Noname manuscript No.
(will be inserted by the editor)

Answering Pattern Match Queries in Large Graph Databases Via Graph Embedding*

Lei Zou · Lei Chen · M. Tamer Özsu · Dongyan Zhao

the date of receipt and acceptance should be inserted later

Abstract The growing popularity of graph databases has generated interesting data management problems, such as subgraph search, shortest path query, reachability verification, and pattern matching. Among these, a pattern match query is more flexible compared to a subgraph search and more informative compared to a shortest path or a reachability query. In this paper, we address *distance-based pattern match* queries over a large data graph G. Due to the huge search space, we adopt a filter-and-refine framework to answer a pattern match query over a large graph. We first find a set of candidate matches by a graph embedding technique and then evaluate these to find the exact matches. Extensive experiments confirm the superiority of our method.

1 Introduction

As one of the most popular and powerful representations, graphs have been used to model many application data, such as social networks, biological networks, and World Wide Web. In order to conduct effective analysis over graphs, various types of queries have been investigated, such as shortest path query [9,18, 7], reachability query [9,29,27,5], and subgraph query [24,34,6,17,31,36,15,25]. These are all interesting, but in this paper, we focus on pattern match queries, since they are more flexible than subgraph queries and more informative than simple shortest path or reachability queries. Specifically, a pattern match query searches over a large labeled graph to look for the existence of a pattern graph in a large data graph. A pattern match query is different from subgraph search in that it only specifies the vertex labels and connection constraints between vertices. In other words, a pattern match query emphasizes the connectivity between labeled vertices rather than checking subgraph isomorphism as subgraph search does.

In this paper, we propose a distance-based pattern match query, which is defined as follows: given a large graph G, a query graph Q with n vertices and a parameter δ, n vertices in G *match* Q iff: (1) these n vertices in G have the same labels as the corresponding vertices in Q, and (2) for any two adjacent vertices v_i and v_j in Q (i.e., there is an edge between v_i and v_j in Q and $1 \leq i, j \leq n$), the *distance* between two corresponding vertices in G is no larger than δ. We need to find all matches of Q in G. In this work, we use the shortest path to measure the distance between two vertices, but our approach is not restricted to this distance function, and can be applied to other *metric* distance functions as well. Note that, for ease of presentation, we use the term "pattern match" instead of "distance-based pattern match" in the rest of this paper, when the context is clear.

A key problem of pattern match queries is huge search space. Given a query Q with n vertices, for each vertex v_i in Q, we first find a list of vertices in data graph G that have the same labels as that of v_i. Then,

Extended version of paper "Distance-Join: Pattern Match Query In a Large Graph Database" that was presented in the Proceeding of 35th International Conference on Very Large Databases (VLDB), pages 886-897, 2009.

Lei Zou ✉, Dongyan Zhao
Peking University, Beijing, China
E-mail: {zoulei,zdy}@icst.pku.edu.cn

Lei Chen
Hong Kong University of Science and Technology, Hong Kong
E-mail: leichen@cse.ust.hk

M. Tamer Özsu
University of Waterloo, Waterloo, Canada
E-mail: Tamer.Ozsu@uwaterloo.ca

for each pair of adjacent vertices v_i and v_j in Q, we need to find all matching pairs in G whose distances are less than δ. This is called an *edge query*. To answer an edge query, we need to conduct a *distance-based join* operation between two lists of matching vertices corresponding to v_i and v_j in G. Therefore, finding pattern Q in G requires a sequence of *distance-based join* operations, which is very costly for large graphs. In order to answer pattern match queries efficiently, we adopt the filter-and-refine framework. Specifically, we need efficient pruning strategies to reduce the search space. Although many effective pruning techniques have been proposed for subgraph search (e.g., [24,34,6,17, 31,36,15,25]), they can not be applied to pattern match queries since these pruning rules are based on the necessary condition of subgraph isomorphism. We propose a novel and effective method to reduce the search space significantly. Specifically, we transform vertices into points in a vector space via graph embedding methods, converting a pattern match query into a distance-based multi-way join problem over the vector space to find candidate matches. In order to reduce the join cost, we propose several pruning rules to reduce the search space further, and propose a cost model to guide the selection of the join order to process multi-way join efficiently.

