Front. Comput. Sci., 2018, 12(5): 966–983
https://doi.org/10.1007/s11704-016-5522-6

RESEARCH ARTICLE

Using partial evaluation in holistic subgraph search

Peng PENG[1], **Lei ZOU** (✉)[2], **Zhenqin DU**[2], **Dongyan ZHAO**[2]

1 Big Data Provincial Key Laboratory, Hunan University, Changsha 410082, China
2 Institute of Computer Science and Technology, Peking University, Beijing 100871, China

Abstract Because of its wide application, the subgraph matching problem has been studied extensively during the past decade. However, most existing solutions assume that a data graph is a vertex/edge-labeled graph (i.e., each vertex/edge has a simple label). These solutions build structural indices by considering the vertex labels. However, some real graphs contain rich-content vertices such as user profiles in social networks and HTML pages on the World Wide Web. In this study, we consider the problem of subgraph matching using a more general scenario. We build a structural index that does not depend on any vertex content. Based on the index, we design a *holistic subgraph matching* algorithm that considers the query graph as a whole and finds one match at a time. In order to further improve efficiency, we propose a "partial evaluation and assembly" framework to find subgraph matches over large graphs. Last but not least, our index has light maintenance overhead. Therefore, our method can work well on dynamic graphs. Extensive experiments on real graphs show that our method outperforms the state-of-the-art algorithms.

Keywords subgraph search, holistic approach, partial evaluation and assembly

1 Introduction

Because of their flexibility, "graphs" have been adopted by an increasing number of applications as an underlying model. These applications include biological networks [1], social networks [2], and resource description framework (RDF) data [3]. Therefore, graph databases have recently gained considerable attention in the database community. Unlike relational databases, graph databases focus on graph structure-based operations such as shortest-path queries [4–7], reachability queries [8–10], and subgraph queries [11–13]. These queries are not optimized by existing relational database techniques.

In this study, we study the problem of subgraph matching that finds all subgraphs in a data graph G that are isomorphic to a query graph Q. Subgraph queries are often issued in various domains. For example, answering a SPARQL query is equivalent to finding all subgraph matches of query graph Q over RDF graph G. Although subgraph queries have been well studied during the past five years, most existing studies assume that the underlying graphs are vertex/edge-labeled graphs, that is, that each vertex/edge has a label (e.g., A, B, C, and so on), such as in [11,12]. However, many real graphs contain rich-content vertices/edges rather than simple labeled vertices/edges. For example, the content can be a numeric value, multidimensional tuple, text file, or other. In addition, the criteria in query vertices include not only exact vertex label constraints but also many complex expressions. These include value range constraints and regular expressions. Let us demonstrate the diversity of vertex/edge content and the usefulness of subgraph queries by the following two examples.

Example 1 Freebase is a large collaborative knowledge base of structured data harvested from many sources. It is a large entity graph of people, places, and things. Each entity has a multidimensional tuple to describe its properties. The relationship between entities is an edge in the graph.

Received December 5, 2015; accepted November 3, 2016

E-mail: hnu16pp@hnu.edu.cn; {zoulei,zhaodongyan@pku}@pku.edu.cn

Assume that we want to find "all 40–50-year-old married American couples who acted in the same biographical film." This query can be represented as a query graph Q in Fig. 1. Obviously, this graph cannot be simplified as a vertex/edge-labeled graph. Each vertex is associated with a multidimensional tuple. The query uses complex semantics (such as range constraints) rather than comparing exact vertex labels.

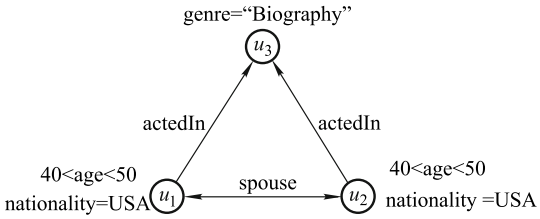


Fig. 1 Query graph Q over freebase

Example 2 Many online professional social networks, such as LinkedIn (linkedIn.com), are also graphs, in which each vertex represents an individual and an edge refers to the collaborative experience between two individuals. Each individual has a profile page to record his or her job titles, experience, skills, and so on. Obviously, some content (such as experience and skills) is represented by text. A recent survey (see MRINETWORK official website) reports that approximately 90% of recruiters use LinkedIn to identify job candidates.

Assume that an IT distributor company wants to build a new division. As we know, effective collaboration and low communication cost are critical for a team. Therefore, qualified candidates who have good cooperative experience are preferred. For example, a department backbone group must contain a division header, product manager, two core sales staff, and a senior product engineer. Furthermore, the division header should have at least ten years expertise in IT product distribution. The two core sales staff should have at least five years of marketing experience. In addition, the division header must have good cooperative experience with the product manager, and the division header must have good relations with the two core sales staff, as should the project manager. That the project manager and senior product engineer have previously worked together is desirable. All of these recruitment requirements can be represented as a query graph Q in Fig. 2.

The subgraph queries in Examples 1 and 2 cannot be modeled as classical subgraph matching queries under the semantics of exact label matching. Therefore, a subgraph query over a general graph model (formally defined in Section 2) is desirable. This is the problem we examine in this study.

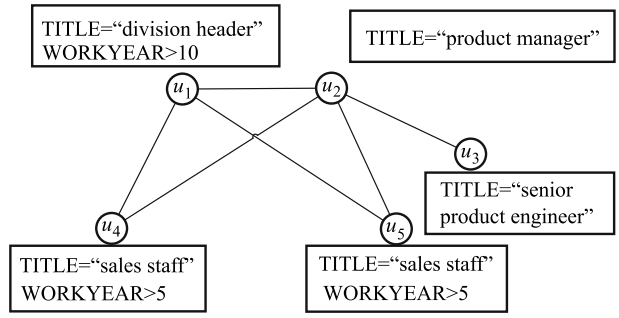


Fig. 2 Query graph Q in IT company recruitment

1.1 Limitations of existing approaches

The new model proposes new challenges. Unfortunately, although previous studies have expended considerable effort on the subgraph query [11–22], few can be applied to this new model. The existing techniques prove inadequate because of the following deficiencies.

(1) Naive model Most existing studies on subgraph search focus on the vertex/edge-labeled graph, in which each vertex/edge has a single label such as "A,B,C." More specifically, they focus on the subgraph query under the semantics of exact label matching. Moreover, these methods build up structural indices based on the vertex labels. For example, NOVA [23] computes the label distribution of the neighborhood around each vertex. ASP [19] classifies all edges by the labels of their endpoints. These indices cannot be used in general graphs (e.g., Examples 1 and 2), because we cannot derive vertex/edge labels in these graphs.

(2) Perform issue Although some studies have examined the semantics of the subgraph query over a general graph, they do not address query optimization issues as do GraphQL [13] and G-SPARQL [20]. Trinity proposes a traversal-based subgraph search algorithm [21]. However, Trinity is a memory system (i.e., the whole graph resides in a memory cloud formed by several machines). However, some real graphs cannot be cached in memory. For example, each node in a social network contains images, text files, and other large content. Neo4j provides the Cypher query language, which can support subgraph queries. The query implementation is based on graph traversal, but it does not work well on a large graph G with more than ten million vertices.

(3) High index complexity In addition, many existing index structures have considerable space costs. GADDI [1] mines some discriminating substructures. Based on these discriminating substructures, GADDI defines a distance measurement and creates an index for every two nodes within

distance k. Thus, GADDI should enumerate all vertex neighborhood subgraphs with length k. The space complexity of GADDI is $O(|V| \times d^k)$, where $|V|$ is the number of vertices and d is the average degree. Analogously, the index space in SPath [18] is $O(|V| \times d^k)$, which is unacceptable for a graph having more than ten million vertices.

(4) Expensive maintenance overhead Furthermore, few studies have addressed index maintenance issues in dynamic graphs. As we know, social networks and knowledge graphs change constantly. Rebuilding indices from scratch in real applications in order to represent frequently updated data is impossible.

1.2 Our contributions

In this research, we concentrate on subgraph query processing in a large general graph in a dynamic environment. In this model, each vertex/edge can be associated with any content and inserted or deleted frequently. Our approach can guarantee that indices are independent of the content of the vertices, whereas most existing subgraph matching approaches build indices based on vertex labels. For example, SPath [18] constructs indices by summarizing the vertex labels within the k-neighborhood subgraph of each vertex. NOVA [23] uses a vector to store the label distribution of the neighbors of a vertex. Clearly, our model is more flexible in real-life applications. The dependent relationship between indices and labels limits the reusability of the existing approaches, as many graphs in many real applications contain rich-content vertices. Thus, our approach is more reusable.

In particular, we integrate existing systems with graph databases as the underlying storage. The vertex/edge-specific content is stored in some existing systems, such as RDBMS for relational tuples and inverted indices for text files. In addition, the graph structure is stored in a native graph system.

Considering the limitations of existing indices, we propose an efficient offline index structure, which has linear space complexity and light maintenance overhead. In general, we design a distance-based vertex coding strategy. If two vertices are close to each other, the difference between their codes is small.

In addition to the index, we also design a holistic online subgraph matching algorithm. In our method, we do not utilize structural join to find matches step by step, as this may generate a considerable number of intermediate results. Instead, we consider the query graph as a whole and find one match at a time. In particular, we maintain a priority queue (according to vertex codes) for each query vertex. When a vertex v must be dequeued, we find subgraph matches containing v directly. In order to speed up this step, we propose the "distance preservation principle" to reduce the search space next to the index. Because our holistic subgraph matching algorithm avoids structural joins, the algorithm can find matches without generating a considerable number of intermediate results.

Furthermore, in order to handle subgraph matches in large graphs, we propose a "partial evaluation and assembly" framework. Specifically, we divide a graph G into several blocks. We use the holistic subgraph matching algorithm to find all *partial matches* (Definition 7), and then assemble all partial matches to find all complete matches.

Last but not least, we evaluate our methods on graphs that contain more than 100 million edges. To the best of our knowledge, these are the largest data graphs described in studies on the subgraph query problem and that use a *single* machine.

Our study makes the following contributions:

- General models Each vertex/edge in our graphs can have any kind of content, including a multidimensional tuple, text, etc. This is because the indices in our methods are independent of the labels in the graphs; they rely only on the link structure of graphs. Thus, our methods are more reusable than are most existing methods.
- Structure-based index We propose a structure-based index with a linear space complexity $O(|V(G)|)$. This index can reduce the search space considerably.
- Efficient solution We first propose a holistic approach to match the query graph. The holistic approach considers the query graph as a whole and finds one match at a time without producing a considerable number of intermediate results. We extend this holistic approach by using the "partial evaluation and assembly" framework to handle large graphs.
- Light maintenance overhead The index maintenance complexity is $O(|E(G)|+|V(G)|\log|V(G)|))$ in the worst case.

