousWork / subsequentWork*

Figure 12: Patterns of edge redirection.

7.1.2 Metrics

Since QALD-4 provides the golden standard, we adopt two classical metrics, i.e., *precision@k* (the ratio of the correctly discovered matches over all discovered top-k matches, denoted by P) and *recall* (the ratio of the correctly discovered matches over all correct matches, denoted by R), to evaluate the effectiveness of our approach. For simplicity, we also employ F1-measure to combine both the precision and recall according to Equation 4.

$$F1 = \frac{2}{1/P + 1/R} \quad (4)$$

To evaluate the effectiveness of our proposed three shortcut graph edit operations (i.e., the edge redirection, star substitution, and path substitution), we propose fact coverage ratio, fcr, as formally defined in Equation 5, where $|Facts|$ represents all facts (i.e., edges) in the knowledge graph, and $|Facts(o)|$ represents the number of facts satisfying the shortcut graph edit operation o.

$$fcr = \frac{|Facts(o)|}{|Facts|} \quad (5)$$

We use the *response time* (the time cost of pruning and refining) to measure the efficiency of our method.

All experiments are conducted on an Intel(R) Xeon(R) CPU E5504 @ 2.00GHz and 30G RAM, on Windows Server 2008. All programs were implemented in C++.

7.2 Effectiveness Evaluation

In this subsection, we first present the mining results for semantic graph pattern, and then report the effectiveness of our proposed method in terms of recall and precision.

7.2.1 Mining Semantic Graph Patterns

As discussed in Section 2.1, semantic meaning equivalent instances are built based on Patty [11] and Probase [21]. It contains 2,560,000 single entities and 3,862,331 pairs of entities.

Using the Algorithms 1 and 2 given in Section 2.1, we have found some interesting results. The precisions of the two methods are 0.61 and 0.64, respectively. Due to the space limitations, we only present some examples in Figs. 11, 12 and 13. Fig. 11 shows some interesting results for the concept generalization. For example, if a *SportsLeague* belongs to *Australia*, we can use a directed edge to represent it, i.e., it is an *AustralianFootballLeague*.

As presented in Fig. 12, one person p_1 influenced another person p_2, which corresponds to (p_1, influenced, p_2). In actual, it is equivalent to the expression that p_2 is influencedBy person p_1, which corresponds to (p_2, influencedBy, p_1).

{(*p1, birthPlace, c1*)};	{(*p1, birthPlace, City, Country, c1*)}
{(*p1, deathPlace, c1*)};	{(*p1, deathPlace, City, Country,c1*)}
{(*p1, grandFather, p2*)};	{(*p1, father, p3, father,p2*)}
{(*o1, type, t1*)};	{(*o1, type, t2, subtype, t1*)}

Figure 13: Patterns of inductive reference.

Table 2: Effectiveness evaluation

	DBpedia			Yago	
Method	Precision	Recall	F1	Precision	|Correct|
gStore	1	0.332	0.496	1	3
NeMa	0.521	0.690	0.593	0.467	7
SLQ	0.583	0.745	0.654	0.6	9
S^4	0.712	0.866	0.781	0.733	11

7.2.2 SPARQL Similarity Search

Since it has been shown that the latest work NeMa [7] outperforms its competitors BLINK [4], IsoRank [17], SAGA [18], and Ness [8], we only need to compare our method with NeMa. We also compare our method with gStore [28].

It is easy to compute the precision and recall with the QALD-4 benchmark dataset. However, we have no gold standard on Yago dataset. Therefore, we only report precision and the number of correct answers. Table 2 presents the evaluating results. In all experiments, the summarization ratio θ is set to be 60% by default.

As shown in Table 2, the precision of gStore is 100%. It is because that gStore is built based on the exact subgraph matching. However, due to the same reason, it can only find the matches that are isomorphic to the query graph, which results in low recall. In comparison, our method achieves the best performance in terms of both precision and recall.

Case study. To answer the question "Give me all cars that are produced in Germany", we only use a simple SPARQL query as follows to search the knowledge graph DBpedia.

```
SELECT ?x WHERE
{?x <type> Automobile.   ?x <production> Germany.}
```

As shown in Table 3, gStore only finds 133 matches. The number of returned answers is set to be 600 for NeMa, SLQ, and S^4. NeMa, SQL, and S^4 find 302, 411, and 554 correct answers, respectively. Actually, QALD-4 gives 554 correct answers for the question, which are all found by our method.

7.2.3 Evaluation of Semantic Graph Edit Operations

We exploit fcr (defined in Equation 5) to measure the proportion of facts that are covered by an graph edit operation. Table 4 gives fcrs for the edge redirection, the star substitution, and the path substitution, respectively. The three shortcut operations cover nearly 88% of all the facts in the knowledge graph.