During the refinement process, we adopt 2-hop distance label [9] to compute the shortest path distance for each candidate match. Unfortunately, finding the optimal 2-hop distance-aware labeling (i.e., one where the size of 2-hop distance-aware labels is minimized) is a NP-hard problem [9]. Although a heuristic method to compute 2-hop distance-aware labels in a large directed graph has been proposed [7], this method cannot work well in a large undirected graph. We will discuss this method in detail in Section 2. In this paper, we propose a *betweenness* (Definition 3) estimation-based method to guide 2-hop distance-aware center selection. Extensive experiments on both real and synthetic datasets confirm the efficiency of our method.

To summarize, in this work, we make the following contributions:

1) We propose a general framework for handling pattern match queries over a large graph. Specifically, we adopt a filter-and-refine framework to answer pattern match queries. During filtering, we map vertices into vectors via an embedding method and conduct distance-based multi-way join over the vector space.

2) We design an efficient distance-based join algorithm (D-join for short) for an edge query in the converted vector space, which well utilizes the block nested loop join and hash join techniques to handle the high dimensional vector space. We also develop an effective cost model to estimate the cost of each join operation, based on which we can select the most efficient join order to reduce the cost of multi-way join.

3) In order to address the high complexity of offline processing, we propose a graph partitioning-based method and bi-level version of D-join algorithm, called bD-join.

4) In order to enable shortest path distance computation efficiently, we propose betweenness estimation-based method to compute 2-hop distance labels in a large graph.

5) In order to answer an approximate subgraph query (Definition 9), we first transform it into a distance-based pattern match query. Then, we find candidates by D-join algorithm. Finally, for each candidate, we verify whether it is an approximate subgraph match (Definition 8).

6) Finally, we conduct extensive experiments with real and synthetic data to evaluate the proposed approaches.

The rest of this paper is organized as follows. We discuss the related work in Section 2. Our framework is presented in Section 3. We propose betweenness estimation-based method to compute 2-hop distance labels in Section 4. The offline process is discussed in Section 5. We discuss the neighbor area pruning technique in Section 6, and a distance-based join algorithm for an edge query and its cost model in Section 7. Section 7 also presents a distance-based multi-way join algorithm for a pattern match query and join order selection method. In Section 8, we propose a graph partition-based method to reduce the cost of offline processing and the bD-join algorithm. In Section 9, we propose a distance-join based solution to answer approximate subgraph queries. We study our methods by experiments in Section 10. Section 11 concludes this paper.

2 Background and Related Work

Let $G = \langle V, E \rangle$ be a graph where V is the set of vertices and E is the set of edges. Given two vertices u_1 and u_2 in G, a *reachability query* verifies if there exists a path from u_1 to u_2, and a *distance query* returns the shortest path distance between u_1 and u_2 [9]. These are well-studied problems, with a number of vertex labeling-based solutions [9]. A family of labeling techniques have been proposed to answer both reachability and distance queries. A 2-hop labeling method over a large graph G assigns to each vertex $u \in V(G)$ a label $L(u) = (L_{in}(u), L_{out}(u))$, where $L_{in}(u), L_{out}(u) \subseteq V(G)$. Vertices in $L_{in}(u)$ and $L_{out}(u)$ are called *centers*. There are two kinds of 2-hop labeling: 2-hop reachability labeling (reachability labeling for short) and 2-hop distance

labeling (distance labeling for short). For reachability labeling, given any two vertices $u_1, u_2 \in V(G)$, there is a path from u_1 to u_2 (denoted as $u_1 \rightarrow u_2$), if and only if $L_{out}(u_1) \cap L_{in}(u_2) \neq \phi$. For distance labeling, we can compute $Dist_{sp}(u_1, u_2)$ using the following equation.