The remainder of the paper is organized as follows. We formally define preliminary concepts in Section 2. Section 3 provides an overview of and explains the concepts behind the proposed approach. We present a single pivot-based vertex encoding technique and an associated holistic subgraph matching algorithm in Section 4. In Sections 5, we introduce our advanced holistic subgraph matching algorithm with mul-

tiple pivots. We discuss the maintenance issues in Section 6 and present experimental results in Section 7. Finally, we survey related studies in Section 8 and conclude our study in Section 9.

2 Problem definition

In this section, we review the terminology used throughout this study. A data graph indicates the relationships between entities, where each entity is represented as a vertex and the relationship between two entities is denoted as an edge. The query is also a graph, in which vertices and edges use query criteria related to vertex/edge content. In this study, we do not discuss the method used to store vertex/edge content or find vertices/edges that satisfy the criteria. We can leverage existing content management systems for this purpose, such as relational database management systems (RDBMS) for relational tuples, inverted indices for text files, and multimedia databases for images. Our method can even support criteria containing regular expressions over query vertices, as long as existing content management systems support such regular expressions. The focus of this work is to find a manner in which the structure of a graph can be stored and indexed and the manner in which the subgraph queries can be answered efficiently. Table 1 shows some frequently used symbols.

Table 1 The definitions of symbols

Symbol	Definition
G, Q	Data graph and query graph, respectively
v, u	Vertices in data and query graphs, respectively
$\overline{vv'}$	Single edge with v and v' being the end points
$dist(v, v')$	Shortest path distance between v and v'
$Dia(Q)$	Diameter of graph Q
$L(v)$	Vertex code of v with a single pivot
$(B(v), L(v))$	Vertex code of v with multiple pivots
$TL(u)$	List of all candidate vertices that match the criteria associated with vertex u
$C(u)$	Cursor that points to the next accessed vertex in $TL(u)$ when our holistic subgraph matching algorithm is executed
$TQ(u)$	Queue that buffers some accessed vertices when our holistic subgraph matching algorithm is executed
T	Vertex vector with length $\|V(Q)\|$ that forms a (partial) match of Q over graph G

In order to handle the vertices and edges in a uniform manner, for each edge $e = \overline{v_i v_j}$ in graph G, we introduce an additional vertex (between v_i and v_j) to replace e. The new vertex has the same content that is associated with edge e. We can also change the query graph Q by introducing additional vertices for edges. Therefore, in the following discussion, we assume that no content is present in edges of G. In addition, edges in the query graph Q have no query criterion. Although we only focus on the undirected graphs in this study, our solution can be easily extended to directed graphs.

Definition 1 (Data graph) A data graph is denoted as $G = \{V(G), E(G), \sum, \Gamma\}$, where (1) $V(G)$ is a set of vertices; (2) $E(G) \subseteq V(G) \times V(G)$ is a set of undirected edges; (3) $\sum$ is a set of content; and (4) $\Gamma : V(G) \rightarrow \sum$ denotes the content assignment function, where $\forall v \in V(G)$, $\Gamma(v)$ is v's corresponding content.

Definition 2 (Query graph) A query graph is denoted as $Q = \{V(Q), E(Q), F\}$, where (1) $V(Q)$ is a set of vertices; (2) $E(Q) \subseteq V(Q) \times V(Q)$ is a set of undirected edges; (3) $F = \{f_i\}$ is a set of query criteria associated with vertex $u_i \in V(Q)$, $i = 1, 2, \ldots, |V(Q)|$.

A data graph G and two query graphs are given in Figs. 3 and 4, respectively, which are used in this study as running examples. Because our method does not rely on vertex content or specific query criteria, we use the same vertex shapes as those data vertices(i.e., considering the query criteria in the vertices of Q) of the query vertices shown in Fig. 4.

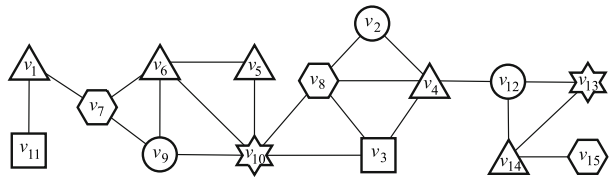


Fig. 3 Data graph (different shapes corresponding to different content in vertices)

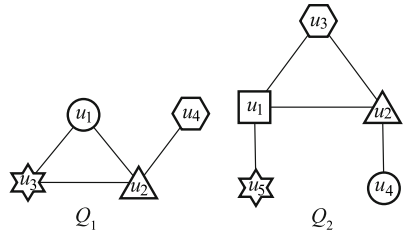


Fig. 4 Example query graph (matching vertices with the same shape in the data graph)

Definition 3 (Subgraph match) Given a data graph $G = \{V(G), E(G), \sum, \Gamma\}$ and a query graph $Q = \{V(Q), E(Q), F\}$, a subgraph M with m vertices $\{u_1, u_2, \ldots, u_n\}$ (in G) is said to be a *match* of Q if and only if a *bijection function* μ from $\{v_1, v_2, \ldots, v_n\}$ to $\{u_1, u_2, \ldots, u_n\}$ exists, where the following conditions hold:

1) $\Gamma(\mu(u_i))$ satisfies the criterion f_i in query vertex u_i, $i = 1, 2, \ldots, n$.

2) $\forall u_i, u_j \in V(Q), \overline{u_i u_j} \in E(Q) \Rightarrow \overline{\mu(u_i)\mu(u_j)} \in E(G)$, $1 \leqslant i, j \leqslant n$.

A *state* $[v_1, v_2, \ldots, v_n]$ is a serialization of a *subgraph match* M.

Problem statement Given a data graph G and a query graph Q, where $|V(Q)| \ll |V(G)|$, the subgraph query problem is to find all *subgraph matches* (Definition 3) of Q in G.

In this study, we assume that data graph G and query graph Q are connected; otherwise, all connected components are considered separately.

3 Overview

A commonly used technique to find subgraph matches is to adopt a structural join such as edge [24], path [18], edge-pair [19], star-pattern [21], or twig [25]. However, the structural join often generates a considerable number of intermediate results and only a few final results. In order to address this issue, we propose a *holistic subgraph matching* (HSM) approach to match query Q as a whole. Unlike in structural joins, the buffer size (for storing the intermediate results) in HSM is linear with the number of vertices in G.

Although holistic pattern matching methods have been studied for XQuery processing in XML databases [26, 27], extending these methods to graph databases is not straightforward. This is because queries and data in XML databases are both trees. In addition, all holistic methods use a tree code to determine ancestor-descendant relationships to enable XQuery processing. However, this study considers a general graph model. No ancestor-descendant relationships exist in a general graph. Thus, the tree code or existing holistic pattern matching algorithms in XML databases cannot be used to solve the subgraph query problem.

We design a distance-based vertex code and propose a structural pruning technique that is based on the *distance preservation principle*. Specifically, given a query graph Q with n vertices, if n vertices $(v_1, v_2, \ldots, v_n)$ in G can form a subgraph match of Q, their pairwise distances must be less than $Dia(Q)$, where $Dia(Q)$ is the diameter of query Q. This observation can help reduce the search space considerably.

Let us first illustrate the intuition by query Q_1 of which the diameter is 2. As shown in Fig. 5 and in u_3 matches to v_{13}, if a subgraph match exists that contains v_{13}, the distance between v_{13} and other matching vertices should be no greater than $Dia(Q_1)$. Therefore, we need to consider only five vertices to find subgraph matches containing v_{13}. Note that the pruning rule depends only on the link structure of graphs. It does not rely on vertex content. Therefore, our method can work with graphs having any kind of content.

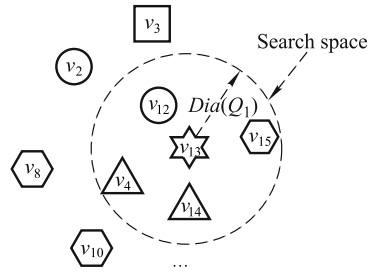


Fig. 5 Example of distance preservation pruning

The previous example motivates us to identify a distance-based pruning technique. However, materializing all pairwise shortest path distances in a large graph G is impractical. Therefore, we propose a pivot-based vertex coding technique. For the ease of presentation, we first discuss a single pivot-based solution in Section 4. This scheme works well with small graphs, but it is not efficient with large graphs. To handle large graphs, we propose a "partial evaluation and assembly" framework, as described in Section 5. Specifically, we divide a large graph into several blocks. In each block, we adopt a similar strategy as described in Section 4 to find partial results in each block *partial match* (Definition 7). Finally, we assemble the partial results to find subgraph matches of Q. Furthermore, in this work, we consider the subgraph query problem in a dynamic environment. In Section 5, we propose an online index maintenance algorithm without rebuilding indices from scratch.

4 Single pivot-based solution

4.1 Single-pivot encoding strategy

Given a pivot vertex v^* in a data graph G, the vertex codes are defined in Definition 4.

Definition 4 (Vertex code) Given a graph G and a pivot $v^* \in V(G)$, $\forall v \in V(G)$, the vertex code of v is $L(v) = dist(v, v^*)$, where $L(v)$ denotes the vertex code and $dist(v, v^*)$ is the shortest path distance between the two vertices.[1)]

We use a two-column table *TB* to store all vertex codes, where the vertex ID and code are listed in the first and second columns, respectively. A clustered B^+-tree is built over the vertex code column. In addition, we maintain the short-

[1)] As previously mentioned, we assume that the data graph G is connected; otherwise, all connected components are considered separately

est path tree of v^* by storing the parent of each vertex in the shortest path tree of v^*.

Given a query Q with n vertices $u_1, u_2, \ldots, u_n$, based on the query criteria in u_i, we can find a candidate list $TL(u_i)$, where all vertices in $TL(u_i)$ satisfy the query criteria in u_i, $i = 1, 2, \ldots, n$. The following theorem can filter out the vertices that violate the distance preservation principle.

Theorem 1 Given a query graph Q and a data graph G, for any two query vertices u_i and u_j in Q, the corresponding candidate lists are $TL(u_i)$ and $TL(u_j)$, respectively. For a candidate v in $TL(u_i)$, if no vertex v' is in $TL(u_j)$, where $|L(v)-L(v')| \leqslant dist(u_i, u_j)$, v can be pruned from $TL(u_i)$ safely. $L(v)$ and $L(v')$ denote the vertex codes and $dist(u_i, u_j)$ denotes the shortest path distance between u_i and u_j in query Q.

