Ontology-based Subgraph Querying

Yinghui Wu Shengqi Yang Xifeng Yan

University of California Santa Barbara

{yinghui, sqyang, xyan}@cs.ucsb.edu

***Abstract*— Subgraph querying has been applied in a variety of emerging applications. Traditional subgraph querying based on subgraph isomorphism requires identical label matching, which is often too restrictive to capture the matches that are semantically close to the query graphs. This paper extends subgraph querying to identify semantically related matches by leveraging ontology information. (1) We introduce the *ontology-based subgraph querying*, which revises subgraph isomorphism by mapping a query to semantically related subgraphs in terms of a given ontology graph. We introduce a metric to measure the similarity of the matches. Based on the metric, we introduce an optimization problem to find top K best matches. (2) We provide a filtering-and-verification framework to identify (top-K) matches for ontology-based subgraph queries. The framework efficiently extracts a small subgraph of the data graph from an *ontology index*, and further computes the matches by only accessing the extracted subgraph. (3) In addition, we show that the ontology index can be efficiently updated upon the changes to the data graphs, enabling the framework to cope with dynamic data graphs. (4) We experimentally verify the effectiveness and efficiency of our framework using both synthetic and real life graphs, comparing with traditional subgraph querying methods.**

I. Introduction

It is increasingly common to find large data modeled as graphs, where each labeled node represents a real life entity with attributes, and each edge denotes a relationship between two entities [23]. With this comes the need for effective *subgraph querying* [15], [32]. Given a query as a graph Q and a data graph G, the subgraph querying is to find the subgraphs of G as *matches* which are isomorphic to Q.

Traditional subgraph querying adopts identical label matching, where a query node in Q can only be mapped to a node in G with the same label. This is, however, an overkill in identifying matches with similar interpretations to the query in some domain of interest [15]. In such matches, a query node may correspond to a data node in G which is *semantically related*, instead of a node with an identical label. The need for this is evident in querying social networks [9], biological networks [31] and semantic Web [3], among others.

Example I.1: Consider the graph G shown in Fig. 1 which depicts a fraction of a social travel network [2]. Each node represents an entity of types such as tourist groups (Holiday Tours (HT), Culture Tours (CT)), attractions (Disneyland, Royal Gallery (RG)), leisure centers (Holiday Plaza (HP), Royal Palace (RP)), or restaurants (Holiday Cafe (HC), riverside); and each edge represents a relation between two entities, *e.g.,* "has guides for" (guide), or "recommend" (recom).

Consider a query Q given in Fig. 1 from a tourist. It is to find some other tourists who (1) recommend museum tours with guide services, and (2) favor a restaurant named "moonlight",

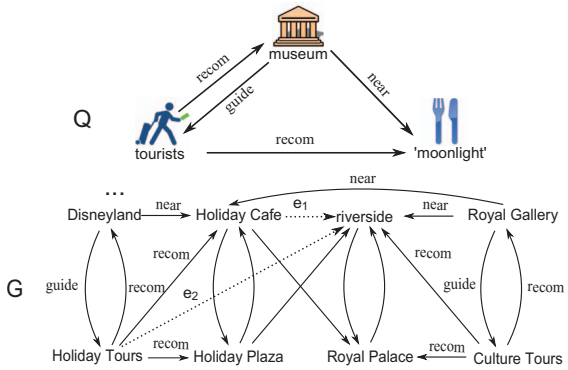


Fig. 1. Searching travel social network

which in turn is close to the museum. Traditional subgraph isomorphism cannot identify any match for Q in G with identical labels. Indeed, there is no node in G with the same, or even textually similar labels for the labels in Q. However, there are data nodes in G which are *semantically close* to the query nodes, and thus should be considered as potential matches. For example, node Royal Gallery in G is intuitively a kind of museum in Q. Nevertheless, it is also difficult to determine their closeness by using Q and G alone. □

The above example illustrates the need to identify node matches that are close to the query nodes, rather than those with identical or similar labels. Several extensions for subgraph isomorphism have been proposed to identify matches with node similarity [10], [14], [33], while assuming as input the similarity information between query nodes and data nodes. However, as observed in [11], users may not have the full knowledge to provide such information.

To this end, we need to understand the semantic relationships among the query nodes and data nodes, *i.e.,* given the label of a query node, which labels are semantically close to the label, in terms of standard description of entities. This is possible given the emerging development of *ontology graphs* [7], [13], [31]. An ontology graph typically consists of (1) a set of concepts or entities, and (2) a set of semantic relationships among the nodes. The ontology graphs may benefit the subgraph query evaluation by providing additional information about the relationships and similarity among the entities. Consider the following example.

Example I.2: Fig. 2 illustrates a travel ontology graph O_g [9] provided by a travel social network service, which illustrates the relationships between the entities in G (Fig. 1). According to O_g, (a) RG is a kind of Museum, while Disneyland is not, (b) riverside and moonlight refer to the same restaurant in O_g, while HC is a different restaurant, and (c) CT and HT are both close to the term tourists. Given this, the subgraph

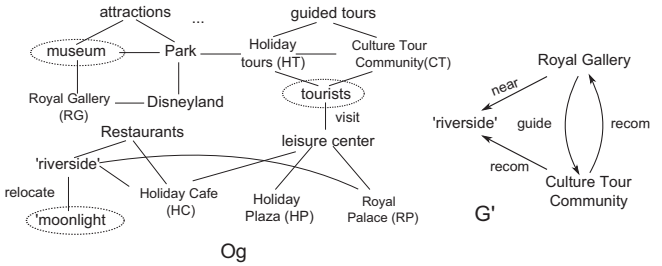


Fig. 2. Ontology-based matching

G' of G given in Fig. 2 *should* be a match close to Q. Indeed, each edge of Q (*e.g.,* (museum, tourist)) can be mapped to an edge of G' with highly related nodes (*e.g.,* (Royal Gallery, Cultural Tour Community)).

On the other hand, consider the subgraph G'' (not shown) induced by Disneyland, Holiday Cafe and Holiday tours. Although its three nodes are related with Museum, tourists and moonlight, respectively, they are not as close as the nodes in G' according to O_g. For example, Disneyland is more similar to the term Park than Museum. Thus, G' should be considered as a better match for Q, according to O_g. □

The ontology information has been used in *e.g.,* keyword searching [18], semantic queries [13], [11], [19], and social networks [9]. Nevertheless, little is known on how to exploit ontology graphs for effective subgraph querying. Moreover, it is important to develop efficient query evaluation techniques, especially when a query may have multiple "interpretations" and matches in terms of ontology-based similarity [18].

Contributions. In this paper we develop query evaluation techniques to efficiently identify matches that are close to a given query graph, by exploiting the ontology graphs.

(1) We propose *ontology-based subgraph querying* in Section II. (a) Given a data graph G, a query graph Q, and an ontology graph O which provides the semantic relationships among different ontologies, the ontology-based subgraph querying is to identify the matches for Q in G, where the nodes in the matches and the query are semantically close according to O. In contrast to subgraph isomorphism and its extensions, ontology-based subgraph querying measures the similarity of the nodes by exploiting the ontology graphs. (b) We introduce a metric to rank the matches of Q, based on the overall similarity of the labels between the query nodes and their matches, in terms of the ontology graphs. The metric gives rise to the the *top K matches problem*, which is to identify the K closest matches of Q in G.

(2) Based on the metric, we propose a filtering-and-verification framework for computing top-K matches (Section IV). (a) We introduce an *ontology index* based on a set of *concept graphs*, which are abstractions of the data graph G *w.r.t.* O. We show that the index can be constructed in *quadratic time*, by providing such an algorithm. (b) Using the index, we develop a filtering strategy, which extracts a small subgraph of G as a compact representation of the query results, in quadratic time. The time complexity is determined *only* by the size of the index and the query, rather than the size of entire G. (c) We provide a query evaluation algorithm (Section III) to compute the (top-K) matches following the filtering-and-verification strategy, which computes matches directly from the extracted subgraph *without* searching G.

(3) In addition, we provide techniques to incrementally maintain the ontology index (Section VI). Upon a set of updates to a data graph, the ontology index can be updated in quadratic time in terms of the size of the total *changes* in the data graph and the index, following [27].

(4) We experimentally verify the effectiveness and efficiency of our querying algorithms, using real-life data and synthetic data. We find that the ontology-based subgraph querying can identify much more meaningful matches than traditional subgraph querying methods. Our query evaluation framework is efficient, and scales well with the size of the data graphs, queries and ontology graphs. For example, our evaluation algorithm only takes up to 22% of the running time of a traditional subgraph querying method over real life graphs with 7.7M nodes and edges. Moreover, the construction and incremental maintenance of the index is efficient. The incremental algorithm outperforms the batch computation, and only takes up to 20% of the running time of batch computation in our tests.We contend that the framework serves as a promising method for subgraph querying using ontology graphs in practice.

Related Work. We categorize the related work as follows.

Subgraph query evaluation. There have been many works on subgraph queries evaluation [8], [25], [17], [32], [34], [35], based on traditional subgraph isomorphism using identical label matching (see [15] for a survey). These works develop pruning rules to reduce search space [25], [32], construct indexes based on graph features [8], [35], [34] *i.e.,* frequent substructures, or hierarchical graph containment relation [17], [36]. In contrast, we develop querying and indexing techniques to identify matches that are semantically close to the subgraph queries, in terms of ontologies, where these techniques can not be directly applied. The subgraph querying is extended with node similarity in [10], [14], [33], [36]. A similarity matrix between query nodes and data nodes are assumed as input, and the data nodes dissimilar with the query nodes are filtered according to a threshold. In contrast to these works, (a) we study subgraph queries with node and edge labels, which are ignored in most of these works; and (b) we provide efficient indexing techniques exploiting the ontology graphs, which can not be obtained from similarity matrix or functions alone. Moreover, as verified in Section VII, our query evaluation techniques *always* outperform the similarity matrix based algorithms in our tests.