To further study the effect of shortcut operations, we conduct experiments by disallowing partial shortcut operations. As shown Table 5, F1 decreases greatly by disallowing any shortcut operations, e.g., F1 is 0.649 if none of the shortcut operations is allowed. It confirms that our proposed shortcut operations are effective.

Table 3: Case study

Method	correct answers	all returned answers
gStore	133	133
NeMa	302	600
SLQ	411	600
S^4	554	600

Table 4: Fact coverage ratio on DBpedia

Operations	edge red.	star sub.	path sub.	all
fcr	0.73	0.68	0.63	0.88

Table 5: Effect of shortcut operations on DBpedia

Operations	precision	recall	F1
None	0.619	0.684	0.649
edge red.	0.672	0.746	0.707
star sub.	0.651	0.731	0.688
path sub.	0.641	0.703	0.671
edge red. + star sub.	0.701	0.792	0.743
edge red. + path sub.	0.699	0.787	0.740
star sub. + path sub.	0.682	0.781	0.728
All	**0.712**	**0.866**	**0.781**

7.3 Efficiency Evaluation

7.3.1 Index Construction

We study the effect of the summarization ratio θ since it controls the summarizing process.

As shown in Table 6, the index size and index time both increase with the growth of θ. That is because if θ becomes larger, it may generate more semantic graphs. In comparison, the indexing size and time for gStore on DBpedia: 6.2 GB and 6,800 seconds, on Yago: 10.4GB and 14,000 seconds. The index size and building time for NeMa on DBpedia: 4.3 GB and 10,350 seconds, on Yago: 7.9 GB and 22,610 seconds. Therefore, our index is easy to build.

7.3.2 Query Performance

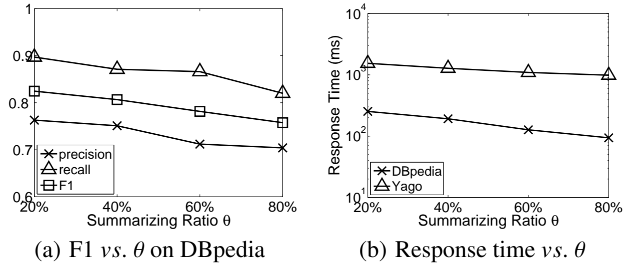
Effect of θ. As shown in Fig. 14(a), the F1 values decrease with the growth of the summarizing ratio. That is because larger θ will generate more semantic summary graphs, which results in more semantic operations of semantic substitution (e.g., semantic edge redirection, semantic star substitution, and semantic path substitution) during the candidate computation. Since these semantic operations are not exactly correct, both precision and recall decrease. However, the response time (the time elapsed from receiving the query to returning results) consumed in the query processing is reduced benefitting from the summarized index.

Effect of k. As depicted in Fig. 15(a), F1 achieves the maximum when k is 100. The reason is that if we increase k, the recall improves greatly at the cost of degrading the precision. Fig. 15(b) gives the effect of k over response time. It is straightforward that delivering more answers will consume more searching time.

Fig. 16(a) shows that S^4 has higher F1 value than NeMa and SLQ by varying the number of returned answers, which indicates S^4 is more effective. Furthermore, the time efficiency of S^4 outperforms NeMa and SLQ greatly as shown in Fig. 16(b). To study our proposed summary index, we turn off the it, denoted by "WSum". Its time efficiency degrades nearly two orders of magnitude, which confirms the effectiveness of our proposed index. Similar results can be also found on the Yago dataset as shown in Figure 17.

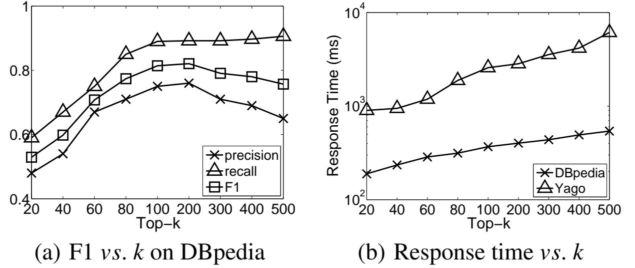
Table 6: Index building cost

	DBpedia		Yago	
θ	size (MB)	time (s)	size (MB)	time (s)
20%	301	63	1102	731
40%	341	68	1190	796
60%	397	70	1267	873
80%	435	71	1289	921



(a) F1 *vs.* θ on DBpedia (b) Response time *vs.* θ

Figure 14: Evaluation of θ



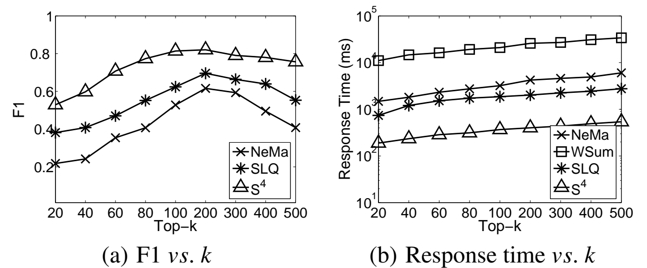
(a) F1 *vs.* k on DBpedia (b) Response time *vs.* k