Proof According to the triangle inequality, $|L(v) - L(v')| \leqslant dist(v, v')$. If no vertex v' exists in $TL(u_j)$, where $|L(v) - L(v')| \leqslant dist(u_i, u_j)$, then for each vertex v' in $TL(u_j)$, $|L(v) - L(v')| > dist(u_i, u_j)$. Therefore, for any vertex v' in $TL(u_j)$, $dist(u_i, u_j) \leqslant dist(v, v')$. Assume that a subgraph match M (of query Q) exists that contains v and v'. v and v' are matched to u_i and u_j, respectively. Knowing $dist(v, v') \leqslant dist(v, v')$ is straightforward. This contradicts the previous analysis. Therefore, no subgraph match exists that contains v and v'. This also means that v can be pruned safely. □

If a vertex v in $TL(u_i)$ cannot be pruned by Theorem 1, the following theorem tells us which vertices should be considered when we find subgraph matches containing v.

Theorem 2 Given a data graph G and a query graph Q, for a vertex v in $TL(u_i)$, when we find a match M containing v, for each $TL(u_j)$, the search space is $\{v' | v' \in TL(u_j) \wedge |L(v) - L(v')| \leqslant dist(u_i, u_j)\}$, where u_i and u_j are two vertices in query Q ($i \neq j$).

Proof If a match M exists that contains both v in $TL(u_i)$ and v' in $TL(u_j)$, $dist(v, v') \leqslant dist(u_i, u_j)$. In addition, because of the triangle inequality, $|L(v) - L(v')| = |dist(v, v^*) - dist(v', v^*)| \leqslant dist(v, v')$. Thus, $|L(v) - L(v')| \leqslant dist(u_i, u_j)$. If $|L(v) - L(v')| > dist(u_i, u_j)$, v and v' cannot form a match. □

Different pivots provide different pruning power. We show the different pruning powers when selecting different pivots in data graph G (in Fig. 3). If we select v_{10} as the pivot, we can examine Fig. 6(a) to identify the shortest path trees rooted at v_{10}. Considering edge $\overline{u_1u_2}$ in query Q_1, $TL(u_1) = \{v_2, v_9, v_{12}\}$ and $TL(u_2) = \{v_1, v_4, v_5, v_6, v_{14}\}$. If we want to find a match containing v_9 (matching u_1), $|L(v_9) - L(v4)| \leqslant dist(u_1, u_2) = 1$, $|L(v_9) - L(v5)| \leqslant dist(u_1, u_2) = 1$ and $|L(v_9) - L(v_6)| \leqslant dist(u_1, u_2) = 1$. According to Theorem 2, we know the search space in $TL(u_2) = \{v_4, v_5, v_6\}$. If we select v_{11} as the pivot, we can find that the search space in $TL(u_2)$ is $\{v_5, v_6\}$. The shortest path tree rooted at v_{11} is given in Fig. 6(b). Obviously, selecting v_{11} as the pivot provides stronger pruning power.

Intuitively, given two vertices, the greater the difference in their distances to a pivot, the more impossible it is that they can be contained by a match based on Theorem 2. Thus, we should select the pivot to which the number of vertices having different distances to it should be as high as possible. Thus, if a shortest path tree rooted at a pivot has more layers and each layer has fewer nodes, stronger pruning power can be provided. For example, as shown in Fig. 6(b), because the shortest path tree rooted at v_{11} has more layers and each layer has fewer nodes, selecting v_{11} as the pivot provides greater pruning power than does v_{10}. Based on this observation, we define the vertex entropy and select the vertex having the maximum vertex entropy.

Definition 5 (Vertex entropy) Given a vertex v, according to the vertex codes, a list $\{(code_1, fre_1), (code_2, fre_2), \ldots, (code_n, fre_n)\}$ exists, where $code_i$ is a vertex code and fre_i denotes the number of vertices whose vertex codes are $code_i$,

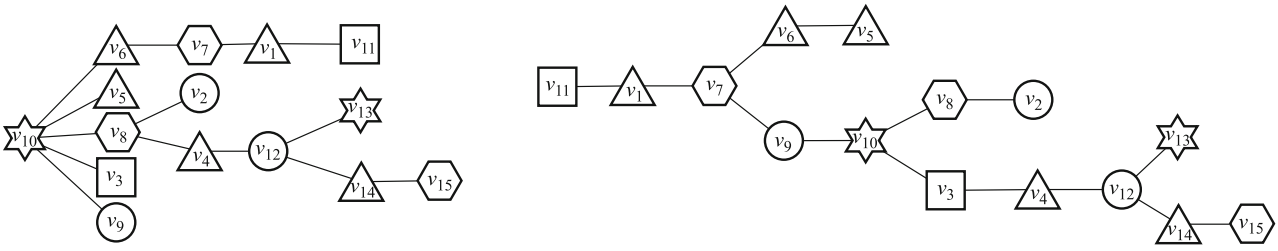


Fig. 6 Evaluation of vertices (a) v_{10} and (b) v_{11} as pivots

$i = 1, 2, \ldots, n$. The *vertex entropy* is defined as follows:

$$E(v) = -\sum_{i=1}^{i=n} \frac{fre_i}{|V(G)|} \log(\frac{fre_i}{|V(G)|}). \quad (1)$$

In order to compute the vertex entropy, we propose the following method. We first compute all pairwise shortest path distances. For each vertex $v \in V(G)$, we compute the vertex entropy according to Eq. (1). The vertex v with the maximum vertex entropy is selected as the pivot v^*. The complexity of this process is $O(|V(G)|^3)$. In order to speed up this step, we randomly select only some sample vertices and compute the shortest path trees of these vertices to estimate their entropy. Thus, the time expended finding pivots is acceptable.

When the graph G is large, randomly selecting vertices to estimate the entropy still incurs high computational cost. We discuss the partition-based solution in Section 5. We partition G into several blocks and select pivots from each block.

4.2 Holistic subgraph matching algorithm

Based on vertex codes, we propose our HSM algorithm. We discuss the data structures used in the HSM algorithm in Section 4.2.1, then describe the algorithm itself in Section 4.2.2.

4.2.1 Data structures

Each vertex u_i (in query Q) is associated with three data structures in HSM: *candidate list* $TL(u_i)$, *cursor* $C(u_i)$, and *queue* $TQ(u_i)$. Figure 7 shows the corresponding data structures of Q_1.

Candidate list $TL(u_i)$ is a list of all candidate vertices that satisfy the criteria associated with vertex u_i. Note that all candidate vertices are sorted in non-descending order by vertex code (see Definition 4) in $TL(u_i)$.

Cursor $C(u_i)$ points to the vertex (in the candidate list $TL(u_i)$) that is currently accessed. For simplicity of notation, we also use "$C(u_i)$" to denote the vertex that $C(u_i)$ points to when the context is clear. Initially, each cursor $C(u_i)$ ($i = 1, 2, \ldots, |V(Q)|$) points to the first vertex in $TL(u_i)$. In each step, the cursor that points to the *minimal* vertex among all vertices $C(u_i)$ moves one step forward to the next vertex in the list. Let us examine Fig. 7. Because v_1 is minimal among $\{v_9, v_1, v_{10}, v_7\}$ (according to vertex codes), cursor $C(u_2)$ moves one step forward to the next vertex. Meanwhile, v_1 is moved into queue $TQ(u_2)$.

Queue $TQ(u_i)$ stores the vertices that have been accessed. For an accessed vertex v in $TQ(u_i)$, the search space for finding subgraph matches containing v is bounded by Theorem 2. If all vertices in the search space have been accessed, we dequeue v from $TQ(u)$. The dequeuing rule is described in detail in Section 4.2.2. When a vertex v is dequeued from $TQ(u)$, we execute a graph exploration algorithm to find all subgraph matches containing v. During this exploration, we use *states* to record each step.

State T is a vertex vector with length $|V(Q)|$. Each dimension of the vector corresponds to a query vertex in Q. A state is a serialization of a (partial) match of query graph Q.

4.2.2 HSM algorithm

The primary purpose of our HSM algorithm is to repeatedly construct subgraph matches in which some vertices are close to each other. During query execution, all queues store some vertices close to each other. If a vertex in a queue is too far from another vertex in another queue (as discussed in Theorem 2), it dequeues. For example, let us consider a moment during query execution as shown in Fig. 8. At this moment, the vertex codes of all candidates in any queue are smaller than 5 and larger than 1. This means that all subgraph matches whose distance to v^* are less than 5 and more than 1 can be found by vertices in all queues. When cursor $C(u_4)$ moves to the next element v_8 and v_8 enqueues, v_7 becomes too far from other vertices. Hence, v_7 dequeues from $TQ(u_4)$.

When a vertex v dequeues, we call the graph exploration function, ExploreGraph(v), to search all queues and find subgraph matches containing v. We utilize *states* to record each step in graph exploration. Each state is a (partial) match of

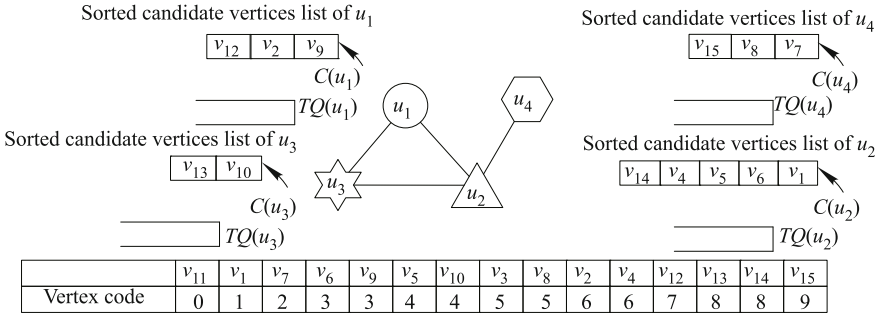


Fig. 7 Candidate lists, cursors, and queues during execution

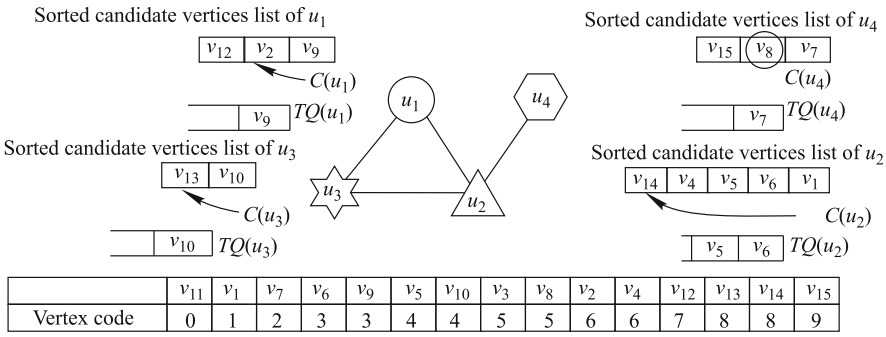


Fig. 8 Example moment during query execution

query graph Q. We can then formulate our problem as one in which we search the space of states that represent all possible partial matches. We propose a graph exploration method over the state space to find matches. Algorithm 1 (Function ExploreGraph(v)) in Algorithm 2 shows the details. When v_7 dequeues from $TQ(u_4)$, we can examine Fig. 9 to see how our graph exploration method finds matches containing v_7.