Ontology-based graph queries. The ontology information has been used for pattern mining [6], keyword searching [18] and the semantic Web [3], [11], [21]. The Ontogator [22] exploits an ontology-based multi-facet search paradigm, which links keyword queries to a set of entities in multiple distinct ontology views, created via ontology projection. [6] proposes techniques to mine the frequent patterns over graphs with

generalized labels in the input taxonomies. Classes hierarchy are used to evaluate queries specified by a SPARQL-style language over RDF graphs in [11], where approximate answers are identified, measured by a distance metric. The template matching with semantic similarity is discussed in [3], where the matches are semantically similar entities. However, the structure of the template is not preserved, *i.e.,* the matches are not isomorphic to the template. Our work differs from theirs in the following. (a) We consider general ontology graphs rather than hierarchical taxonomies or class lattice. (b) We find matches for a given query graph, instead of discovering frequent patterns in graphs as in [6]. (c) The queries in [11] are defined in terms of a query language specified for semantic Web. In contrast, we study general subgraph queries with node and edge labels. Moreover, the queries in [11] are posed over RDF graphs with predefined schema, where we consider subgraph queries over general data graphs without any schema. (d) The query evaluation is not discussed in [3], [11], [21].

Closer to our work is [21], which extends template graph searching by interpolating ontologies to data graphs. The data graphs are recursively extended by a set of ontologies from ontology queries, and are then queried by a template graph. Our work differs from theirs in (a) instead of merging ontology graphs with data graphs, we leverage ontology graph to develop filtering strategies to identify matches, and (b) we provide query evaluation and indexing techniques, while [21] focus on data fusion techniques. The incremental querying techniques are also not addressed in [21].

Graph abstraction. The concepts of bisimulation [26] and regular equivalence [5] are proposed to define the equivalent graph nodes, which can be grouped to form abstracted graphs as indexes [24]. In this work we use the similar idea to construct the ontology index, by abstracting data graphs as a set of concept graphs for efficient subgraph filtering and querying. However, while the notions in [5], [26] are based on label equality, a concept graph groups nodes in a data graph in terms of an external ontology graph, thus unifies the ontology similarity and graph abstraction, as discussed in Section IV.

II. Ontology-based subgraph querying

Below we introduce data graphs and query graphs, as well as the ontology graphs. We then introduce the notion of the ontology-based subgraph querying.

A. Graphs, queries and ontology graphs

Data graph. A *data graph* is a directed graph $G = (V, E, L)$, where V is a finite set of *data nodes*, E is the edge set where $(u, u') \in E$ denotes a *data edge* from node u to u'; and L is a labeling function which assigns a label $L(v)$ (resp. $L(e)$) to a node $v \in V$ (resp. an edge $e \in E$) In practice the function L may specify (1) the node labels as the description of entities, *e.g.,* URL, location, name, job, age; and (2) the edge labels as relationships between entities *e.g.,* links, friendship, work, advice, support, exchange, co-membership [23].

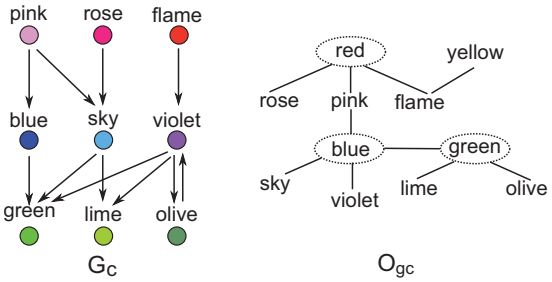


Fig. 3. Data graph and ontology graph

Query graph. A *query graph* is a directed graph defined as $Q = (V_q, E_q, L_q)$, where (1) V_q and E_q are a set of *query nodes* and *query edges*, respectively; and (2) L_q is a labeling function such that for each node $v \in V_q$ (resp. $e \in E_q$), $L_q(u)$ (resp. $L_q(e)$) is a node (resp. edge) label.

Ontology graph. In real life applications the ontologies and their relationships can typically be represented as standardized *ontology graphs* [4], [7], [13], [19]. An ontology graph $O = (V_r, E_r)$ is an undirected graph, where (1) V_r is a node set, where each node $v_r \in V_r$ is a label referring to an entity; and (2) E_r is a set of edges among the labels, where each edge $e_r \in E_r$ represents a semantic relation (*e.g.,* "refer to", "is a", "specialization" [19]) between two nodes.

In addition, we denote as $\mathsf{sim}(v_{r_1}, v_{r_2})$ a *similarity function*, which computes the similarity of two nodes v_{r_1} and v_{r_2} in O as a real value in $[0, 1]$. Following ontology-based querying [19], (1) $\mathsf{sim}(v_{r_1}, v_{r_2})$ is a *monotonically decreasing function* of the distance from v_{r_1} to v_{r_2} in O, and (2) $\mathsf{sim}(v_{r_1}, v_{r_2}) = \mathsf{sim}(v_{r_2}, v_{r_1})$. Intuitively, the closer v_{r_1} and v_{r_2} are in O, the more similar they are [11], [12], [19]. For example, $\mathsf{sim}(v_{r_1}, v_{r_2})$ can be defined as $0.9^{\mathsf{dist}(v_{r_1}, v_{r_2})}$, where $\mathsf{dist}(v_{r_1}, v_{r_2})$ is the distance from v_{r_1} to v_{r_2} in O [19].

Remarks. In practice, the ontology graphs and sim () can be obtained from *e.g.,* semantic Web applications [13], Web mining [20], or domain experts [4]. While proposing more sophisticated models for ontologies and similarity functions are beyond the scope of this paper, we focus on technique that applies to a class of similarity functions sim (). Note that sim () can also be revised for directed ontology graphs.

Example II.1: The graph Q (resp. G) in Fig. 1 depicts a query graph (resp. a data graph). There are three types of edge relations in both G and Q, *i.e.,* recom, near, and guide. All the other edges in G share a same type (not shown). The ontology graph O_g in Fig. 3 illustrates the relationships among the entities in G, *e.g.,*moonlight is *relocated* as riverside (edge e(moonlight, riverside)). A similarity function $\mathsf{sim}(v_{r_1}, v_{r_2})$ for O_g can be defined as $0.9^{\mathsf{dist}(v_{r_1}, v_{r_2})}$. For example, $\mathsf{sim}(\mathsf{museum}, \mathsf{Disneyland}) = 0.9^2 = 0.81$.

As another example, consider the data graph G_c and an ontology graph O_{g_c} given in Fig. 3. The nodes in G_c are labeled with colors (*e.g.,* blue). All the edges in G_c indicates the relationship "similar with", *e.g.,* the edge (red, rose) indicates that red is close to rose. Similarly, we define $\mathsf{sim}(v_{r_1}, v_{r_2})$ as $0.9^{\mathsf{dist}(v_{r_1}, v_{r_2})}$ for nodes v_{r_1} and v_{r_2} in O_{g_c}. □

B. Ontology-based Subgraph Querying

We next introduce the ontology-based subgraph querying.

Given a query graph $Q = (V_q, E_q, L_q)$, an ontology graph O, a data graph $G = (V, E, L)$, a similarity function $\mathsf{sim}()$ and a similarity threshold θ, the *ontology-based querying* is to find the subgraphs $G' = (V', E', L')$ of G, such that there is a *bijective function* h from V_q to V' where (a) $\mathsf{sim}(L(h(u)), L_q(u)) \geq \theta$, and (b) (u, u') is a query edge if and only if$(h(u), h(u'))$ is a data edge, and they have the same edge label. We refer to G' as a *match* of Q in G induced by the mapping h, and denote all the matches in G for Q as $Q(G)$. In addition, the *candidate set* for a query node u as the set of nodes v where $\mathsf{sim}(u, v) \geq \theta$. Here we assume *w.l.o.g.* that all the node labels in G are from O.

Top-K subgraph querying. In practice one often wants to identify the matches that are semantically "closest" to a query. We present a quantitative metric for the overall similarity between a query graph Q and its match G' induced by a mapping h, defined by a function C as follows.

$$C(h) = \sum \mathsf{sim}(L_q(u), L(h(u))), u \in V_q.$$

The metric favors the matches that are semantically close to the query: the larger the similarity score $C(h)$ is, the better the mapping is. On the other hand, if a subgraph G' matches Q with identical node labels, *i.e.,* via a subgraph isomorphism mapping h, $C(h)$ has the maximum value. Indeed, traditional subgraph isomorphism is a special case of the ontology-based subgraph querying, when the similarity threshold $\theta = 1$.

The metric naturally gives rise to an optimization problem. Given Q, G, O and an integer K, the *top K matches problem* is to identify K matches for Q in G with the largest similarity.

Example II.2: Recall the query Q, the data graph G in Fig. 1 and the ontology graph O_g in Fig. 2. Assume the similarity threshold $\theta = 0.9$. One may verify that the candidate set of query node museum $\mathsf{can}(\mathsf{museum}) = \{\mathsf{Royal\ Gallery}, \mathsf{attractions}, \mathsf{park}\}$, and similarly, $\mathsf{can}(\mathsf{moonlight}) = \{\mathsf{riverside}, \mathsf{Holiday\ Cafe}, \mathsf{Holiday\ Plaza}\}$. The match G' for Q in G has the maximum similarity $\mathsf{sim}(\mathsf{museum}, \mathsf{Royal\ Gallery}) + \mathsf{sim}(\mathsf{tourists}, \mathsf{Culture\ Tour\ Community}) + \mathsf{sim}(\mathsf{moonlight}, \mathsf{riverside}) = 0.9 * 3 = 2.7$. □

One may verify that the top K matches problem is NP-hard. Indeed, the traditional subgraph isomorphism is a special case of the problem, which is known to be NP-complete [33]. We next provide a query evaluation framework for the problem.