$$Dist_{sp}(u_1, u_2) = \min\{Dist_{sp}(u_1, w) + Dist_{sp}(u_2, w) | \\ w \in (L_{out}(u_1) \cap L_{in}(u_2))\} \tag{1}$$

where $Dist_{sp}(u_1, u_2)$ is the shortest path distance between vertices u_1 and u_2. The distances between vertices and centers (i.e., $Dist_{sp}(u_1, w)$ and $Dist_{sp}(u_2, w)$) are pre-computed and stored. The size of 2-hop labeling is defined as $\sum_{u \in V(G)} (|L_{in}(u)| + |L_{out}(u)|)$, while the size of 2-hop distance labeling is $O(|V(G)||E(G)|^{1/2})$ [8]. Thus, according to Equation 1, we need $O(|E(G)|^{1/2})$ time to compute the shortest path distance by distance labeling because the average vertex distance label size is $O(|E(G)|^{1/2})$.

Note that, this work focuses on the shortest path distance computation. Finding the optimal 2-hop distance labeling is NP-hard [9]. Therefore, a heuristic algorithm based on the minimal set-cover problem is proposed [9]. Initially, all pairwise shortest paths are computed in G (denoted by D_G). Then, in each iteration, one vertex w in $V(G)$ is selected as a 2-hop center to maximize Equation 2.

$$\frac{|D_{(G,w)} \cap D_G|}{|A_w| + |D_w|} \tag{2}$$

where $D_{(G,w)}$ denotes the shortest paths that are *covered* by w, and A_w contains all vertices that can reach w and D_w contains all vertices that are reachable from w. Note that, A_w and D_w are also called the ancestor and descendant clusters of w.

Then, all paths in $D_{(G,w)}$ are removed from D_G. This process is iterated until $D_G = \phi$, and all selected 2-hop centers are returned. According to the 2-hop centers, the corresponding ancestor and descendant clusters are built, according to which, a 2-hop label for each vertex in G is generated. Note that, in order to evaluate Equation 2, all pairwise shortest paths (i.e., D_G) need to be kept in memory. Obviously, this requirement is prohibitive for a large graph due to high space complexity $O(|V(G)|^2)$. Furthermore, also due to high time complexity, it is impossible to pre-compute all pairwise shortest paths in a large graph G in a reasonable time.

There are many proposals for computing reachability labeling in a large graph (e.g., [27,29]). However, 2-hop distance-aware labeling computation in a large graph has not attracted much attention except for the work of Cheng and Yu [7], where a large directed graph G is first converted into a directed acyclic graph (DAG) $G^{\downarrow}$ by removing some vertices in each strongly connected component (SCC) of G. These removed vertices are selected as *2-hop centers*. Obviously, all shortest paths that pass through these removed vertices are covered by these selected 2-hop centers. Then, $G^{\downarrow}$ is partitioned into two subgraphs, denoted as $G_{\top}$ and $G_{\perp}$, by a set of node-separators V_w. These vertices in V_w are selected as 2-hop centers as well. All shortest paths across $G_{\top}$ and $G_{\perp}$ must be covered by V_w. These two partitions can be considered separately. If $G_{\top}$ is small enough, the method in [9] can be employed to compute 2-hop labels in $G_{\top}$; otherwise, the partition process is repeated until the 2-hop label in each partition can be computed by the method in [9] directly. The same process is followed for $G_{\perp}$.

Given a 2-hop center w, it is not necessary that A_w (and D_w) contains all ancestor (and descendant) nodes of w in a directed graph G, or all vertices in an undirected graph G. Some pruning methods have been proposed for this purpose based on the previously identified 2-hop clusters [7].