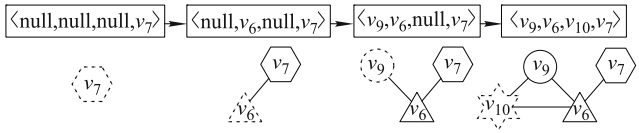


Fig. 9 Finding matches through graph exploration

Algorithm 1 ExploreGraph(v)

```
1: Initialize a state T with T[u] = v.
2: Push T into a stack S.
3: while S ≠ ∅ do
4:    Pop the first state T ∈ S.
5:    if all edges in E(Q) have been matched by T then
6:       Insert T to RS.
7:    for each unmatched edge u'u'' where u' has been matched by T[u'] do
8:       if T[u''] ≠ null then
9:          if (T[u'], T[u'']) ∈ E(G) then
10:            Push T into S
11:         else
12:            for each neighbor v' of T[u'] do
13:               if v' ∈ TQ(u'') then
14:                  Initialize a state T' with T.
15:                  T'[u'] ← v'.
16:                  Push T' into S.
```

Algorithm 2 shows the pseudo-codes of our holistic subgraph matching algorithm. We first obtain all sorted vertex lists $TL(u_i)$, $i = 1, \ldots, |V(Q)|$ according to the query criteria in query vertices u_i, $i = 1, 2, \ldots, |V(Q)|$. All cursors $C(u_i)$ point to the head of $TL(u_i)$ and all queues $TQ(u_i)$ are empty (lines 1–4). Let $v^* = C(u^*) = \text{Min}\{C(u_i)\}$, $i = 1, 2, \ldots, n$, where $C(u_i)$ refers to the current vertex that cursor $C(u_i)$ points to (line 6). We put cursor $C(u^*)$ one step forward and move v^* into the corresponding queue $TQ(u^*)$ (lines 7 and 8). At this moment, we also check if some vertices can be dequeued (Line 9). Consider a vertex v_i in queue $TQ(u_i)$, $i = 1, 2, \ldots, n$. Let $Dia(u_i) = \text{MAX}\{dist(u_i, u) | u \in V(Q)\}$. If $(L(v^*) - L(v_i)) > Dia(u_i)$, then the vertices whose vertex codes are larger than $L(v^*)$ cannot be in a subgraph match containing v_i (which can be proved by Theorem 1). In order to find subgraph matches containing v_i, considering the vertices that are visited after v^* is not necessary. In other words, we need to consider only the vertices in the current queues to find subgraph matches containing v_i. At this moment, vertex v_i is dequeued from $TQ(u_i)$. When v_i is dequeued from $TQ(u_i)$, we first determine whether v_i can be pruned according to Theorem 1 (lines 11 and 12). If not, we call a *graph exploration* function to find subgraphs (of Q) containing v_i (line 13). Essentially, the graph exploration function is the same as the VF2 subgraph isomorphism algorithm. The only difference is that our algorithm begins the search process from vertex v_i.

Algorithm 2 Holistic subgraph matching algorithm

```
   Input: Query graph Q and data graph G.
   Output: All subgraph matches of Q.
1: for each vertex u_i in query Q do
2:    Find the candidate list TL(u_i) that contains all vertices that satisfy
      the query criteria f_i in order of vertex codes.
3:    Cursor C(u_i) points to the head vertex of TL(u_i).
4:    TQ(u_i) = φ
5: while at least one cursor C(u_i) does not point to the tail of TL(u_i) do
6:    Let v* = Min{C(u_i)}, i = 1, 2, ..., n and v*'s corresponding cursor
      is C(u*).
7:    Forward C(u*) to the next vertex in TL(u*) if any.
8:    Find the set of all vertices that need to be dequeued and denote as
      VS according to Theorem 2.
9:    for each vertex u_i in query Q do
10:      if v can be pruned by Theorem 1. then
11:         Continue.
12:      Call ExploreGraph(v) (Algorithm 1) to find the set of all matches
         of Q, denoted as RS.
13:   Dequeue all vertices in VS.
14:   Put v* into TQ(u*).
15: Return RS.
```

5 Partial evaluation and assembly-based solution

Given a pivot v^* in a large graph G, many vertices exist that have the same vertex codes, because their distances to v^* are equal. Obviously, this will affect the pruning power described in Theorem 1. To address this issue, we propose a "partial evaluation and assembly" framework, described in this section.

We partition the whole graph G into several blocks. Here, because graph partitioning is a well-studied problem in computer science, we can leverage previously existing code to perform the partitioning for us. Note that the partitioning algorithm is orthogonal to our approach. Any vertex-disjoint partitioning method can be adopted by our approach, such as METIS [28] or MLP [29]. In our study, we adopt the METIS algorithm [28], which is the most famous graph partitioning method, to find a vertex-disjoint partition. In a future study, we will switch to a better partitioner.

After partitioning is conducted, edges whose endpoints are in two different blocks are called *crossing edges*. The endpoints of these crossing edges are called *boundary vertices*. For each block B, we introduce one-hop neighborhoods of all boundary vertices in B to form an extended block B'. We refer to the vertices initially in block B as *inner vertices*. The new vertices in B' (i.e., $B' - B$) are called *extended vertices*. Given a graph G in Fig. 3, we can partition it into three blocks. The dashed lines in Fig. 10 represent the extended vertices. The bold lines in Fig. 10 represent the pivot in each block.

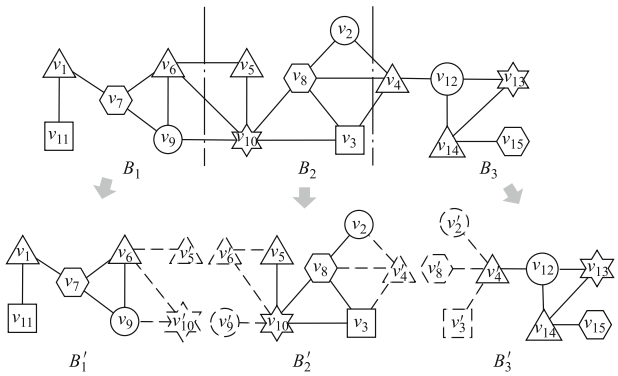


Fig. 10 Graph partition

Definition 6 Given a subgraph match M of query Q over graph G, if all vertices of M are inner vertices of a block, M is called an *inner match*; otherwise, M is called a *crossing-match*.

Figure 11 shows a crossing-match M of Q_1 that crosses two blocks B_1 and B_2.

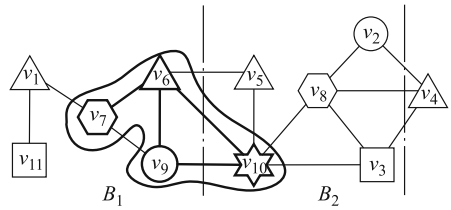


Fig. 11 Crossing-match of Q_1

Employing the method described in Section 4.2 is straightforward for finding inner matches in each block. Therefore, in the following discussion, we focus on finding crossing-matches.

The "partial evaluation and assembly" framework consists of two phases. First, we must find partial results in each block (Section 5.1). Then, we assemble these partial results to find crossing-matches (Section 5.2).

5.1 Partial evaluation

In this subsection, we focus on the partial evaluation phase. In this phase, we consider each block separately. Assume that we want to find partial results in an extended block B'. According to Definition 4, we can compute the vertex codes for each vertex in block B'. Figure 12 shows the vertex codes of all vertices after partitioning. Then, the complexity of the vertex code is $O(|V(G)| + |E_P(G)|)$, where $E_P(G)$ is the set of all crossing edges. In real applications, $|E_P(G)|$ is often smaller than $|V(G)|$. Hence, the complexity of the vertex code is still $O(|V(G)|)$.

	v_{11}	v_1	v_7	v_6	v_9	v'_5	v'_{10}
Vertex code in B'_1	0	1	2	3	3	4	4

	v_5	v'_6	v_{10}	v_3	v_8	v'_9	v_2	v'_4
Vertex code in B'_2	0	1	1	2	2	2	3	3

	v_{15}	v_{14}	v_{12}	v_{13}	v_4	v'_2	v'_3	v'_8
Vertex code in B'_3	0	1	2	2	3	4	4	4

Fig. 12 Vertex codes in each extended block

When we consider crossing-match M and block B', we can examine three cases between edges in M and block B'.

1) The two endpoints are both inner vertices of B'.
2) One endpoint is an inner vertex in B' and the other is an extended vertex in B'. For example, as shown in Fig. 11, both edges $\overline{v_6v_{10}}$ and $\overline{v_9v_{10}}$ have an endpoint as an inner vertex of B'_2 and an endpoint as an extended vertex of B'_2.
3) The two endpoints are both extended vertices in B',

such as edge $\overline{v_6v_9}$ for B'_2 as shown in Fig. 11.

Obviously, the edges in the first two cases should be found in the extended block B'. The edges in the third case of B' must be the first or second case in another block and therefore can be found in other blocks. For example, $\overline{v_6v_9}$ in Fig. 11 is the first case in block B'_1.

Based on the previous analysis, we define a *partial match* as follows. The aforementioned three cases correspond to Conditions 2–4 in Definition 7, respectively.