III. Querying Framework

Traditional ontology-based querying, by and large, relies on query rewriting techniques [6], which replaces query nodes with its candidates and may yield an exponential number of queries. These queries are then evaluated to produce all the results. This may not be practical for ontology-based subgraph querying. Alternatively, a similarity matrix can be computed, where each entry records the similarity between the query nodes and its candidates. Nevertheless, (1) the matrix incurs high space and construction cost ($O(|Q||G|)$), and needs to be computed upon each query, and (2) the time complexity is relatively high for both the exact algorithms (*e.g.,* [33]) and approximation algorithms [14] over the entire data graph.

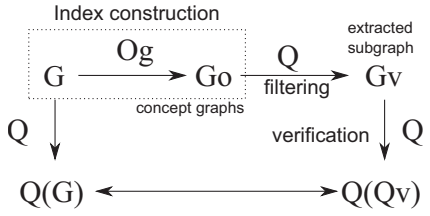


Fig. 4. Ontology-based querying framework

Using ontology graphs, we can do better. Since it is hard to reduce the complexity of the isomorphism test, we develop a filtering-and-verification framework to reduce the input of the ontology-based querying. Upon receiving a query, the framework evaluates the query as follows. (1) During the filtering phase, the framework uses an *ontology index* to either extract a small subgraph of the data graph that *contains* all the matches, or determine the nonexistence of the match, in *polynomial time*; and (2) during the verification phase, the framework extracts the best matches from the small subgraph in (1), *without* searching the entire data graph.

Overview of the framework. The framework has three components, as illustrated in Fig. 4. The ontology index is constructed once for all in the first phase, while the query is evaluated via the filtering and verification phases.

Index construction. The framework first constructs an *ontology index* for a data graph G, as a set of *concept graphs*. Each concept graph is an abstraction of G by merging the nodes with similar labels in the ontology graph. The index is precomputed once, and is dynamically maintained upon changes to G.

Filtering. Upon receiving a query Q, the framework extracts a small subgraph as a compact representation of all the matches that are similar to Q, by visiting the concept graphs iteratively. If such a subgraph is empty, the framework determines that Q has no match in G. Otherwise, the matches can be extracted from the subgraph directly without accessing G.

Verification. The framework then performs isomorphism checking between the query and the extracted subgraph to extract the (top K) matches for Q.

We next provide the details of each phase of the framework.

IV. Ontology-based Indexing

In this section we introduce the indexing and filtering phases of the ontology-based subgraph querying framework. We introduce the ontology index in Section IV-A, and present the filtering phase in Section IV-B based on the index.

A. Ontology Index

The ontology index consists of a set of abstractions of a data graph G. Each abstraction, denoted as a *concept graph*, is constructed by grouping and merging the nodes in G, which all have a label similar to a label in the ontology graph O.

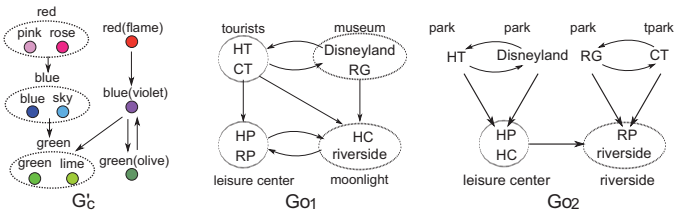


Fig. 5. Ontology index and concept graphs. Each concept graph is an abstraction of the same data graph by grouping nodes similar to a concept label, and provides different "perspectives" by using distinct concept label set.

Given a data graph $G = (V, E, L)$ and an ontology graph O (with similarity function sim), as well as a similarity threshold β, a *concept graph* $G_o = (V_o, E_o, L_o)$ is a directed graph where (1) V_o is a partition of V, where each $v_o \in V_o$ is a set of data nodes, (2) each v_o has a label $L_o(v_o)$ from O, such that for any data node $v \in v_o$ and its label $L(v)$, $\mathsf{sim}(L(v), L_o(v_o)) \geq \beta$, and (3) (v_{o_1}, v_{o_2}) is an edge in E_o if and only if for *each* node v_1 in v_{o_1} (resp. v_2 in v_{o_2}), there is a node v_2 in v_{o_2} (resp. v_1 in v_{o_1}), such that (v_1, v_2) (resp. (v_2, v_1)) is an edge in G. We refer the set of the labels $L_o(v_o)$ to as *concept labels*.

Intuitively, a concept graph provides a "perspective" of the data graph in terms of several concept labels from the ontology graph. (1) Each node in the concept graph represents a group of nodes that are all similar to (extended from) a same label as a "concept" [19]. (2) Each edge in the concept graph represents a set of edges connecting the nodes in the two groups of nodes corresponding to two concepts. Hence, a concept graph is an abstraction of a data graph, by capturing both the semantics of its node labels as well as its topology.

Remarks. The abstraction of a graph is typically constructed by grouping a set of similar or equivalent nodes. Bisimulation [26] and regular equivalence [5] are used to generate abstract views of graphs [24], where two nodes are equivalent if they have a set of equivalent children with the same set of labels. In contrast, the nodes in a concept graph contains the nodes that are similar to a same label in a given ontology graphs, even they themselves may not have the same label.

Based on the concept graphs, an *ontology index* $\mathcal{I}$ of G is a set of concept graphs $\{G_{o_1}, \ldots, G_{o_m}\}$, where each concept graph G_{o_i} has distinct concept label set and similarity threshold β. Note that we distinguish the similarity threshold β for generating concept graphs from the threshold θ for the queries (Section II), although they may have the same value.

Example IV.1: Consider the data graph G_c and the ontology graph O_{g_c} in Fig. 3. Fixing a similarity threshold $\beta = 0.81$, and setting $\Sigma = \{\mathsf{red}, \mathsf{blue}, \mathsf{green}\}$ in O_{g_c} as concept labels, a concept graph G'_c *w.r.t.* Σ is as shown in Fig 5. Each node in G'_c represents a set of nodes with labels similar to a concept label, *e.g.,* the node red is a set $\{\mathsf{rose}, \mathsf{pink}\}$, where $\mathsf{sim}(\mathsf{red}, \mathsf{rose})$ and $\mathsf{sim}(\mathsf{red}, \mathsf{pink})$ are both 0.9 (as defined in Example II.1). On the other hand, although the node violet is similar to a concept label blue, it is not grouped with the node sky in G'_c. Indeed, while violet has a parent olive similar with the concept label green, the node sky has no such parent.

Fig. 5 illustrates two concept graphs G_{o_1} and G_{o_2} for the data graph G in Fig. 1, where the similarity threshold β is 0.81. The concept graphs G_{o_1} and G_{o_2} are constructed in terms of two sets of concept labels {museum,tourists,moonlight, leisure center}, and {park,,riverside, leisure center}, respectively. An ontology index $\mathcal{I}$ is the set $\{G_{o_1}, G_{o_2}\}$. □

Input: Ontology graph O, a data graph G,
similarity threshold β, integer N;
Output: Ontology index $\mathcal{I}$.
1. $\mathcal{I} := \emptyset$;
2. generate N distinct concept label sets $\{C_1, \ldots, C_N\}$;
3. **for each** C_i **do**
4. $\quad \mathcal{I} := \mathcal{I} \cup \mathsf{CGraph}(\beta, C_i, O, G)$;
5. **return** $\mathcal{I}$;

Procedure CGraph
Input: Ontology graph O, data graph $G = (V, E, L)$, threshold β, concept label set $C = \{l_1, \ldots, l_m\}$.
Output: a concept graph $G_o = (V_o, E_o, L_o)$.
1. construct partition V_o of V as $\{V_1, \ldots, V_m\}$, where $L_o(V_i) := l_i$, $V_i = \{v | \mathsf{sim}(L(v), l_i) \geq \beta\}$;
2. set $E_o := \{(V_1, V_2) | (v_1, v_2 \in E), v_1 \in V_1, v_2 \in V_2\}$;
3. **while** there is change in V_o **do**
4. $\quad$ **if** there is an edge (v_{o_1}, v_{o_2}) where $v_1 \in v_{o_1}$ has no child in v_{o_2} (resp. $v_2 \in v_{o_2}$ has no parent in v_{o_1}) **then**
5. $\quad\quad$ SplitMerge(v_{o_1}, G_o) (resp. SplitMerge(v_{o_2}, G_o));
6. $\quad$ update G_o;
7. **return** $G_o := (V_o, E_o, L_o)$;

Fig. 6. Algorithm OntoIdx

Index construction. We next present an algorithm to construct the ontology index for a given data graph, in quadratic time.

Proposition 4.1: *Given a data graph $G(V, E, L)$, an ontology index can be constructed in $O(|E| \log |V|)$ time.* □

The algorithm, denoted as OntoIdx, takes as input the graphs G and O, a similarity threshold β, and an integer N as the number of the concept graphs to be generated. As shown in Fig. 6, the algorithm first initializes a set $\mathcal{I}$ as the ontology index (line 1). It then performs the following two steps.

Concept labels selection (line 2). OntoIdx uses the following strategy to generate concept label sets by exploiting the partition techniques. For a given similarity threshold β, (1) it partitions O via graph clustering or ontology partitioning techniques [1], [30], [28], where the nodes in O are partitioned into several clusters, and (2) for each cluster, OntoIdx iteratively selects a label l and add it into a set C, and removes all the labels l' where $\mathsf{sim}(l, l') \geq \beta$ in the cluster. The process repeats until there is no label remains in the cluster, and the set C is returned after all the clusters are processed in O. One may verify that the strategy produces a set of concept labels C, such that for any label in a data graph l', there exists a concept label $l \in C$ where $\mathsf{sim}(l, l') \geq \beta$. OntoIdx uses the strategy to generate N distinct sets of concept labels (line 2).

Concept graph construction (lines 3-5). After the concept labels are selected, OntoIdx then invokes procedure CGraph to compute a concept graph and extend $\mathcal{I}$ (lines 3-4) for each

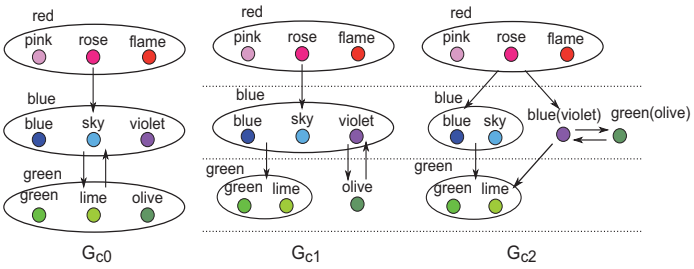


Fig. 7. Construction of concept graphs

concept label set C_i, until all C_i are processed (line 5).