There are two problems in the method of [7]. Firstly, if G is not a sparse directed graph, there may exist a large number of strongly connected components (SCC) in G. Consequently, a large number of vertices need to be removed from G to generate a DAG. This means that the size of 2-hop labeling in G is very large. Furthermore, if G is an undirected graph, it is impossible to generate a DAG by removing some vertices in G. Secondly, the proposed pruning methods reduce the redundancy and the labeling size, but the pruning strategy is based on all previously identified 2-hop clusters. Thus, all previously identified 2-hop clusters need to be cached in memory; otherwise, frequent swap-ins/outs will affect the performance dramatically. Furthermore, the cost of redundancy checking is also expensive.

To the best of our knowledge, there exists little work on pattern match queries over a large data graph, except for [8,26]. In [8], based on the reachability constraint, authors propose a pattern match problem over a large *directed* graph G. Specifically, given a query pattern graph Q (that is a directed graph) that has n vertices, n vertices in G can match Q if and only if these corresponding vertices have the same reachability connection as those specified in Q. This is the most related work to ours, although our constraints are on "distance" instead of "reachability". We call our match "distance pattern match", and the match in [8] "reachability pattern match". We first illustrate the method in [8] using Figure 1, and then discuss how it can be

extended it to solve our problem and present the shortcomings of the extension.

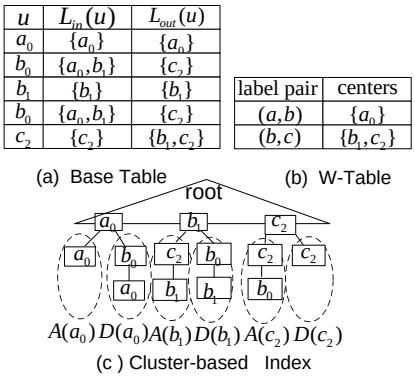


Fig. 1 R-join

Without loss of generality, assume that there is only one directed edge $e = (v_1, v_2)$ in query Q. Figure 1(a) shows a base table to store all vertex distance labels. For each center w_i in graph G, A_{w_i} and D_{w_i} are 2-hop reachability clusters for w_i. A_{w_i} and D_{w_i} contain all ancestor nodes and descendant nodes of w_i in G, respectively, where for every vertex u_1 in A_{w_i}, every vertex u_2 in D_{w_i} can be reached via w_i. Then, an index structure is built based on these clusters, as shown in Figure 1(c). For each vertex label pair (l_1, l_2), all centers w_i are stored (in table W-Table), where there exists at least one vertex labeled l_1 (and l_2) in $A(w_i)$ (and $D(w_i)$). Consider a directed edge $e = (v_1, v_2)$ in query Q and assume that the labels of vertices v_1 and v_2 (in query Q) are 'a' and 'b', respectively. According to W-Table in Figure 1b, we can find centers w_i, in which there exists at least a vertex u_1 labeled 'a' in A_{w_i}, and there exists at least a vertex u_2 labeled 'b' in D_{w_i}. For each such center w_i, the Cartesian product of vertices labeled 'a' in A_{w_i} and vertices labeled 'b' in D_{w_i} can form the matches of Q. This operation is called *R-join* [8]. In this example, there is only one center a_0 that corresponds to vertex label pair (a, b), as shown in Figure 1(b). According to index structure in Figure 1(c), $F(a_0)$ and $T(a_0)$ can be found. When the number of edges in Q is larger than one, a reachability pattern match query can be answered by a sequence of R-joins.