Definition 7 (Partial match) Given a match M of query Q with the mapping function μ, we say that M is a *partial match* in an extended block B' (the original block is called B) if and only if the following conditions hold:

1) $\mu(u_i) \in B' \vee \mu(u_i) = \text{null}, i \in [1, |V(Q)|]$;
2) $\mu(u_i) \in B \wedge \mu(u_j) \in B \wedge (\exists \overline{u_iu_j} \in Q) \rightarrow \exists \overline{\mu(u_i)\mu(u_j)} \in M$, $i \neq j \in [1, |V(Q)|]$;
3) $\mu(u_i) \in (B' - B) \wedge \mu(u_j) \in B \wedge (\exists \overline{u_iu_j} \in Q) \rightarrow \exists \overline{\mu(u_i)\mu(u_j)} \in M$, $i \neq j \in [1, |V(Q)|]$;
4) $\mu(u_i) \in (B' - B) \wedge \mu(u_j) \in (B' - B) \rightarrow \nexists \overline{\mu(u_i)\mu(u_j)} \in M$, $i \neq j \in [1, |V(Q)|]$;
5) The subgraph induced by all inner vertices in M is connected.

For example, Fig. 13(a) shows a partial match of the extended block B'_2 for query Q_1, denoted as $\langle v_{10}, v_6, v_9, \text{null}\rangle$. Fig. 13(b) shows a partial match of block B_2 for query Q_2, denoted as $\langle v_3, v_4, v_8, \text{null}, v_{10}\rangle$. Note that $\langle v_3, v_4, v_8, v_2, v_{10}\rangle$ is not a partial state in B'_2, because the subgraph induced by inner vertex $\{v_3, v_8, v_2, v_{10}\}$ is not a connected subgraph.

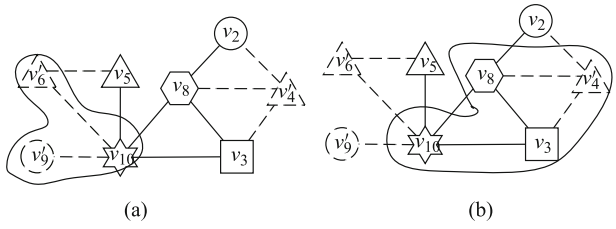


Fig. 13 Example partial matches over block B'_2. (a) Partial match corresponding to state $\langle v_{10}, v_6, v_9, \text{null}\rangle$; (b) partial match corresponding to state $\langle v_3, v_4, v_8, \text{null}\rangle$

In addition, we should change the pruning rule given in Theorem 1. After partitioning is performed, an extended block contains only a partial graph. Therefore, two vertices in a crossing-match may not be in the same extended block. However, because an extended block contains all one-hop neighbors, two adjacent vertices in a match must be in the same extended block. Thus, given a candidate v in $TL(u)$, if no neighbor of v matching u's neighbor exists, v can be pruned. This pruning rule is described in detail as follows.

Theorem 3 Considering two vertices u_i and u_j in query Q, where u_i is adjacent to u_j, their corresponding candidate lists are $TL(u_i)$ and $TL(u_j)$ in block B', respectively, $i \neq j$. For a candidate v in $TL(u_i)$, if no vertex v' in $TL(u_j)$ exists, where $|L(v) - L(v')| \leqslant 1$, v can be pruned from $TL(u_i)$ safely. $L(v)$ and $L(v')$ denote the vertex codes in B' of v and v', respectively.

Proof The proof is similar to Theorem 1. □

Algorithm 3 is our subgraph matching algorithm based on the "partial evaluation and assembly" framework. It is the same as Algorithm 2 except for five minor differences, which are described as follows.

Algorithm 3 Partial-Assembly subgraph matching algorithm

Input: Query graph Q and data graph G with n blocks $\{B'_1, B'_2, \ldots, B'_n\}$.
Output: All subgraph matches of Q.

```
1: for each vertex u_i in query Q do
2:    Find the candidate list TL(u_i) that contains all vertices satisfying the
      query criteria f_i in order of vertex codes.
3:    Cursor C(u_i) points to the head vertex of TL(u_i).
4:    TQ(u_i) = φ
5: while at least one cursor C(u_i) does not point to the tail of TL(u_i) do
6:    Let v* = Min{C(u_i)}, i = 1, 2, ..., n and v*'s corresponding cursor
      is C(u*).
7:    Forward C(u*) to the next vertex in TL(u*), if it exists.
8:    Find the set of all vertices that need to be dequeued and denote as
      VS according to Theorem 2.
9:    for each vertex v in VS do
10:      if v can be pruned by Theorem 3 then
11:         Continue.
12:   Call ExploreGraph(v) to find the set of all partial matches of Q,
      denoted as MS.
13:   for each partial match m in MS do
14:      if the size of m is equal to |E(Q)| then
15:         Put m into RS. // RS is the final result set.
16:      else
17:         Put m into IS. // IS is the intermediate result set.
18:   Dequeue all vertices in VS.
19:   Put v* into TQ(u*).
20: Find subgraph matches of Q by calling Algorithm 4.
21: Return RS
```

1) Theorem 3 can be used as a pruning rule to filter out some candidate vertices (lines 10 and 11 in Algorithm 3);
2) When an inner vertex v of $B(v)$ is dequeued from $TQ(u)$, we perform the graph exploration-based func-

tion (ExploreGraph(v) in Algorithm 3) to find all partial matches of Q in block B. If the size of a partial match is equal to $|E(Q)|$, meaning that it is already a subgraph match of Q, the match is inserted into the result set RS. Otherwise, we insert the partial subgraph matches into the intermediate result set IS (lines 13–18 in Algorithm 3);

3) The terminating condition in ExploreGraph(v) is changed. We terminate the function when we find partial matches containing v in block $B(v)$ (line 5 of function ExploreGraph(v) in Algorithm 3);

4) According to the fourth condition in Definition 7, a partial match cannot contain an edge whose endpoints are both extended vertices. Thus, when we explore the next vertex from the current vertex $T[u']$, we do not need to consider $T[u']$ if $T[u']$ is an extended vertex (lines 8 and 9 of function ExploreGraph(v) in Algorithm 3);

5) According to the topology structures of crossing edges, we call Algorithm 4 to perform self-join recursively over IS (discussed in Section 5.2). We can find all crossing-matches of Q by merging partial intermediate matches in IS (line 20 in Algorithm 3).

For example, let us consider the moment shown in Fig. 14. The cursor with the minimal vertex code in B_2' is $C(u_3) = v_8$ and $|L(v_8) - L(v_5)| = 2 > Dia(u_2) = 1$. Therefore, we need to dequeue v_5 and move v_8 into query $TQ(u_4)$. When we dequeue v_5, no elements are present in $T(Q_4)$. Thus, we can prune v_5 safely.

5.2 Assembly

The final step in our solution is to assemble all partial matches. The following definition tells us which two partial matches can be joined and the join result.

Algorithm 4 Joining partial matches

Input: The intermediate result set IS.
Output: All crossing-matches.

```
1:  IS_0 ← IS.
2:  while |IS| > 0 do
3:    IS' ← ∅.
4:    for each partial match M_1 in IS do
5:      for each partial match M_2 in IS_0 do
6:        if M_1 can join with M_2 and M_2 ⊄ M_1 then
7:          M ← M_1 ⋈ M_2.
8:          if M is a final match of Q then
9:            Put M into the answer set RS //M is a crossing-match.
10:         else
11:           Put M into IS'
12:   IS ← IS'.
```

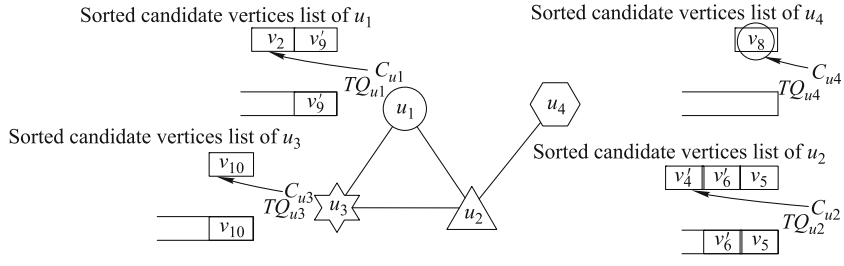
Definition 8 (Join condition and result) Given two partial matches M_1 and M_2 with the mapping functions μ_1 and μ_2, we can join them if and only if the following conditions hold:

$$\forall i \in [1, n], \mu_1(u_i) \neq \text{null} \wedge \mu_2(u_i) \neq \text{null} \rightarrow \mu_1(u_i) = \mu_2(u_i).$$

If two partial matches M_1 and M_2 can be joined, the join result $M = M_1 \bowtie M_2$ is defined as follows:

1) If $\mu_1(u_i) = \mu_2(u_i)$, $\mu(u_i) = \mu_1(u_i)$.
2) If $\mu_1(u_i) = \text{null} \wedge \mu_2(u_i) \neq \text{null}$, $\mu(u_i) = \mu_2(u_i)$.
3) If $\mu_1(u_i) \neq \text{null} \wedge \mu_2(u_i) = \text{null}$, $\mu(u_i) = \mu_1(u_i)$.
4) If $\mu_1(u_i) = \text{null} \wedge \mu_2(u_i) = \text{null}$, $\mu(u_i) = \text{null}$.

Based on the previous definition, we propose Algorithm 4 to join intermediate results to find crossing-matches of Q. In general, we recursively perform self-join over the intermediate result set IS and obtain all subgraph matches.



	v_5	v_6'	v_{10}	v_3	v_8	v_9'	v_2	v_4'
Vertex code in B_2'	0	1	1	2	2	2	3	3

Fig. 14 Example moment during query execution over B_2'

6 Maintenance

In this work, we consider the subgraph query problem over a dynamic graph. In order to support online updates, we address index maintenance issues in this section. Because our vertex codes rely on the shortest path tree rooted at pivots, we discuss only how to update the shortest path tree in a dynamic graph. As we know, the shortest path tree maintenance problem has been well examined in [5, 30]. We can apply existing algorithms to our index maintenance problem. To make our study self-contained, we briefly introduce the shortest path tree maintenance method as given in [5]. Interested readers can refer to [5] for more details. This actually represents an advantage of our solution. Our index is easy to maintain, whereas existing indices for subgraph queries must be rebuilt from scratch to support updates.

Note that, as discussed in Section 5, when the graph is large, we partition it into many blocks and build the shortest path tree-based indices over these blocks. Then, we only need to update the indices over the blocks for the purpose of index maintenance. Hence, index maintenance for a block is dependent only on the size of the block (i.e., independent of the whole graph).

Inserting or deleting an isolated vertex does not affect the shortest path tree. Deleting a vertex means deleting all adjacent edges to the vertex. Therefore, in this section, we only discuss deleting or introducing an edge between two vertices in a block B.