Procedure CGraph constructs a concept graph G_o as follows. It constructs the node set V_o as a node partition of the data graph G, where each node of V_o consists of the nodes with similarity to a concept label bounded by β (line 1). The edge set E_o is also constructed accordingly (line 2). It then checks the condition that whether for each edge (v_{o_1}, v_{o_2}) of G_o, each node in v_{o_1} (resp. v_{o_2}) has a child in v_{o_2} (resp. parent in v_{o_1}) (line 4). If not, it invokes a procedure SplitMerge (omitted) to refine V_o by splitting and merging the node v_{o_1} (resp. v_{o_2}) to make the condition satisfied (line 5). The graph G_o is updated accordingly with the new node and edge set (line 6). The refinement process repeats until a fixpoint is reached, and G_o is returned as a concept graph (line 7).

Example IV.2: Recall the data graph G_c and ontology graph O_{g_c} in Fig. 3. To compute a concept graph of G_c, The algorithm OntoIdx first generates a set of concept labels as {red, blue, green} (line 2). It then invokes CGraph to construct a node partition of G as the node set of G_c (line 4), and generates G_{c_0} as shown in Fig. 7. Each node and edge is refined according to the definition of the concept graph (lines 3-6). For example, the node (green, lime, olive) labeled with green is split into two nodes (green, lime) and olive by invoking procedure SplitMerge (line 5), which updates G_{c_0} to G_{c_1} (Fig 7). Similarly, CGraph (1) splits the node (blue, sky, violet) into (blue, sky) and violet, and updates G_{c_1} to G_{c_2}, and (2) splits the node (pink, rose, flame) to produce G'_c (Fig. 5) as the final concept graph. □

Correctness and Complexity. The algorithm CGraph correctly computes a set of concept graphs as the ontology index. For the complexity, (a) the concept labels can be selected in $O(|\mathsf{O}|)$ time; (b) the time complexity of SplitMerge and CGraph is $O(|V|+|E|)$ and $O(|E|\log|V|)$, respectively; and (c) the procedure CGraph is invoked at most N times (lines 4-5). Thus, the total time complexity of OntoIdx is $O(N * |E|\log|V|)$. As N is typically small comparing with $|V|$ and $|E|$, the overall complexity of OntoIdx is thus $O(|E|\log|V|)$. The above analysis also completes the proof of Proposition 4.1.

B. Ontology-based filtering

In this section, we illustrate the filtering phase of the query evaluation framework based on the ontology index. As remarked earlier, instead of performing subgraph isomorphism directly over the data graph G, we extract a (typically small) subgraph of G that contains all the matches of the query. Ideally, one wants to identify the minimum subgraph which is simply the union of all its matches. Nevertheless, to find such an optimal subgraph is already NP-complete [16].

Instead, we use ontology index to efficiently reduce the nodes and edges that are not in any matches as much as possible, and extract a subgraph G_v of G, which is induced by a relation M between the query nodes in a query Q and the nodes in a concept graphs G_c. The relation M is a relaxation of the subgraph isomorphism which guarantees the following condition: (1) for each query node u in Q and its matches v (if any), v is in a node $M(u)$ in G_c, (2) for each query node u and each edge (u, u_1) (resp. (u_2, u)) in Q, $(M(u), M(u_1))$ (resp. $(M(u_2), M(u))$ is an edge in G_c. The subgraph G_v is extracted from G_c by "collapsing" $M(u)$ for each query node u to a set of corresponding data nodes in G. If multiple matching relations are computed from a set of concept graphs in $\mathcal{I}$, for each query node u, $M(u)$ is refined as $\bigcap M_i(u)$, where $M_i(u)$ is collected from G_{c_i} in the ontology index.

The following result shows the relationship between the subgraph G_v and ontology index.

Proposition 4.2: *Given an ontology index $\mathcal{I}$, a query graph Q and a data graph G, if the subgraph G_v is empty, then $\mathsf{Q}(G)$ is empty; otherwise, $\mathsf{Q}(G) = \mathsf{Q}(G_v)$, and (2) G_v can be computed in $O(|\mathsf{Q}||\mathcal{I}|)$ time, where $|\mathcal{I}|$ (resp. $|\mathsf{Q}|$) is the total number of nodes and edges in $\mathcal{I}$ (resp. $|\mathsf{Q}|$).* □

To see Proposition 4.2 (1), observe that if Q has a match G' induced by an ontology-based isomorphism mapping h, then a relation M can be constructed such that for any query node u, $h(u) \in M(u)$. Thus, G_v contains all the matches of Q, and $\mathsf{Q}(G) = \mathsf{Q}(G_v)$. On the other hand, if G_v is empty, then no match exists in G for Q and $\mathsf{Q}(G)$ is empty, since no relation M exists even as a relaxation of subgraph isomorphism.

To prove Proposition 4.2 (2), we introduce an algorithm, denoted as Gview, to generate G_v from $\mathcal{I}$ in $O(|\mathsf{Q}||\mathcal{I}|)$ time.

Algorithm. The algorithm Gview is illustrated in Fig. 8, which takes as input a query Q, data graph G and a user-defined similarity threshold θ. It has the following three steps.

Initialization (lines 1-2). For each query node v_q, it initializes a *match set* $\mathsf{mat}(v_q)$, to record the final matches identified by the matching relation M (as remarked earlier), as well as the candidate set $\mathsf{can}(v_q)$ (line 2) to keep track of the matches when a single concept graph is processed.

Matching and refinement (lines 3-10). Gview computes the relation M as follows. It first initializes the candidate set $\mathsf{can}(v_q)$ for each query node v_q, using a "lazy" strategy (as will be discussed) (line 4). It then conducts a fixpoint computation (lines 5-7), by checking if there exists a query edge (v_{q_1}, v_{q_2}), such that there is a node $v_{o_1} \in \mathsf{can}(v_{q_1})$ which has no child in $\mathsf{can}(v_{q_2})$. If so, v_{q_1} (and all the data nodes contained in it) is no longer a candidate for v_q. Gview thus removes v_{q_1} from $\mathsf{can}(v_{q_1})$ (line 6). If $\mathsf{can}(v_{q_1})$ is empty, then query node q_1 has no valid candidate in some concept graph, and Gview returns $\emptyset$ (line 7). Otherwise, $\mathsf{mat}(v_{q_1})$ is refined by $\mathsf{can}(v_{q_1})$: if $\mathsf{mat}(v_{q_1})$ is empty, it is initialized with $\mathsf{can}(v_{q_1})$ (line 8);

Input: query $Q = (V_q, E_q, L_q)$, ontology index $\mathcal{I}$, similarity threshold θ;
Output: a subgraph G_v.

1. set $V_{q_v} := \emptyset$, $E_{qv} := \emptyset$;
2. **for each** $v_q \in V_q$ **do** set $\mathsf{mat}(v_q) := \emptyset$; $\mathsf{can}(v_q) := \emptyset$;
3. **for each** $G_o \in \mathcal{I}$ **do**
4. **for each** $v_q \in V_q$ **do** compute $\mathsf{can}(v_q)$ with lazy strategy;
5. **while** there is an edge $(v_{q_1}, v_{q_2}) \in E_q$ and $v_{o1} \in \mathsf{can}(v_{q_1})$ such that $C(v_{o1}, G_o) \cap \mathsf{can}(v_{q_2}) = \emptyset$ **then**
6. $\mathsf{can}(v_{q_1}) := \mathsf{can}(v_{q_1}) \setminus \{v_{o1}\}$;
7. **if** $\mathsf{can}(v_{q_1}) = \emptyset$ **then return** $\emptyset$;
8. **if** $\mathsf{mat}(v_q) = \emptyset$ **then** $\mathsf{mat}(v_q) := \mathsf{can}(v_q)$;
9. **else** $\mathsf{mat}(v_q) := \mathsf{mat}(v_q) \cap \mathsf{can}(v_q)$;
10. **if** $\mathsf{mat}(v_q) = \emptyset$ **then return** $\emptyset$;
11. **for each** $v_q \in V_q$ **do**
12. construct V_{q_v} and E_{q_v} with $\mathsf{mat}(v_q)$ and G_o;
13. construct $G_v := (V_{q_v}, E_{q_v}, L_{q_v})$;
14. **return** G_v;

Fig. 8. Algorithm Gview

otherwise, $\mathsf{mat}(v_{q_1})$ only keeps those candidates that are in $\mathsf{can}(v_{q_1})$ (line 9). If $\mathsf{mat}(v_{q_1})$ becomes empty, no candidate can be find in G for v_{q_1}, and Gview returns $\emptyset$ (line 10).

G_v construction (lines 11-14). After all the concept graphs in $\mathcal{I}$ are processed, Gview constructs G_v with a node set V_{q_v}, which contains a node for each match set, and a corresponding edge set E_{q_v} (lines 11-13). G_v is then returned (lines 14).

It is costly to identify the candidates for the query nodes in Q by accessing the ontology graph O and G, which may take up to $O(|\mathsf{Q}||G|)$ time. Instead of identifying the candidates for a query node v_q and the user-defined similarity threshold θ, a "lazy" strategy (line 4) only identifies a set of nodes (as $\mathsf{can}(v_q)$) in the concept graph G_o, such that the candidates of v_q are contained in these nodes. To this end, it simply selects the nodes in G_o labeled with the concept labels l, where the distance of l and the label of v_q in O is less than $\mathsf{sim}^{-1}(\theta) + \mathsf{sim}^{-1}(\beta)$. Here β is the similarity threshold adopted to generate G_o. One may verify that each candidate of v_q *w.r.t.* the similarity threshold θ is in one of such nodes, since the similarity function sim() is a monotonically decreasing function of the label distances in O (Section II). Moreover, the total candidate selection time is reduced to as $O(|\mathsf{Q}||\mathsf{O}|)$. Note that $|\mathsf{Q}|$ and $|\mathsf{O}|$ are typically small comparing to $|G|$.