2-hop distance labeling has the structure similar to 2-hop reachability labeling, we can extend the method proposed in [8] to answer distance pattern match query. Specifically, in the last step, for each vertex pair (u_1, u_2) in the Cartesian product, we need to compute $dist = Dist_{sp}(u_1, w_i) + Dist_{sp}(u_2, w_i)$. If $dist \leq \delta$, (u_1, u_2) is a match. Note that this step is different from reachability pattern match in [8], in which no distance computation is needed. Assume that there are n_1 vertices labeled 'a' and n_2 vertices labeled 'b' in a graph G. It is clear that the number of distance computations is at least $n_1 \times n_2$, which is exactly the same as naive join processing. Therefore, this extension method will not reduce the search space. Thus, the motivation of our work is exactly this: *is it possible to avoid unnecessary distance computation to speed up the search efficiency?* Several efficient and effective pruning techniques are proposed in this paper.

The best-effect algorithm [26] returns K matches with large scores. That algorithm cannot guarantee that the k result matches are the k largest over all matches. We cannot extend this method to apply to our problem, since it cannot guarantee the completeness of results. In [11], authors propose ranked twig queries over a large graph, however, a "twig pattern" is a directed graph, not a general graph.

3 Framework

Definition 1 (Distance-based Pattern Match). Consider a data graph G, a connected query graph Q that has n vertices $\{v_1, ..., v_n\}$ and m edges $\{e_1, ..., e_m\}$, and m parameters $\delta_1, ..., \delta_m$. A set of n distinct vertices $\{u_1, ..., u_n\}$ in G is said to be a *Distance-based Pattern Match* of Q, if and only if the following conditions hold:
1) $\forall i, L(u_i) = L(v_i)$, where $L(u_i)$ $(L(v_i))$ denotes u_i's (v_i's) label; and
2) For each edge $e_j = (v_{i_1}, v_{i_2})$ in Q, $j = 1, ..., m$, the shortest path distance between u_{i_1} and u_{i_2} in G is no larger than δ_j. We denote the shortest path between u_{i_1} and u_{i_2} as $\overline{u_{i_1} u_{i_2}}$.

For ease of presentation, we assume that all distance constraint parameters have the same value, i.e., $\delta = \delta_1 = ... = \delta_m$. The problem that we study in this paper is defined as follows:

Definition 2 (Distance-based Pattern Match Query). Consider a data graph G, a connected query graph Q that has n vertices $\{v_1, ..., v_n\}$, and a parameter δ. *A pattern match query* finds all matches (Definition 1) of Q over G.

In this paper, we use the terms "match" and "pattern match query" instead of "distance-based pattern match" and "distance-based pattern match query" for simplicity, when the context is clear.

According to Definition 1, any match is always contained in some connected component of G, since Q is connected. Without loss of generality, we assume that

G is connected. If not, we can sequentially perform pattern match queries in each connected component of G to find all matches.

One way of executing the pattern match query (that we call *naive join processing*) is the following. Using the vertex label predicates associated with each vertex v_i of a query graph Q, derive n lists of vertices $(R_1, \ldots, R_n)$, where each list R_i contains all vertices u_i whose labels are the same as v_i's label (list R_i is said to correspond to a vertex v_i in Q). Then, perform a shortest path distance-based multi-way join over these lists. Join requires the definition of a join order, which, in this case, corresponds to a traversal order in Q. At each traversal step, the subgraph induced by all visited edges (in Q) is denoted as Q'. Figure 2 shows a join order (i.e., traversal order in Q) of a sample query Q. In the first step, there is only one edge in Q', thus, the pattern match query degrades into an *edge query*. After the first step, we still need to answer an edge query for each new encountered edge. It is clear that different join orders will lead to different performance.