6.1 Deletion

Given a pivot v^* in block B, the shortest path tree rooted at v^* is denoted as $T(v^*)$. Assume that we delete an edge $e = \overline{v_1v_2}$ from B. Three cases can then be considered.

1) $p[v_1] \neq v_2 \wedge p[v_2] \neq v_1$, where $p[v_1]$ is the parent of v_1 in the shortest path tree. This means that e is not in the shortest path tree $T(v^*)$. Thus, deleting e does not affect any vertex code.

2) $p[v_1] = v_2$. We only need to recompute the codes of vertices in the subtree of $T(v_1)$. We employ the Dijkstra algorithm with the remaining vertices, $T - T(v_1)$, to compute the new vertex code of vertices in $T(v_1)$.

In Fig. 15, v_{15} is a pivot in B'_3 and the shortest path tree of $T(v_{15})$ is represented by the bold lines. If we delete edge $e = \overline{v_{14}v_{12}}$, we need to recompute the codes of v_2, v_3, v_4, v_8, and v_{12}. We run the Dijkstra algorithm from vertex v_{14} and the initial distance is $dist(v_{14}, v_{15})$ to update these vertex codes. Obviously, the complexity of deleting an edge is $O(|E(B)| + |V(B)|log|V(B)|)$ in the worst case.

(3) $p[v_2]v_1$. It is analogous to the second case.

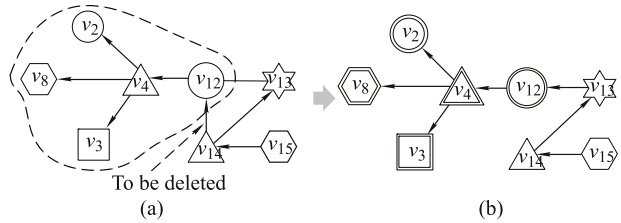


Fig. 15 Deletion. (a) Before deletion; (b) after deletion

6.2 Insertion

Assume that we insert an edge $e = \overline{v_1v_2}$ into B'. The vertex codes of v_1 and v_2 are $L(v_1)$ and $L(v_2)$. If $L(v_1) < L(v_2)$, we employ the Dijkstra algorithm from v_1 to recompute all other vertex codes, and vice versa. For example, if we insert edge $e = \overline{v_4v_{14}}$ as shown in Fig. 16, we run the Dijkstra algorithm from v_4 and update the vertex codes of v_2, v_3, v_4, and v_8. Knowing that the time complexity is $O(|E(B')| + |V(B')|log|V(B')|)$ is clear.

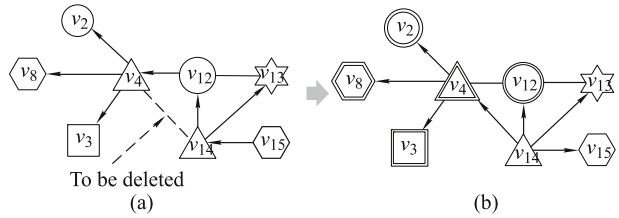


Fig. 16 Insertion. (a) Before deletion; (b) after deletion

7 Experimental evaluation

We evaluated our method using both real and synthetic datasets and compared it with the state-of-the-art algorithms such as GADDI [1], NOVA [23], and ASP [19].

7.1 Datasets and setup

We tested four real-life datasets in our experiments. Statistics related to the graphs are given in Table 2.

Table 2 Real graph datasets

Dataset	$\|V\|$	$\|E\|$	Label number
US Patents	3,774,768	16,518,948	-
Yago2	10,557,345	130,447,832	-
Yago	368,587	543,815	45,450
HPRD	9,460	37,000	307

• **US Patents** US Patents contain details on U.S. patents granted between January 1963 and December 1999. Each

vertex represents a patent and has six attributes: "AppYear", "Subcat", "Class", "Date", "Year", and "Country". Each edge represents the citation between them.

• **Yago2** Yago2 [31] is a huge semantic knowledge base derived from Wikipedia, WordNet, and GeoNames. Vertices correspond to entities in Yago2 and edges correspond to the relationships among them. In addition, each vertex is associated with a string and is derived from the rdfs:label value of each entity.

• **Yago** Yago is the original version of Yago2 without spatial and temporal features. For each vertex, we used its corresponding type information as its vertex label.

• **HPRD** HPRD is a human protein interaction network. For each vertex, we used its GO term description as its label.

The graphs of US Patents and Yago2 do not have vertex labels. Because existing state-of-the-art subgraph search algorithms are based on the semantics of exact label matching, we compared our method to them with respect to vertex-labeled graphs such as Yago and HPRD.

In addition, we used two classic data models, the Erdos Renyi (ER) and scale-free (SF) models, to generate two synthetic datasets. We used these datasets to show the performance of our methods with increasing $|V(G)|$. The ER model is a classical random graph model. It defines a random graph as N vertices connected by M edges, chosen randomly from $N(N-1)/2$ possible edges. By contrast, the degree distribution of the SF model follows a power law distribution.

In addition, our query set was generated using depth first search traversal from a randomly chosen node. The first N nodes were stored as the query pattern. Note that, as discussed in Section 2, our methods did not need to transfer the queries; they only needed to consider the criteria associated with a query vertex to derive lists of all candidate vertices.

We conducted all experiments on a computer with a 2.0-GHz Intel Core 2 Duo processor and 32 GB memory running Linux. In our experiments, we used MySQL (version 5.5.15.0) as the vertex content management system (i.e., using MySQL to find candidate vertices that satisfy the criteria in each query vertex).

7.2 Performance comparison

7.2.1 In graphs with rich-content nodes

We first evaluated our subgraph search methods (Algorithms 2 and 3) on US Patents and Yago2. Because most subgraph match algorithms can work only on vertex-labeled graphs, we compared our solution only to SQLs and the graph exploration method [21]. For SQLs, we stored the graph structure in two types of tables: vertex and edge. A subgraph query can be modeled as an SQL query. In order to speed up query processing, we built the B^+-tree indices over all columns of the tables. For the graph exploration method, we cached the whole graph in memory.

Over the US Patent dataset, Fig. 17(a) shows our method was faster than SQLs by an order of magnitude and twice as fast as the graph exploration. Note that SQLs could not complete query processing within a reasonable time when $|E(Q)| > 8$, nor could the graph exploration method when $|E(Q)| > 12$. Furthermore, we found that Partial-Assembly (i.e., Algorithm 3) was better than HSM (Algorithm 2), particularly when $|E(Q)|$ was large. In Partial-Assembly, we set the pivot number to 1,000. Our evaluation of the effect of the pivot number is provided in Section 1. We obtained similar results over the Yago2 dataset, as shown in Fig. 17(b). Note that although Yago2 was much larger than the US Patent dataset, because the query criteria of each query vertex in Yago2 were more selective, the performance of our method over Yago2 was better than that over US Patent. In addition, the SQL method over Yago2 was too slow to finish the query when $|E(Q)| \geqslant 4$. This was because SQL required numerous expensive join steps and generated a considerable number of intermediate results.

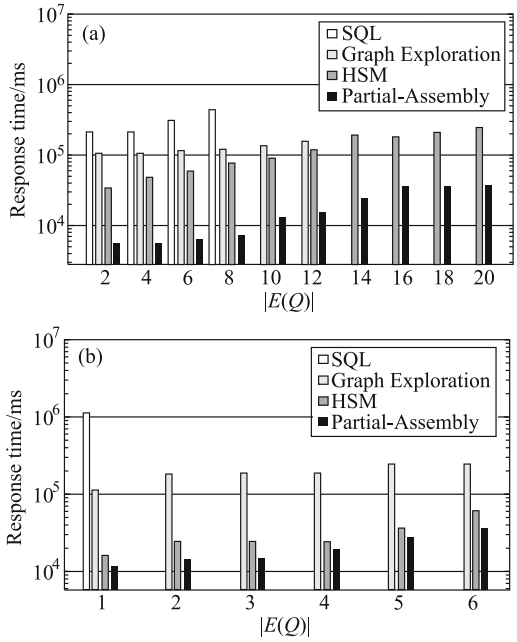


Fig. 17 Online performance comparison over graphs with rich-content nodes. (a) Online performance over US Patent; (b) online performance over Yago2

7.2.2 In vertex-labeled graphs

For the comparison with existing solutions, we degraded our

solution into a vertex-labeled graph, as existing solutions only work on vertex-labeled graphs. We compared our methods (HSM and Partial-Assembly) with the following state-of-the-art methods: GADDI [1], NOVA [23], ASP [19], and $\text{Turbo}_{\text{ISO}}$ [32]. When a data graph was small, such as the HPRD graph, our methods were not as effective as ASP and $\text{Turbo}_{\text{ISO}}$. Partial-Assembly was even worse than NOVA. However, GADDI and Nova could not complete the index construction for a large graph using the Yago dataset. Therefore, we compared our methods only with ASP, as shown in Fig. 18(b). We confirmed that both HSM and Partial-Assembly are faster than ASP and $\text{Turbo}_{\text{ISO}}$. We also found that Partial Assembly is better than HSM in large graphs.

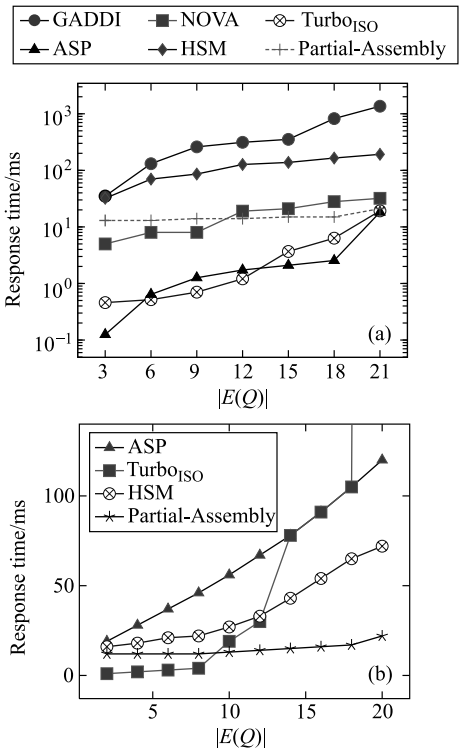


Fig. 18 Online performance comparison with existing vertex-labeled graphs methods over real datasets. (a) Online performance over HPRD; (b) online performance over Yago

Figure 19 shows the performance of our methods (HSM and Partial-Assembly) over two synthetic datasets. This experiment was designed to study the performance of our methods with increasing $|V(G)|$. In this experiment, we used the ER and SF datasets and fixed $|V(Q)| = 10$ (i.e., the number of vertices in query Q). In addition, the default number of labels was $1,000$. As shown in Fig. 19, although HSM and Partial-Assembly performed worse than NOVA, ASP, and $\text{Turbo}_{\text{ISO}}$ when $|V(G)|$ was small, the gap between them became smaller as $|V(G)|$ became larger. When $|V(G)| = 100$K, our algorithms performed the best over the SF and ER datasets. Note that GADDI did not work when $|V(G)| > 20$K.