Example IV.3: Recall the query Q in Example. I.1. Using the ontology index $\mathcal{I} = \{ G_{o_1}\ G_{o_2} \}$ (Fig. 5), Gview extracts G_v as follows. (1) Using G_{o_1}, Gview initializes $\mathsf{can}(\mathsf{moonlight})$ with the node moonlight in G_{o_1}, and similarly initializes $\mathsf{can}(\mathsf{museum})$ and $\mathsf{can}(\mathsf{tourists})$ (line 4). For *e.g.,* query edge $\{\mathsf{tourist}, \mathsf{moonlight}\}$, Gview refines $\mathsf{can}(\mathsf{tourists})$ by checking if every node in $\mathsf{can}(\mathsf{tourists})$ has a child in $\mathsf{can}(\mathsf{moonlight})$ (line 5-10). After the refinement, $\mathsf{mat}(\mathsf{moonlight}) = \{\mathsf{HC}, \mathsf{riverside}\}$, $\mathsf{mat}(\mathsf{museum}) = \{\mathsf{Disneyland}, \mathsf{RG}\}$ and $\mathsf{mat}(\mathsf{tourists}) = \{\mathsf{HT}, \mathsf{CT}\}$. (2) Using G_{o_2}, Gview identifies that $\mathsf{can}(\mathsf{tourists}) = \{\mathsf{CT}\}$, $\mathsf{can}(\mathsf{museum}) = \{\mathsf{RG}\}$, and $\mathsf{can}(\mathsf{moonlight}) = \{\mathsf{riverside}, \mathsf{RP}\}$. (3) Putting these together, the final match sets $\mathsf{mat}(moonlight) = \{\mathsf{riverside}\}$, $\mathsf{mat}(\mathsf{tourists}) = \{\mathsf{CT}\} \cap \{\mathsf{HT}, \mathsf{CT}\} = \mathsf{CT}$, and $\mathsf{mat}(\mathsf{museum}) = \{\mathsf{RG}\} \cap \{\mathsf{Disneyland}, \mathsf{RG}\} = \{\mathsf{RG}\}$. G_v (as shown in Fig. 9) is then constructed as the subgraph of G induced by the nodes riverside, CT, and RG (lines 11-14). □

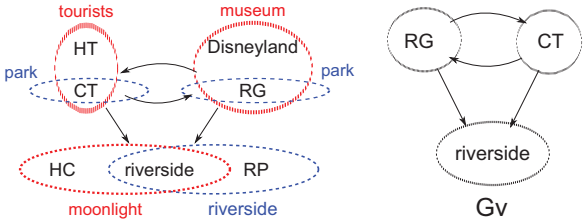


Fig. 9. Generating subgraph G_v. The small subgraph G_v is constructed by iteratively computing and intersecting the corresponding nodes in concept graphs for each query node.

Correctness and complexity. The algorithm Gview correctly computes a subgraph G_v. To see this, observe that (1) G_v is initialized using the lazy strategy contains all the possible matches (line 4); (2) for each query edge (v_{q_1}, v_{q_2}), Gview uses $\mathsf{can}(v_{q_2})$ to refine $\mathsf{can}(v_{q_1})$ in each concept graph, and only removes those nodes that are not matches (non-matches) for v_{q_1} (lines 5-6); and (3) if $\mathsf{can}(v_q)$ is empty when processing a concept graph, then there is no match in G for v_q (line 7,10). Since if there indeed exists a data node v that can match v_q, then for every query edge (v_q, v'_q), there must exist a corresponding edge (v_o, v'_o) ($v \in v_o$) in *every* concept graph. Thus, Gview only removes non-matches of Q from the initialized G_v. The correctness of Gview thus follows.

For the complexity, (1) it takes $O(|V_q||C|)$ to identify the candidates for the query nodes (lines 3-4) using lazy strategy, where $|C|$ is the total number of concept labels . The filtering process (lines 5-10) can be implemented in time $O(|E_q||\mathcal{I}|)$. The construction of G_v is in time $O(|\mathcal{I}|)$. Putting these together, the total time of Gview is in $O(|E_q||\mathcal{I}|)$. In practice $|E_q|$ is typically small, and the complexity of Gview can be considered as near-linear *w.r.t.* $|\mathcal{I}|$.

V. Subgraph Query Processing

In the verification phase, the framework performs the subgraph isomorphism tests over the subgraph extracted from the ontology index. We provide a global ontology-based subgraph querying algorithm for the top K matches problem. The algorithm, denoted as KMatch, is as shown in Fig. 10.

Algorithm. Upon receiving a query Q, the algorithm KMatch first extracts the subgraph G_v by invoking the procedure Gview (line 3) (see Section IV), For each query node v_q, it constructs a candidate list $\mathcal{L}(v_q)$, sorted in the descending order of the similarity (line 6-7). KMatch then iteratively constructs a subgraph G_s using the candidates with the largest similarity from the candidate lists, and if G_s is a match, it inserts G_s to a heap H (lines 10-12). The above process repeats until all such G_s is processed (line 10), or H contains the top K matches with maximum similarity scores (line 8).

It takes $O(|\mathsf{Q}||\mathcal{I}|)$ time to compute G_v (line 3), as remarked earlier. The total time of KMatch is thus in $(|\mathsf{Q}||\mathcal{I}| + |G_v|^{|V_q|})$. As verified in our experiment, in practice G_v is significantly smaller than G (see Section VII).

Input: query $\mathsf{Q} = (V_q, E_q, L_q)$, ontology graph O, ontology index $\mathcal{I}$, data graph G;
Output: a set of top-K matches for Q in G *w.r.t.* O.

1. set $\mathsf{Match} = \emptyset$;
2. /* filtering */
3. extract a subgraph $G_v := \mathsf{Gview}(\mathsf{Q}, \mathcal{I})$;
4. /* verification */
5. heap $H = \emptyset$;
6. **for each** $v_q \in V_q$ **do**
7. construct a sorted candidate node list $\mathcal{L}(v_q)$ in G_v;
8. **while** $|H| \leq K$ **do**
9. construct a distinct node list L' with $\mathcal{L}()$ maximizing the overall similarity;
10. **if** $L' = \emptyset$ **then break** ;
11. construct G_s induced by the data nodes in L';
12. **if** G_s is a match of Q **then** insert G_s to H;
13. **return** H;

Fig. 10. Algorithm KMatch

Remarks. The ontology-based subgraph querying framework can be easily adapted to support traditional subgraph isomorphism. Indeed, when the user-defined similarity threshold is 1.0, (1) the ontology index can be used to extract a subgraph G_v, which only contains the candidate nodes with identical labels for the query nodes, and (2) any match extracted from G_v is a subgraph isomorphic to Q in terms of identical subgraph isomorphism.

VI. Ontology Index Maintenance

The ontology-based subgraph querying framework can efficiently extracts a compact subgraph of a data graph from the ontology index, which is then queried for verification and results generation. In practice the data graphs are changing frequently over time. In this section we investigate the incremental maintenance of the ontology index, which further enables the ontology-based subgraph querying to cope with dynamic data graphs. Indeed, a dynamic subgraph querying framework can be readily developed by incrementally updating the ontology index, and then performs the filtering and verification phases to compute the new matches.

Given a set of updates ΔG to the data graph G, and an ontology index $\mathcal{I}$, the index maintenance is to update $\mathcal{I}$ to an ontology index for the updated data graph $G \oplus \Delta G$. Instead of recomputing the concept graphs from scratch each time the data graph is updated, we show that the index $\mathcal{I}$ can be directly updated by only accessing ΔG.

As observed in [27], it is no longer adequate to measure the complexity of incremental algorithms by using the traditional complexity analysis for batch algorithms. Following [27], we characterize the complexity of an incremental graph algorithm in terms of the size of the *affected area (*AFF*)*, which indicates the changes in the input ΔG and the $\Delta\mathcal{I}$, *i.e.,* $|\mathsf{AFF}| = |\Delta G| + |\Delta\mathcal{I}|$. Specifically, $\Delta\mathcal{I}$ contains the nodes and edges that are not shared by $\mathcal{I}$ and $\mathcal{I}'$ as the updated $\mathcal{I}$.

Proposition 6.1: *Given a data graph G and a set of updates ΔG (edge insertions and deletions), the ontology index $\mathcal{I}$ can be maintained in $O(|\mathsf{AFF}|^2 + |\mathcal{I}|)$ time.* □

Input: A graph G, ontology index $\mathcal{I}$, batch updates ΔG;
Output: An updated $\mathcal{I}$.

1. **for each** $e \in \Delta G$ and each $G_o \in \mathcal{I}$ **do**
2. **if** $e = (u, v)$ is an edge insertion **then**
3. $\mathsf{incIdx}^+(e, G_o)$;
4. **else** $\mathsf{incIdx}^-(e, G_o)$;
5. **return** $\mathcal{I}$;

Procedure incIdx^+
Input: a concept graph $G_o = (V_o, E_o, f_o)$, an edge insertion (u', u),
Output: An updated G_o.

1. set $\mathsf{AFF} = \emptyset$;
2. find $v_{o'_u}$ and v_{o_u}, where $u' \in v_{o'_u}$ and $u \in v_{o_u}$;
3. split v_{ou} to $v_{ou_1} := v_{ou} \setminus \{u\}$ and $v_{ou_2} := \{u\}$;
4. split v_{ov} to v_{ov_1} and v_{ov_2} similarity; update AFF;
5. **if** $\mathsf{mCondition}$ **then**
6. merge v_{ou_2} and v_{ov_2}; update AFF;
7. **for each** $v_o \in \mathsf{AFF}$ **do**
8. $\mathsf{propUp}(v_o, G_o)$; $\mathsf{propDown}(v_o, G_o)$;
9. **return** G_o;

Fig. 11. Algorithm incIdx

We next present an algorithm, denoted as incIdx, to update the ontology index upon changes to the data graph G.