Figure 15: Evaluation of k

Effect of Noise. To study the impact of noises, we randomly generate some noises on both vertex labels and structures. The noise ratio is defined as $(\Delta_V + \Delta_E)/|G|$, where Δ_V denotes the number of changed vertices, Δ_E denotes the number of changed edges, and $|G|$ is the sum of vertices and edges in G. As shown in Fig. 18(a), F1 values of the four methods all decrease as we increase noises. The performance of gStore decreases sharply. The reason is that it resorts to the exact subgraph match, and does not tolerate noises. In comparison, S^4 has the best robustness using the semantic graph edit distance. The response time of all methods increase slightly with the growth of noises as presented in Fig. 18(b).

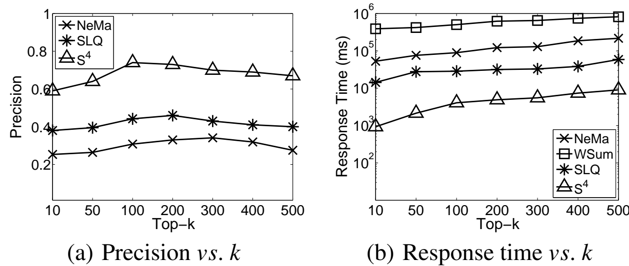
8. RELATED WORK

Structure-based similarity Search. As a classical measure of the structure similarity, graph edit distance is widely used in the existing approaches [2, 26, 18, 19]. Based on the graph edit distance, i.e., the minimum number of edit operations required to transform g_1 into g_2, Zheng et al., define the graph similarity search over a large number of graphs [26]. SAGA [18] employs a flexible model that is a variation of graph edit distance. It allows for node gaps, node mismatches and graph structural differences. However, it does not take the edge edit operations into consideration. Similar to the SAGA, TALE [19] proposes the node match quality based on some node misses and mismatches. Although *kGPM* [2] allows a path to match an edge, it restricts that vertex/edge labels specified by query graph q should be exactly matched.



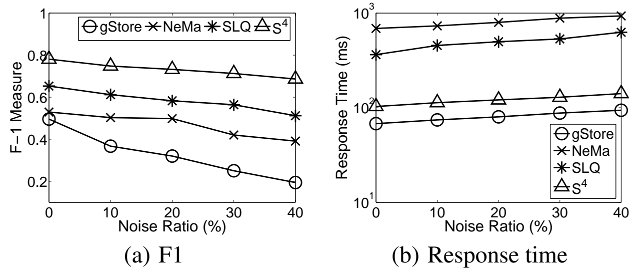
(a) F1 *vs.* k (b) Response time *vs.* k

Figure 16: Comparison with NeMa and SLQ (DBpedia)



(a) Precision *vs. k* (b) Response time *vs. k*

Figure 17: Comparison with NeMa and SLQ (Yago)



(a) F1 (b) Response time

Figure 18: Evaluation of noise (DBpedia)

To convey the global structural information, Ness [8] and NeMa [7] propose the neighborhood-based similarity measure, which unifies both the label matching cost and neighborhood matching cost.

Semantics is a key factor that affects the matching results over RDF knowledge graphs. Therefore, the major problem of the methods above is that they do not take the semantics into consideration.

Concept-level similarity Search. To enable semantic queries over a knowledge graph, some efforts have been made to apply the "semantic" similarity, but they only consider the "concept-level" similarity. KMatch [22] defines a quantitative metric to measure the similarity between the query graph q and its matches in the knowledge graph G. Specifically, it computes the similarity between vertex labels based on an ontology-based distance. However, it restricts that q and its matches must share the same graph structure including the edge label constraints. Thus, it is not able to convey the structure similarity. Recently, SLQ [23] introduces a framework that integrates a set of transformation functions, such as "synonym", "ontology" and "distance" (i.e., transforming an edge to a shortest path). In other words, it predefines several rules that can be plugged into the system. Note that the transformations of the structure (corresponding to "distance") and semantics (corresponding to "ontology") are exploited separately. Hence, it is hard to deal with more complicated structures (e.g., stars). In comparison, we propose a novel similarity metric, semantic graph edit distance, to integrate both structure similarity and semantic similarity together.

9. CONCLUSIONS

In this paper, we focus on the problem of semantics-based SPARQL similarity search over RDF knowledge graphs. Considering the diverse semantically equivalent graph structures, we propose an instance-driven approach to mine the semantic graph patterns. Based on the semantic graph patterns, we propose a novel similarity measure, i.e., *semantic graph edit distance*. We devise an efficient index, semantic summary graph, to facilitate the query processing. At query time, the input query graphs are rewritten and a two-level pruning technique is performed to prune the search space. The experimental results on real datasets confirm the effectiveness and efficiency of our method.