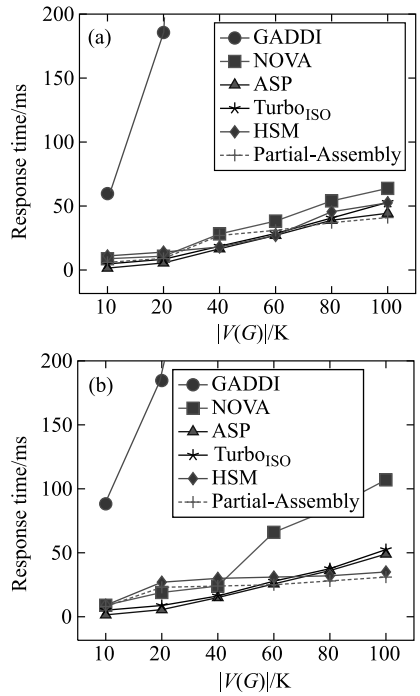


Fig. 19 Online performance comparison with existing vertex-labeled graphs methods over synthetic datasets. (a) Online performance over ER dataset; (b) online performance over SF dataset

7.3 Online performance

7.3.1 Performance based on the number of blocks

In the Partial-Assembly algorithm, a large graph G was divided into several blocks. In this experiment, we studied the effect based on the number of blocks. In Fig. 20(a), we varied the block number from 1 to 2,000 (note that a single block means that the Partial-Assembly algorithm was degraded to HSM algorithm) and fixed $|E(Q)|$ to be 2. We proposed two measures to evaluate the performance. The first was the operation number. As we know, function ExploreGraph(v) is called recursively in both the HSM and Partial-Assembly algorithms. In line 13 of the ExploreGraph(v) algorithm (Algorithm 1), we needed to determine $v' \in TQ(u'')$. The operation number refers to the number of operations necessary for this determination. Figure 20 shows that the number of operations decreased as the pivot number increased from 1 to 1,000 in the US Patent dataset. This was because more blocks led to a reduced search space. However, if the partition number was too large, we had to assemble more small-size local partial matches. Therefore, with an additional increase of the block number from 1,000 to 4,000, the operation number increased accordingly, as shown in Fig. 20(a). We also discovered that

the time shown in Fig. 20(b) had a similar trend as that in Fig. 20(a). Thus, the default pivot number in the US Patent dataset was set to 1,000. Similar observations were made with the Yago2 dataset. However, we do not report these in this section due to space limitations.

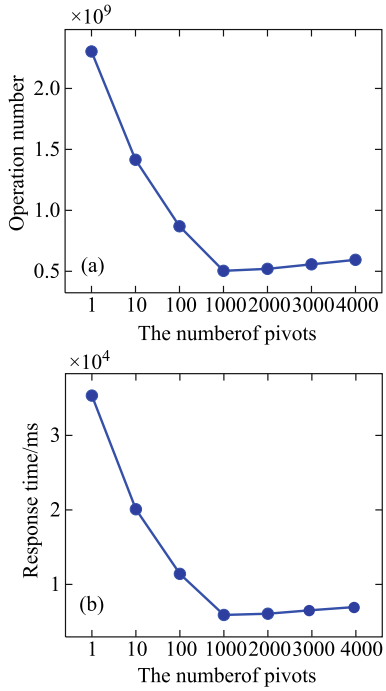


Fig. 20 Effectiveness of multiple pivots. (a) Operation number; (b) total response time

7.3.2 Performance of pivot selection method

This experiment was conducted to study the effectiveness of entropy-based pivot selection (see Section 1). We compared it with some other vertex importance measures, such as vertex degree, vertex betweenness, and random selection. We partitioned the graph into 1,000 blocks and fixed $|E(Q)| = 2$. In addition, we counted the number of operations of Algorithm 3 and reported the total response time. As shown in Fig. 21, the entropy-based solution yielded the best performance. Larger vertex entropy means that the shortest path tree rooted at this vertex had more layers and each layer had fewer vertices. According to the prune rule in Theorem 3, the vertex codes based on this pivot will lead to the minimal search space.

7.4 Offline performance

We next evaluated the offline performance of our method. The index size and construction time are reported in Table 3. The offline processing consisted of three steps. The first and fastest step was graph partitioning. We then identified the vertex having the maximal vertex entropy in each block. In order to speed up this step, we selected only a few sample vertices (we sampled 1,000 vertices in each block) and computed the shortest path trees of these vertices. We utilized these trees to estimate the entropy of all vertices. Thus, the time expended finding pivots was still acceptable. Finally, the vertex with the maximal entropy value was selected as the pivot. We computed the shortest path trees of these pivots offline to obtain the vertex codes of all vertices. Furthermore, the index sizes were not large, as shown in Table 3, because our index has linear space complexity.

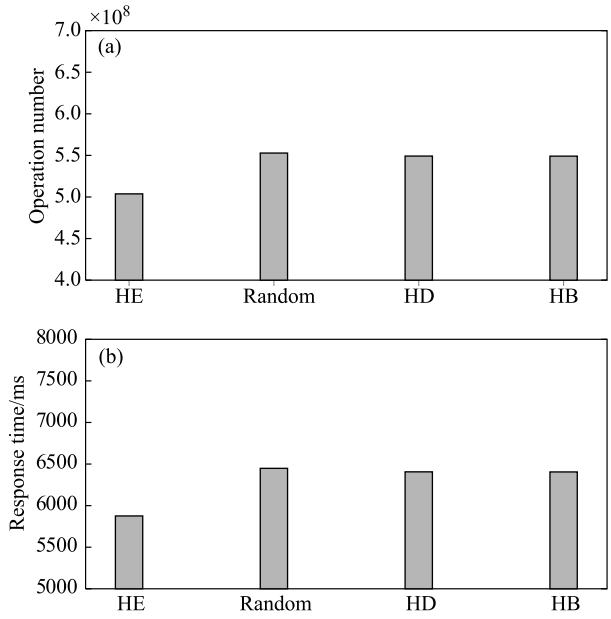


Fig. 21 Effectiveness of our pivot selection method (HE: Highest Entropy; Random: Random Selection; HD: Highest Degree; HB: Highest Betweenness). (a) Operation number; (b) total time

Table 3 Index size and index construction time

Dataset	Index construction time/min				Index size/MB
	Partitioning graph	Finding pivots	Computing vertex codes	Total	
US Patent	2.4	34.2	22.8	59.4	167.659
Yago2	5.4	223.8	30	259.2	523.026

7.5 Index maintenance

We evaluated the performance of our index maintenance method by simulating a random sequence of 1,000 edge insertions/deletions. Table 4 presents the average time for a single edge insertion/deletion.

Table 4 Maintenance performance

Dataset	Average insertion time/ms	Average deletion time/ms
US Patent	0.145	0.016
Yago2	249.3	26.1

8 Related work

Ullmann [33] and VF2 [34] are the two early efforts to address the subgraph isomorphism problem. In order to speed up the query response time, most subgraph search methods pre-compute some structural indices to reduce the search space. They assume that the data graph is a vertex-labeled graph (i.e., each vertex has a single label). These structural indices are built based on vertex labels. For example, SPath [18] constructs an index using neighborhood signature that summarizes vertex labels within the k-neighborhood subgraph of each vertex. For each vertex in a data graph, NOVA [23] uses a vector to store the label distribution of its neighborhood vertices. ASP [19] divides all edges in a data graph into several classes according to vertex labels and uses bitmap structures to index them. SSP [35] extends ASP and proposes some optimizations to further improve query performance. Obviously, these methods cannot be adapted to a data graph with various kinds of vertex-specific content. Another problem of existing methods is the super-linear space complexity of the index structure. Lee et al. [36] reimplemented some of the aforementioned methods and provided a fair comparison of them. They then presented a solution called $\text{Turbo}_{\text{ISO}}$ [22]. $\text{Turbo}_{\text{ISO}}$ defines a concept of the *neighborhood equivalence class (NEC)*. All query vertices in the same NEC have the same matching data vertices. Thus, when $\text{Turbo}_{\text{ISO}}$ finds all subgraph matches, only combinations for each NEC are generated. Turbo_{HOM++} [32] further extends $\text{Turbo}_{\text{ISO}}$ to handle SPARQL queries over RDF graphs. BoostIso [37] extends the concept of neighborhood equivalence class in the data graph and defines four types of relationships between vertices in the data graph to further reduce duplicate computation.

Recently, some subgraph isomorphism approaches have been conducted in distributed environments. Sun et al. [21] used the Microsoft distributed graph database system, Trinity, to answer the subgraph query. However, this method assumes that the whole graph is cached in memory. If a graph has some large-sized vertex/edge contents, such as texts, this assumption cannot hold. In [38], the authors used Map-Reduce and applied the multiway join to find all matches. This study focused on how to speed up the process of the multiway join.

In order to avoid graph isomorphism computation, many works have revised the definition of graph matching [39–43]. Fan et al. [39–41] and Ma et al. [42] defined graph matching based on graph simulation, and Ness [43] used an information propagation model to redefine graph matching. With these revisions, graph matching can be performed in polynomial time.

Although holistic pattern matching has been studied extensively in XML databases, XML adopts a "tree" as the underlying structure, which is different from a general "graph". These holistic pattern matching methods in XML databases [26, 27] adopt a tree code to determine ancestor-descendant relationships in order to support XQuery. The basic idea of holistic pattern matching is to reduce the number of intermediate results. To the best of our knowledge, we are the first to propose a holistic pattern matching for the subgraph query problem.

Partial evaluation and assembly solution for graph problems were proposed by Buneman et al. [44–47]. In [44–46], the focus was on Boolean XPath queries over an XML tree. In [47], the authors discussed how to deal with reachability queries over distributed graphs. Partial evaluation-based graph simulation was well studied in Fan et al. [39] and Ma et al. [42]. However, SPARQL query semantics is different from these and pose additional challenges. In this study, we discuss how to use this framework for the subgraph search problem. The main contribution of this part of our study is the partial computing results derived from addressing the subgraph search problem.