Algorithm. The algorithm incIdx is shown in Fig. 11. Given a set of updates ΔG (edge insertions and deletions), the algorithm processes each update $e \in \Delta G$ for each $G_o \in \mathcal{I}$. If e is an edge insertion, incIdx invokes procedure incIdx^+ to update G_o (line 3); otherwise, it invokes incIdx^- to process e (line 4). After all the updates are processed in each concept graph G_o, it returns the updated index $\mathcal{I}$.

Edge insertions. Procedure incIdx^+ takes as input a concept graph G_o and an edge insertion (u', u), and update G_o by taking a *split-merge-propagation* strategy as follows. It first initializes a set AFF to record the nodes and edges that need to be updated (line 1), and identifies $v_{o'_u}$ and v_{o_u} in G_o that contains u' and u, respectively (line 2). It then separates u' from $v_{o'_u}$, and splits v_{o_u} similarly (lines 3-4), since $v_{o'_u}$ and v_{o_u} violates the structural constraints of a concept graph due to the insertion. The set AFF is updated accordingly by adding the newly formed nodes (line 4). It then checks if the nodes u' and u can be merged with other nodes in G_o, due to sharing the common children and parents, *i.e.,* the merge condition $\mathsf{mCondition}$, and if so, merges u' or u with other nodes and update AFF (lines 5-6). For each affected node in AFF, incIdx^+ then *propagates* the changes to its ancestors and descendants, by invoking procedures propUp and $\mathsf{propDown}$ (omitted), following the same split-merge strategy until AFF is empty (lines 7-8). It finally returns the updated G_o (line 9).

Edge deletions. Procedure incIdx^- (not shown) processes edge deletion and updates the index similar as incIdx^+. After processing the changes directly caused by the edge deletion, it propagates the changes, following the same split-merge-propagation strategy.

Example VI.1: Consider the data graph G and the ontology index $\mathcal{I} = \{G_{o_1}, G_{o_2}\}$ in Fig. 1 and Fig. 3. Suppose the edges $e_1 = (\mathsf{HC}, \mathsf{riverside})$ is inserted to G. Upon the insertions of

e_1, IncIdx splits the node {HC, riverside} into {HC} and {riverside}, which are added to AFF (lines 3-4). It then propagates the changes and splits the node {HP, RP} in G_{o_1}. The updated concept graph G'_{o_1} contains six nodes, with two newly separated nodes as remarked earlier. On the other hand, no change is incurred to G_{o_2}. The updated $\mathcal{I}$ thus contains G'_{o_1} and G_{o_2}. The total AFF includes the nodes {HC, riverside}, {HP, RP} in G_{o_1}, and the newly separated nodes in G'_{o_1}.

Now suppose edge e_2 = (HT, riverside) is inserted into G. Similarly, one may verify that the insertion of e_2, while does not affect G_{o_1}, changes G_{o_2} by splitting its node {RP, riverside}. The affected area AFF includes the node {RP, riverside} and the two nodes RP and riverside. □

Analysis. One may verify that both IncIdx$^+$ and IncIdx$^-$ preserve the following invariants: each split, merge and propagate operation do not introduce nodes and edges that violates the node and topological constraints of concept graphs. The correctness of IncIdx thus follows. The complexity of IncIdx is in $O(|\mathsf{AFF}|^2 + |\mathcal{I}|)$. To see this, for IncIdx$^+$, it takes $O(|\mathcal{I}|)$ time to perform the split and merge operation, and the propagation propUp and propDown takes $O(|\mathsf{AFF}|^2)$ time as fixpoint computation. Similarly, the complexity of procedureincIdx$^-$ is in $O(|\mathsf{AFF}|^2 + |\mathcal{I}|)$. Thus, the total complexity of IncIdx is in $O(|\mathsf{AFF}|^2 + |\mathcal{I}|)$. As verified in our experiments, |AFF| is typically small and the index can be efficiently updated.

VII. Experimental Evaluation

We next present an experimental study using both real-life and synthetic data. We conducted three sets of experiments to evaluate: (1) the effectiveness of the ontology-based subgraph querying, (2) the efficiency of the query evaluation framework, and (3) the performance and cost of the ontology index.

Datasets. We used the following datasets.

(1) Real-life graphs. We used the following two real-life datasets, each consists of a data graph and an ontology graph. (a) CrossDomain is taken from a benchmark suite FebBench [29], which consists of (i) an RDF data graph with 1.07M nodes and 3.86M edges where nodes represent entities from different domains (*e.g.,* Wikipedia, locations, biology, music, newspapers), and edges represent the relationship between the entities (*e.g.,* born in, locate at, favors); and (ii) an ontology graph with 1.44M concepts and 5.30M relations. The data graph takes in total 150Mb physical memory. (b) Flickr contains a data graph taken from `http://press.liacs.nl/mirflickr/` with 1.3M nodes and 6.42M edges, where the nodes represent images, tags, users or locations, and edges represent their relationship. It also contains an ontology graph from DBpedia (`http://dbpedia.org`) with more than 3.64 million entities. The data graph takes in total 194Mb physical memory. In our experiments, we employ the ontology graph to describe the tags in Flickr.

(2) Synthetic data. We designed a graph generator to produce randomly generated synthetic graphs, which was controlled by three parameters: the number of nodes $|V|$, the number of edges $|E|$, and the size $|L|$ of the node label set. We also generate ontology graphs for the set of synthetic graphs sharing the same set of label L, controlled by the same set of parameters. We use $(|V|, |E|)$ to denote the size of a data graph, and the ontology graph.

Following [19], we set the similarity function as $\mathsf{sim}(l', l) = 0.9^{\mathsf{dist}(l', l)}$ for all the ontology graphs O, where $\mathsf{dist}(l', l)$ is the distance between two nodes l' and l in O. For example, if a label l is 2 hops away from l' in O, $\mathsf{sim}(l', l) = 0.81$.

Implementation. We implemented the following algorithms in C++: (1) algorithm OntoIdx; (2) algorithm KMatch; (3) SubIso, the subgraph isomorphism algorithm in [32], which identifies the matches using identical label matching; (4) SubIso$_r$, which, as a comparison to KMatch, is revised from [32] that rewrites the query graph, and directly computes all the matches and select the best ones; (4) VF2, which computes the minimum weighted matches, by exploiting a *similarity matrix* between the query label and all the labels in the data nodes; (4) our incremental algorithm IncIdx.

To favor VF2, we precomputed a similarity matrix, where each entry records $\mathsf{sim}(u, v)$ as the similarity between a query node u and a data node v *w.r.t.* the ontology graph O. We also optimized VF2 such that it terminates as soon as the top K matches are identified. The time cost of computing the similarity matrix is not counted for VF2.

We used a machine powered by an Intel(R) Core 2.8GHz CPU and 8GB of RAM, using Ubuntu 10.10. Each experiment was run 5 times and the average is reported here.

Experimental results. We next present our findings.

Exp-1: Effectiveness and flexibility. In this set of experiments, we first evaluated the effectiveness of KMatch and SubIso. We generated 5 query templates for CrossDomain, and 4 query templates for Flickr. We use $(|V_p|, |E_p|, |L_p|)$ to denote the size of a query $\mathsf{Q}(V_p, E_p, L_p)$. For CrossDomain, (1) Q_{T_1} is a tree of size $(4, 3, 3)$ searching for movies, directors and distributors, and Q_{T_2} of size $(4, 4, 3)$ is a cycle obtained by inserting an edge to Q_{T_1}; (2) Q_{T_3} of size $(4, 6, 4)$ is to search pop stars, record companies, albums and songs, and Q_4 is obtained by only "generalizing" the query label of Q_{T_3}, *e.g.,* from "Green Record Company" to "Record Company"; and (3) Q_{T_5} of size $(5, 6, 4)$ is to identify the soccer stars, clubs and their teammates. Similarly, for Flickr the 4 queries Q_{T_6} to Q_{T_9} are to identify photos of animals taken at specified locations. Each template Q_{T_i} is populated as a query set of 100 queries (also denoted as Q_{T_i}) by varying the node labels only. For ontology index, we employ the graph partitioning algorithm in [28] to generate concept labels with similarity threshold $\beta = 0.8$, unless otherwise specified.

Effectiveness. We first compared the number of matches found by SubIso and KMatch over CrossDomain and Flickr, as shown in Table I. Fixing card $\mathcal{I} = 1$, *i.e.,* the ontology index $(\mathcal{I})$ contains a single concept graph, we varied the similarity threshold of the queries from 1.0 to 0.8, and identify *all* the matches. For all the queries over CrossDomain, SubIso only

Query	CrossDomain		
	θ=1	θ=0.9	θ=0.8
Q_{T_1}	1	2,687	9,099
Q_{T_2}	0	24	271
Q_{T_3}	1	170	342
Q_{T_4}	0	405	991
Q_{T_5}	0	30,854	48,225

Query	Flickr		
	θ=1	θ=0.9	θ=0.8
Q_{T_6}	2	6	307
Q_{T_7}	0	177	2,160
Q_{T_8}	0	448	6,028
Q_{T_9}	0	799	15,052

TABLE I

EFFECTIVENESS OVER REAL LIFE GRAPHS

finds in average 1 exact match for query set Q_{T_1} and Q_{T_3}, and no match for all other queries. In contrast, KMatch identifies much more matches that are semantically close to the query according to our observation. It also finds more meaningful matches than SubIso over Flickr.