Acknowledgments

This work was supported by China 863 project under grant No. 2015AA015402, NSFC under grant No. 61532010, 61402020, 61572272 and NSF CCF under grant No. 1548848. This work was also supported by a grant from the Ph.D. Programs Foundation of Ministry of Education of China (No. 20130001120021). The first author is currently a postdoc research fellow at The Chinese University of Hong Kong. Lei Zou is the corresponding author.

10. REFERENCES

[1] R. Angles and C. Gutiérrez. Querying RDF data from a graph database perspective. In *ESWC*, 2005.

[2] J. Cheng, X. Zeng, and J. X. Yu. Top-k graph pattern matching over large graphs. In *ICDE*, 2013.

[3] A. Fader, S. Soderland, and O. Etzioni. Identifying relations for open information extraction. In *EMNLP*, 2011.

[4] H. He, H. Wang, J. Yang, and P. S. Yu. BLINKS: ranked keyword searches on graphs. In *SIGMOD*, 2007.

[5] J. Hoffart, F. M. Suchanek, K. Berberich, and G. Weikum. YAGO2: A spatially and temporally enhanced knowledge base from wikipedia. *Artif. Intell.*, 194, 2013.

[6] H.W.Kuhn. The hungarian method for the assignment problem. In *Naval Research Logistics*, 1955.

[7] A. Khan, Y. Wu, C. C. Aggarwal, and X. Yan. Nema: Fast graph search with label similarity. *PVLDB*, 6(3), 2013.

[8] A. Khan, X. Yan, and K.-L. Wu. Towards proximity pattern mining in large graphs. In *SIGMOD Conference*, 2010.

[9] W. Le, F. Li, A. Kementsietsidis, and S. Duan. Scalable keyword search on large RDF data. *IEEE TKDE*, 26(11), 2014.

[10] N. Nakashole, T. Tylenda, and G. Weikum. Fine-grained semantic typing of emerging entities. In *ACL*, 2013.

[11] N. Nakashole, G. Weikum, and F. M. Suchanek. PATTY: A taxonomy of relational patterns with semantic types. In *EMNLP-CoNLL*, 2012.

[12] T. Neumann and G. Weikum. RDF-3X: a risc-style engine for RDF. *PVLDB*, 1(1), 2008.

[13] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *VLDB J.*, 10(4), 2001.

[14] K. Riesen, S. Fankhauser, and H. Bunke. Speeding up graph edit distance computation with a bipartite heuristic. In *MLG*, 2007.

[15] P. Shvaiko and J. Euzenat. A survey of schema-based matching approaches. 2005.

[16] P. Shvaiko and J. Euzenat. Ontology matching: State of the art and future challenges. *IEEE Trans. Knowl. Data Eng.*, 25(1), 2013.

[17] R. Singh, J. Xu, and B. Berger. Global alignment of multiple protein interaction networks. In *Biocomputing*, 2008.

[18] Y. Tian, R. C. McEachin, C. Santos, D. J. States, and J. M. Patel. Saga: a subgraph matching tool for biological graphs. *Bioinformatics*, 23(2), 2007.

[19] Y. Tian and J. M. Patel. Tale: A tool for approximate large graph matching. In *ICDE*, 2008.

[20] 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.

[21] W. Wu, H. Li, H. Wang, and K. Q. Zhu. Probase: a probabilistic taxonomy for text understanding. In *SIGMOD*, 2012.

[22] Y. Wu, S. Yang, and X. Yan. Ontology-based subgraph querying. In *ICDE*, 2013.

[23] S. Yang, Y. Wu, H. Sun, and X. Yan. Schemaless and structureless graph querying. *PVLDB*, 7(7), 2014.

[24] S. Zhang, J. Yang, and W. Jin. Sapper: Subgraph indexing and approximate matching in large graphs. *PVLDB*, 3(1), 2010.

[25] X. Zhao, C. Xiao, X. Lin, Q. Liu, and W. Zhang. A partition-based approach to structure similarity search. *PVLDB*, 7(3), 2013.

[26] W. Zheng, L. Zou, X. Lian, D. Wang, and D. Zhao. Efficient graph similarity search over large graph databases. *TKDE*, 27(4), 2015.

[27] W. Zheng, L. Zou, X. Lian, and D. Zhao. Graph similarity search with edit distance constraint in large graph databases. In *CIKM*, 2013.

[28] L. Zou, J. Mo, L. Chen, M. T. Özsu, and D. Zhao. gstore: Answering SPARQL queries via subgraph matching. *PVLDB*, 4(8), 2011.