9 Conclusion

In this study, we investigated the subgraph search problem over a large general graph. We proposed a distance-based vertex code. Based on the codes, we proposed a holistic subgraph matching algorithm to address the subgraph search problem. To further improve the performance, we proposed a "partial evaluation and assembly" framework to reduce the search space. Extensive experiments over large real datasets confirmed the superiority of our solutions.

Acknowledgements This work was partially supported by the National Key Research and Development Program of China (2016YFB1000603), Fundamental Research Funds for the Central Universities, the National Natural Science Foundation of China (Grant Nos. 61622201, 61472131, and 61272546), and Science and Technology Key Projects of Hunan Province (2015 TP1004).

References

1. Zhang S J, Li S R, Yang J. GADDI: distance index based subgraph matching in biological networks. In: Proceedings of the 12th International Conference on Extending Database Technology. 2009, 192–203
2. Watts D J, Dodds P S, Newman M E J. Identity and search in social networks. Science, 2002, 296(5571): 1302–1305
3. Stocker M, Seaborne A, Bernstein A, Kiefer C, Reynolds D. SPARQL

basic graph pattern optimization using selectivity estimation. In: Proceedings of the 17th International Conference on World Wide Web. 2008, 595–604

4. Cohen E, Halperin E, Kaplan H, Zwick U. Reachability and distance queries via 2-hop labels. SIAM Journal on Computing, 2003, 32(5): 1338–1355
5. Chan E P F, Lim H. Optimization and evaluation of shortest path queries. The VLDB Journal, 2007, 16(3): 343–369
6. Jing N, Huang Y W, Rundensteiner E A. Hierarchical encoded path views for path query processing: an optimal model and its performance evaluation. IEEE Transactions on Knowledge and Data Engineering, 1998, 10(3): 409–432
7. Cheng J F, Yu J X. On-line exact shortest distance query processing. In: Proceedings of the 12th International Conference on Extending Database Technology. 2009, 481–492
8. Wang H X, He H, Yang J, Yu P S, Yu J X. Dual labeling: answering graph reachability queries in constant time. In: Proceedings of the 22nd International Conference on Data Engineering. 2006, 75
9. Trißl S, Leser U. Fast and practical indexing and querying of very large graphs. In: Proceedings of the ACM SIGMOD International Conference on Management of Data. 2007, 845–856
10. Chen Y J, Chen Y B. An efficient algorithm for answering graph reachability queries. In: Proceedings of the 24th International Conference on Data Engineering. 2008, 893–902
11. Shasha D, Wang J T L, Giugno R. Algorithmics and applications of tree and graph searching. In: Proceedings of the 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems. 2002, 39–52
12. Yan X F, Yu P S, Han J W. Graph indexing: a frequent structure-based approach. In: Proceedings of the ACM SIGMOD International Conference on Management of Data. 2004, 335–346
13. He H H, Singh A K. Graphs-at-a-time: query language and access methods for graph databases. In: Proceedings of the ACM SIGMOD International Conference on Management of Data. 2008, 405–418
14. Zhang S J, Hu M, Yang J. TreePi: a novel graph indexing method. In: Proceedings of the 23rd International Conference on Data Engineering. 2007, 966–975
15. Zhao P X, Yu J X, Yu P S. Graph indexing: tree + delta >= graph. In: Proceedings of the 33rd International Conference on Very Large Data Bases. 2007, 938–949
16. He H H, Singh A K. Closure-tree: an index structure for graph queries. In: Proceedings of the 22nd International Conference on Data Engineering. 2006, 38
17. Tian Y Y, Patel J M. TALE: a tool for approximate large graph matching. In: Proceedings of the 24th International Conference on Data Engineering. 2008, 963–972
18. Zhao P X, Han J W. On graph query optimization in large networks. Proceedings of the VLDB Endowment, 2010, 3(1-2): 340–351
19. Peng P, Zou L, Chen L, Lin X M, Zhao D Y. Subgraph search over massive disk resident graphs. In: Proceedings of the 23rd International Conference on Scientific and Statistical Database Management. 2011, 312–321
20. Sakr S, Elnikety S, He Y X. G-SPARQL: a hybrid engine for querying large attributed graphs. In: Proceedings of the 21st ACM International Conference on Information and Knowledge Management. 2012, 335–344
21. Sun Z, Wang H Z, Wang H X, Shao B, Li J Z. Efficient subgraph matching on billion node graphs. Proceedings of the VLDB Endowment, 2012, 5(9): 788–799
22. Han W S, Lee J, Lee J H. Turbo$_{iso}$: towards ultrafast and robust subgraph isomorphism search in large graph databases. In: Proceedings of the ACM SIGMOD International Conference on Management of Data. 2013, 337–348
23. Zhu K, Zhang Y, Lin X M, Zhu G P, Wang W. NOVA: a novel and efficient framework for finding subgraph isomorphism mappings in large graphs. In: Proceedings of the 15th International Conference on Database Systems for Advanced Applications. 2010, 140–154
24. Zou L, Chen L, Özsu M T. DistanceJoin: pattern match query in a large graph database. Proceedings of the VLDB Endowment, 2009, 2(1): 886–897
25. Yu J X, Zeng X G, Cheng J F. Top-k graph pattern matching over large graphs. In: Proceedings of IEEE International Conference on Data Engineering. 2013, 1033–1044
26. Bruno N, Koudas N, Srivastava D. Holistic twig joins: optimal XML pattern matching. In: Proceedings of ACM SIGMOD International Conference on Management of Data. 2002, 310–321
27. Jiang H F, Wang W, Lu H J, Yu J X. Holistic twig joins on indexed XML documents. In: Proceedings of the 29th International Conference on Very Large Data Bases. 2003, 273–284
28. Karypis G, Kumar V. Analysis of multilevel graph partitioning. In: Proceedings of ACM/IEEE Conference on Supercomputing. 1995, 29
29. Wang L, Xiao Y H, Shao B, Wang H X. How to partition a billion-node graph. In: Proceedings of the 30th International Conference on Data Engineering. 2014, 568–579
30. Tretyakov K, Armas-Cervantes A, García-Ba nuelos L, Vilo J, Dumas M. Fast fully dynamic landmark-based estimation of shortest path distances in very large graphs. In: Proceedings of the 20th ACM Conference on Information and Knowledge Management. 2011, 1785–1794
31. Hoffart J, Suchanek F M, Berberich K, Weikum G. YAGO2: a spatially and temporally enhanced knowledge base from Wikipedia. Artificial Intelligence, 2013, 194: 28–61
32. Kim J, Shin H, Han W S, Hong S, Chafi H. Taming subgraph isomorphism for RDF query processing. Proceedings of the VLDB Endowment, 2015, 8(11): 1238–1249
33. Ullmann J R. An algorithm for subgraph isomorphism. Journal of the ACM, 1976, 23(1): 31–42
34. Cordella L P, Foggia P, Sansone C, Vento M. A (sub) graph isomorphism algorithm for matching large graphs. IEEE Transactions on Pattern Analysis and Machine Intelligence, 2004, 26(10): 1367–1372
35. Peng P, Zou L, Chen L, Lin X M, Zhao D Y. Answering subgraph queries over massive disk resident graphs. World Wide Web: Internet and Web Information Systems, 2016, 19(3): 417–448
36. Lee J, Han W S, Kasperovics R, Lee J H. An in-depth comparison of subgraph isomorphism algorithms in graph databases. Proceedings of the VLDB Endowment, 2012, 6(2): 133–144
37. Ren X G, Wang J H. Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs. Proceedings of the VLDB Endowment, 2015, 8(5): 617–628
38. Afrati F N, Fotakis D, Ullman J D. Enumerating subgraph instances us-

ing Map-Reduce. In: Proceedings of the 29th International Conference on Data Engineering. 2013, 62–73

39. Fan W F, Wang X, Wu Y H, Deng D. Distributed graph simulation: impossibility and possibility. Proceedings of the VLDB Endowment, 2014, 7(12): 1083–1094
40. Fan W F, Li J Z, Ma S, Tang N, Wu Y H, Wu Y P. Graph pattern matching: from intractable to polynomial time. Proceedings of the VLDB Endowment, 2010, 3(1): 264–275
41. Fan W F, Wang X, Wu Y H. Diversified top-k graph pattern matching. Proceedings of the VLDB Endowment, 2013, 6(13): 1510–1521
42. Ma S, Cao Y, Huai J P, Wo T Y. Distributed graph pattern matching. In: Proceedings of the 21st World Wide Web Conference. 2012, 949–958
43. Khan A, Li N, Yan X F, Guan Z Y, Chakraborty S, Tao S. Neighborhood based fast graph search in large networks. In: Proceedings of the ACM SIGMOD International Conference on Management of Data. 2011, 901–912
44. Buneman P, Cong G, Fan W F, Kementsietsidis A. Using partial evaluation in distributed query evaluation. In: Proceedings of the 32nd International Conference on Very Large Data Bases. 2006, 211–222
45. Cong G, Fan W F, Kementsietsidis A. Distributed query evaluation with performance guarantees. In: Proceedings of the ACM SIGMOD International Conference on Management of Data. 2007, 509–520
46. Cong G, Fan W F, Kementsietsidis A, Li J Z, Liu X M. Partial evaluation for distributed XPath query processing and beyond. ACM Transactions on Database Systems, 2012, 37(4): 32
47. Fan W F, Wang X, Wu Y H. Performance guarantees for distributed reachability queries. Proceedings of the VLDB Endowment, 2012, 5(11): 1304–1315

Peng Peng received his BS and PhD degrees in computer science from Beijing Normal University, China and Peking University, China in 2009 and 2016, respectively. He is currently an assistant professor at Hunan University, China. His research interests include graph databases and distributed database systems.

Lei Zou received his BS and PhD degrees in computer science from Huazhong University of Science and Technology, China in 2003 and 2009, respectively. He is currently an associate professor at Peking University, China. His research interests include graph databases and knowledge graph data management.

Zhenqin Du was an intern student in the Institute of Computer Science and Technology of Peking University (PKU), China. He joined the research project in the ICST of PKU concerning graph data management and knowledge graph application.

Dongyan Zhao received the BS, MS, and PhD degrees from Peking University (PKU), China in 1991, 1994, and 2000, respectively. He is currently a professor at PKU. His research interests are in information processing and knowledge management, including computer networks and intelligent agents.