Two sample patterns and their closest matches are shown in Fig. 13. (1) Query Q_2 in Q_{T_2} (Fig. 13(a)) over CrossDomain is to find two movies distributed by Walt Disney and directed by James Cameron, where one is screened out of competition at Cannes Film Festival, and the other is related with Aliens. The closest match is shown in Fig. 13(b) where Aliens is matched to the movie Aliens of the Deep, and Cannes Film Festival has a match Ghosts of the Abyss. (2) Query Q_3 (Fig. 13(c)) of Flickr is to identify two photos both related with "Flamingo" with color "Pink", and one is taken in San Diego while the other in Miami. The closest match is given in Fig. 13(d) where Miami is matched to "Seaworld" in Florida.

The algorithm VF2, via carefully processed similarity matrix, identifies the same set of matches as KMatch (thus is not shown) with much more running time, as will be shown.

Query flexibility. As shown in Table I, (a) for all the queries, the match number increases when the similarity threshold θ decreases, since more data nodes become candidates and more subgraphs become matches; (b) fixing node number and labels, the insertion of edges increases the topological complexity of the query, *e.g.,* from Q_1 to Q_2, and thus, reduces the number of matches; and (c) fixing the structure, the query label generalization (from *e.g.,* Q_3 to Q_4) increases the candidates of the query nodes, which in turn increases the match number.

Exp-2: Efficiency and scalability. We evaluated the performance of KMatch, SubIso$_r$ and VF2 using real-life datasets and synthetic data, and their scalability using synthetic data. In these experiments, the indexes were precomputed, and thus their construction time were not counted.

Real life graphs. Fig. 12(a) and Fig. 12(b) (both in log scale) show the running time of KMatch and VF2 for evaluating Q_{T_1} to Q_{T_5} over CrossDomain in Table I. The results tells us the following. (1) KMatch *always* outperform VF2. For example, KMatch takes only 1% of the running time of VF2 to evaluate Q_{T_2}. When θ = 0.9 (resp. θ = 0.8), KMatch takes 30% (resp. 22%) of the running time of VF2 in average for all the queries. (2) When θ decreases, both algorithms takes more time due to more candidates. In addition, KMatch improves the efficiency of VF2 better for larger θ due to the filtering power of the ontology index even with only a single concept graph.

To evaluate the scalability with card($\mathcal{I}$), *i.e.,* the number of concept graphs, we used CrossDomain, varied card($\mathcal{I}$) from 1 to 7, and tested the cases where θ is 0.9 and 0.8, respectively. The results, shown in Fig. 12(c), tells us that the running time of KMatch, decreases while card($\mathcal{I}$) increases. Specifically, when θ = 0.8, the verification (resp. filtering) time decreases (resp. increases) from 396 (resp. 2) seconds to 110 (resp. 30) seconds when card($\mathcal{I}$) increases from 1 to 4, and the total time decreases from 398 seconds to 168 seconds. The total time increases when card($\mathcal{I}$) is increased from 4 to 7. This is because (a) more concept graphs effectively filter more candidates, and reduce the verification time, and (b) when card($\mathcal{I}$) $>$ 4, while the index spends more time in filtering phase, it cannot further reduce the verification time, thus the total time increases. Similarly, the running time of KMatch decreases when θ= 0.9 and card($\mathcal{I}$) $<$ 3.

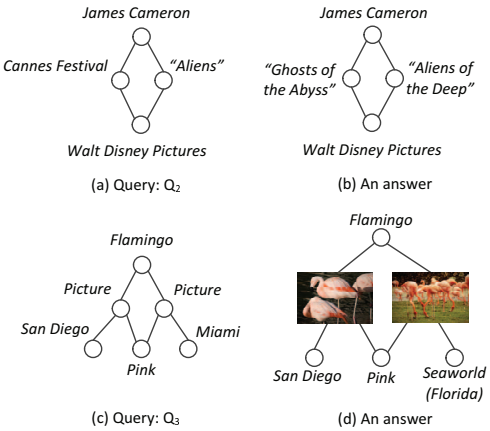


Fig. 13. Sample queries and matches

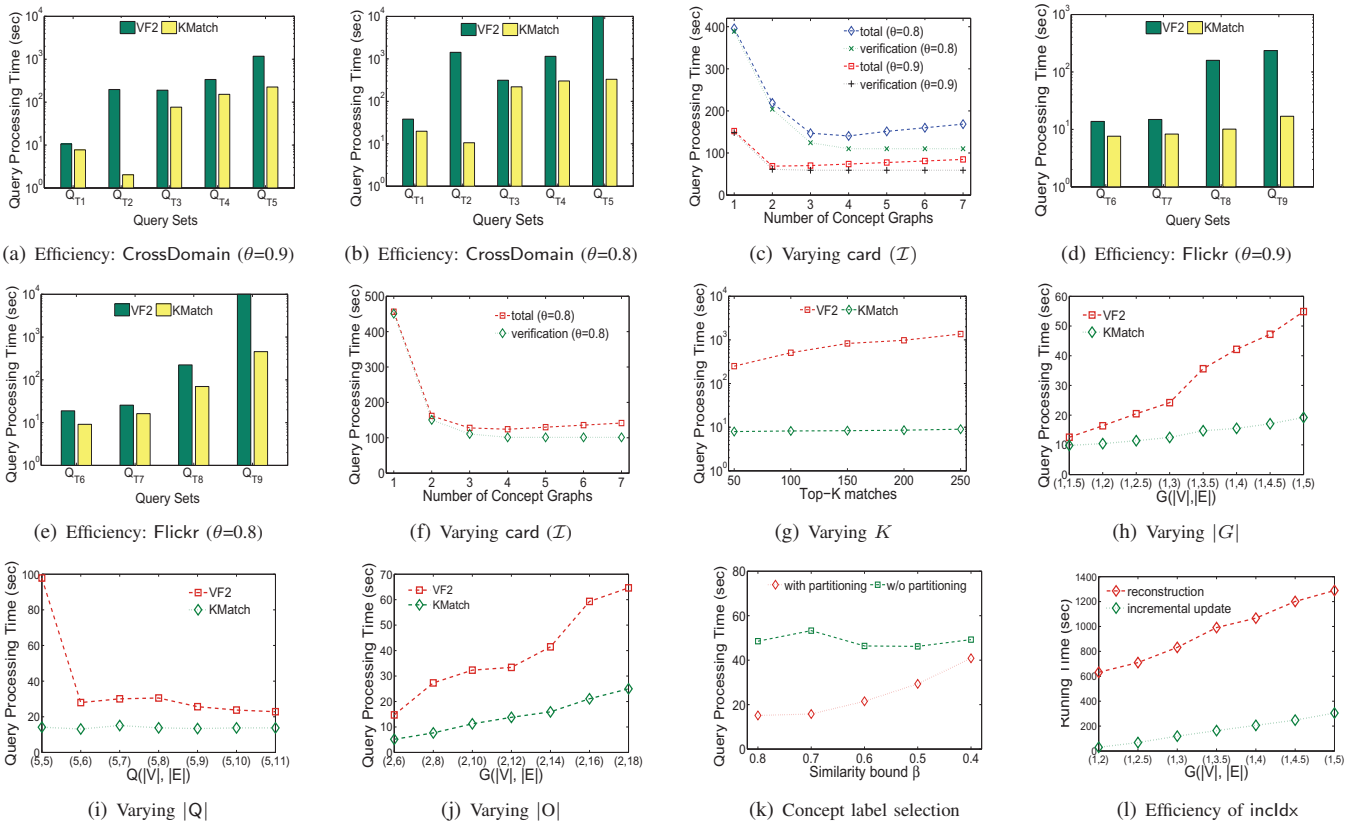
The efficiency of KMatch and VF2 over Flickr is given in Fig. 12(d), Fig. 12(e), and Fig. 12(f), which verify the results of their CrossDomain counterparts Fig. 12(a), Fig. 12(b), and Fig. 12(c), respectively. In average, the running time of KMatch is 30% of that of VF2 over Flickr when θ = 0.9. When θ = 0.8, VF2 does not run to complete for Q_{T_4}.

To evaluate the impact of K in finding the top K matches, We evaluated the efficiency of KMatch and VF2 by varying K from 50 to 250, and used query set Q_{T_2} over Flickr. The result is as shown in Fig. 12(g). It takes more time for KMatch and VF2 to identify K best matches when K is increasing, as expected. Moreover, the performance of KMatch is less sensitive than that of VF2. This is because KMatch extracts all the matches from a small subgraph after filtering phase, while VF2 needs to run isomorphism test over G to identify each new match. The tests with other queries verify our observation.

The algorithm SubIso$_r$ does not scale even over small queries such as Q_{T_2}, thus its result is not reported.

Synthetic graphs. Using synthetic graphs, we provide an in-depth analysis of the efficiency and scalability of KMatch and VF2. We fixed card($\mathcal{I}$) = 1, the user-defined similarity threshold θ = 0.8, and the similarity threshold β = 0.8. We construct random query templates populated with 100 queries, and the average result is reported.

We first evaluate the scalability of KMatch and VF2 with $|G|$. Fixing the size of an ontology graph |O| as $(2K, 12K, 2K)$, and query size |Q| = $(5, 8, 5)$, we varied



(a) Efficiency: CrossDomain (θ=0.9) (b) Efficiency: CrossDomain (θ=0.8) (c) Varying card ($\mathcal{I}$) (d) Efficiency: Flickr (θ=0.9)

(e) Efficiency: Flickr (θ=0.8) (f) Varying card ($\mathcal{I}$) (g) Varying K (h) Varying $|G|$

(i) Varying |Q| (j) Varying |O| (k) Concept label selection (l) Efficiency of incIdx

Fig. 12. Ontology-based subgraph querying: performance evaluation

$|G|$ from $(1M, 1.5M, 2K)$ to $(1M, 5M, 2K)$, corresponding to $(1, 1.5)$ to $(1, 5)$ in Fig. 12(h). Fig. 12(h) tells us that (a) both algorithms takes more time when $|G|$ increases, (b) KMatch*always* outperforms VF2: it takes running time up to 25% of that of VF2, over all the data graphs, and takes less than 14 seconds to identify *all* the matches in G with size $(1M, 5M, 2K)$; and (c) KMatch is less sensitive than VF2.

To evaluate the impact of query size on the performance of KMatch, we varied |Q| from $(5, 5, 5)$ to $(5, 11, 5)$, while fixing $|G| = (1M, 3.5M, 2K)$ and |O| = $(2K, 12K, 2K)$. Fig. 12(i) shows that (a) both algorithm take less time for larger queries, since for both algorithms, the more complex the queries are, the better their filtering strategies work, and (b) KMatch is less sensitive. This is because for queries with more edges, KMatch takes more time in filtering but less time in verification.

To evaluate the scalability with the size of O, we varied |O| from $(2K, 6K, 2K)$ to $(2K, 18K, 2K)$, by inserting edges in $2K$ increments, while fixing $|G| = (1M, 3.5M, 2K)$ and |Q| = $(5, 8, 5)$. Fig. 12(j) shows that both KMatch and VF2 take more time for larger O. This is because the average candidate numbers are increased due to larger ontology graphs.

Exp-3: Effectiveness of ontology index. Using real life data, we next investigate (1) ctime, *i.e.,* the running time of algorithm Ontoldx; (2) the compression rate cr = $\frac{|E_o|}{|E|}$, where $|E_o|$ is the average edge size in $\mathcal{I}$, and $|E|$ is the edge size of the data graph, (3) the memory reduction mr = $\frac{|M_o|}{|M|}$, where $|M_o|$ and M are the physical memory cost of $\mathcal{I}$ and the data graph, respectively; and (4) the filtering rate fr = $\frac{|G_v|}{|G_{sub}|}$, where $|G_v|$ is the average size of the induced subgraphs G_v in filtering phase, and $|G_{sub}|$ is the size of all the nodes and edges visited by VF2. We fixed card $\mathcal{I}$ = 1, and β = 0.8. The result is shown below.

Dataset	ctime	cr	mr	fr
CrossDomain	694s	0.43	0.51	0.06
Flickr	383.83s	0.71	0.52	0.24

The above results tell us the following. (1) For both data sets, the efficiency of Ontoldx is comparable to that of VF2 for processing a single query (see Exp-2). (2) $\mathcal{I}$ contains much less nodes and edges over the data graph, and takes only half of of its physical memory cost. (3) Even when only a single concept graph is used, the index effectively filters the search space. Indeed, the size of G_v for verification is only 6% and 24% of $|G_{sub}|$ over CrossDomain and Flickr, respectively.

Concept label selection. We compared the performance of KMatch over CrossDomain using different concept label selection strategies. Fixing $|\mathcal{I}|$ = 1, we generated concept labels by varying β from 0.8 to 0.4, and by using (a) partitioning strategy, and (b) random selection without partitioning. The results are as shown in Fig. 12(k), which tell us (1) KMatch takes more time when β decreases, due to that the concept graphs are too "abstract" to perform effective filtering with less concept labels, since more candidates are merged as a single node; and (2) the partitioning strategy improves the efficiency of KMatch by up to 70% due to a better filtering process. The results using other queries also verify our observation.

Efficiency of incremental maintenance. We finally compare the performance of incIdx and OntoIdx upon data graph changes, where OntoIdx recomputes the index from scratch. Fixing β = 0.8, |O| =$(2K, 12K, 2K)$, and $|V| = 1M$, we varied $|E|$ from 2M to 5M by inserting edges in 0.5M increments. Fig. 12(l) tells us that incIdx greatly outperforms OntoIdx. The running time of incIdx is only 20% of that of OntoIdx even when $|E|$ is increased from 2M to 5M in a single batch of updates.

Summary. We find the following. (1) The ontology-based subgraph querying can efficiently identify more matches that are semantically close to the query, comparing with the traditional subgraph isomorphism. (2) Our query evaluation framework is more efficient than conventional subgraph querying, *e.g.,*VF2. (3) The ontology index improves the performance of ontology-based subgraph querying. Better still, it can be efficiently updated upon data graph changes.

VIII. Conclusion

We have proposed the ontology-based subgraph querying, based on a quantitative metric for the matches. These notions support finding matches that are semantically close to the query graphs. We have proposed a framework for finding the (top K) closest matches, via a filtering and verification strategy using ontology index. In addition, we have proposed an incremental algorithm to update indexes upon data graph changes. Our experimental study have verified that the framework is able to efficiently identify the matches, which cannot be found by conventional subgraph isomorphism and its extensions.

This work is a first step for subgraph querying with ontologies. We are evaluating our techniques over various real graphs with different ontology similarity metrics. Another topic is to extend the techniques for other types of graph queries.

Acknowledgement. This research was sponsored in part by NSF IIS-0954125 and by the Army Research Laboratory under cooperative agreements W911NF-09-2-0053. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notice herein.

References

[1] C. C. Aggarwal and H. Wang. A survey of clustering algorithms for graph data. In *Managing and Mining Graph Data*, pages 275–301. 2010.
[2] N. Aizenbud-Reshef, A. Barger, I. Guy, Y. Dubinsky, and S. Kremer-Davidson. Bon voyage: social travel planning in the enterprise. In *CSCW*, pages 819–828, 2012.
[3] B. Aleman-Meza, C. Halaschek-Wiener, S. S. Sahoo, A. P. Sheth, and I. B. Arpinar. Template based semantic similarity for security applications. In *ISI*, 2005.
[4] K. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor. Freebase: a collaboratively created graph database for structuring human knowledge. SIGMOD, 2008.
[5] S. Borgatti and M. Everett. The class of all regular equivalences: Algebraic structure and computation. *Social Networks*, 11(1):65 – 88, 1989.
[6] A. Cakmak and G. Ozsoyoglu. Taxonomy-superimposed graph mining. In *EDBT*, 2008.
[7] G. Cheng and Y. Qu. Term dependence on the semantic web. In *International Semantic Web Conference*, 2008.
[8] J. Cheng, Y. Ke, W. Ng, and A. Lu. Fg-index: towards verification-free query processing on graph databases. In *SIGMOD*, 2007.
[9] C. Choi, M. Cho, J. Choi, M. Hwang, J. Park, and P. Kim. Travel ontology for intelligent recommendation system. In *AMS*, 2009.
[10] T. Coffman, S. Greenblatt, and S. Marcus. Graph-based technologies for intelligence analysis. *Commun. ACM*, 47, 2004.
[11] O. Corby, R. Dieng-Kuntz, F. Gandon, and C. Faron-Zucker. Searching the semantic web: approximate query processing based on ontologies. *Intelligent Systems, IEEE*, 21(1):20 – 27, 2006.
[12] V. Cordì, P. Lombardi, M. Martelli, and V. Mascardi. An ontology-based similarity between sets of concepts. In *WOA*, pages 16–21, 2005.
[13] R. Dieng-Kuntz and O. Corby. Conceptual graphs for semantic web applications. In *ICCS*, volume 3596, pages 19–50, 2005.
[14] W. Fan, J. Li, S. Ma, H. Wang, and Y. Wu. Graph homomorphism revisited for graph matching. *PVLDB*, 3, 2010.
[15] B. Gallagher. Matching structure and semantics: A survey on graph-based pattern matching. *AAAI FS.*, 2006.
[16] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.
[17] H. He and A. K. Singh. Closure-tree: An index structure for graph queries. In *ICDE*, 2006.
[18] H. H. Hoang and A. M. Tjoa. The state of the art of ontology-based query systems: A comparison of existing approaches. In *In Proc. of ICOCI06*, 2006.
[19] R. Knappe, H. Bulskov, and T. Andreasen. Perspectives on ontology-based querying. *Int. J. Intell. Syst.*, 22(7):739–761, 2007.
[20] R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins. On semi-automated web taxonomy construction. In *WebDB*, 2001.
[21] E. Little, K. Sambhoos, and J. Llinas. Enhancing graph matching techniques with ontologies. In *Information Fusion*, pages 1–8, 2008.
[22] E. Mäkelä, E. Hyvönen, and S. Saarela. Ontogator - a semantic view-based search engine service for web applications. In *ISWC*, 2006.
[23] M. McPherson, L. Smith-Lovin, and J. M. Cook. Birds of a feather: Homophily in social networks. *Annual Review of Sociology*, 27:415–444, 2001.
[24] T. Milo and D. Suciu. Index structures for path expressions. In *ICDT*, 1999.
[25] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26:1367–1372, 2004.
[26] R. Paige and R. E. Tarjan. Three partition refinement algorithms. *SICOMP*, 16(6), 1987.
[27] G. Ramalingam and T. Reps. On the computational complexity of dynamic graph problems. *TCS*, 158(1-2), 1996.
[28] K. Schloegel, G. Karypis, and V. Kumar. Parallel multilevel algorithms for multi-constraint graph partitioning (distinguished paper). In *Euro-Par*, pages 296–310, 2000.
[29] M. Schmidt, O. Görlitz, P. Haase, G. Ladwig, A. Schwarte, and T. Tran. Fedbench: A benchmark suite for federated semantic data query processing. In *International Semantic Web Conference (1)*, 2011.
[30] H. Stuckenschmidt and M. Klein. Structure-based partitioning of large concept hierarchies. In *ISWC*, 2004.
[31] S. Tu, L. Tennakoon, M. O'Connor, R. Shankar, and A. Das. Using an integrated ontology and information model for querying and reasoning about phenotypes: The case of autism. In *AMIA Annual Symposium Proceedings*, volume 2008, page 727, 2008.
[32] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1), 1976.
[33] V. Vassilevska and R. Williams. Finding, minimizing, and counting weighted subgraphs. In *STOC*, pages 455–464, 2009.
[34] X. Yan, P. S. Yu, and J. Han. Graph indexing: a frequent structure-based approach. In *SIGMOD*, 2004.
[35] S. Zhang, M. Hu, and J. Yang. Treepi: A novel graph indexing method. *ICDE*, 2007.
[36] L. Zou, L. Chen, and Y. Lu. Top-k subgraph matching query in a large graph. In *PIKM*, pages 139–146, 2007.