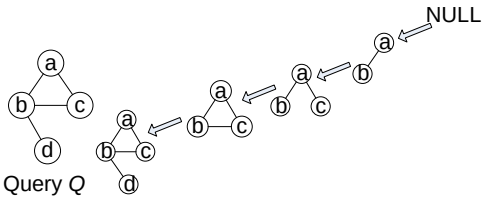


Fig. 2 A Join-Order

According to Definition 1, we have to perform shortest path distance computation online. The straightforward solution to reduce the cost is to pre-compute and store all pairwise shortest path distances (*Pre-compute* method). The method is fast, but prohibitive in space usage (it needs $O(|V(G)|^2)$ space). Graph labeling technique enables the computation of shortest path distance in $O(|E(G)|^{1/2})$ time, while the space cost is only $O(|V(G)||E(G)|^{1/2})$ [9]. Thus, we adopt the graph labeling technique to perform shortest path distance computation. However, computing the optimal 2-hop distance labels (i.e., the size of 2-hop distance labels is minimized) is NP-hard [9]. In Section 4, we propose a betweenness estimation-based method to compute 2-hop distance-aware labels in a large graph G.

The key problem in naive join processing is its large number of distance computations. In order to speed up the query performance, we need to address two issues: reducing the number of shortest path distance computations, and finding a distance computation method to find all candidate matches that are more efficient than shortest path distance computation.

In order to address these issues, we utilize the *LLR embedding* technique [20,23] to map all vertices in G into points in vector space $\Re^k$, where k is the dimensionality of $\Re^k$. We then compute L_∞ distance between the points in $\Re^k$ space, since it is much cheaper to compute and it is the lower bound of the shortest path distance between two corresponding vertices in G (see Theorem 2). Thus, we can utilize L_∞ distance in vector space $\Re^k$ to find candidate matches. We also propose several pruning techniques based on the properties of L_∞ distance to reduce the number of distance computations in join processing. Furthermore, we propose a novel cost model to guide the join order selection. Note that we do not propose a general method for distance-join (also called *similarity join*) in vector space [1,3]; we focus on L_∞ distance in the converted space simply because we use L_∞ distance to find candidate matches.

Figure 3 depicts the general framework to answer a pattern match query. We first use LLR embedding to map all vertices into points in vector space $\Re^k$. We adopt k-medoids algorithm [12] to group all points into different clusters. Then, for each cluster, we map all points u (in this cluster) into a 1-dimensional block. According to the Hilbert curve in $\Re^k$ space, we can define the total order for all clusters. According to this total order, we link all blocks to form a flat file. We also propose a heuristic method to compute 2-hop distance labeling to enable fast shortest path distance computation, which is discussed in Section 4. When query Q is received, according to join order selection algorithm, we find the cheapest query plan (i.e., join order). As discussed above, a join order corresponds to a traversal order in query Q. At each step, we perform an edge query for the new introduced edge. During edge query processing, we first use L_∞ distance to obtain all candidate matches (Definition 6), then, we compute the shortest path distance for each candidate match to fix final results. Join processing is iterated until all edges in Q are visited.

4 Computing 2-Hop Distance-Aware Label

In this section, we propose a "betweenness" based method to compute 2-hop distance-aware labels in a large graph. The relative importance of a vertex in a graph can be evaluated using measures based on the centrality of a vertex in graph theory, such as degree centrality, betweenness, closeness, and eigenvector centrality [22]. Among these measures, "betweenness" measures the relative importance of a vertex that is needed by others when connecting along shortest paths [10], where vertices that occur on many shortest paths between other

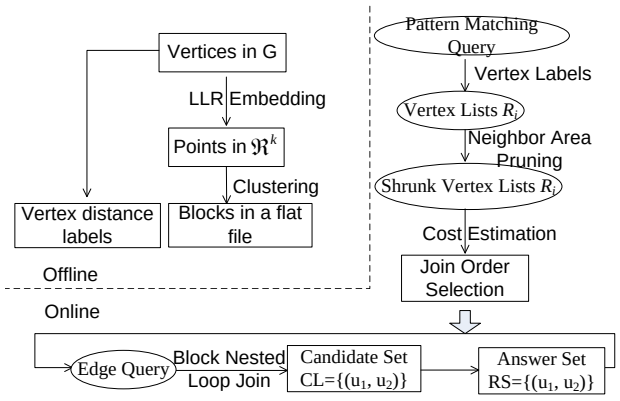
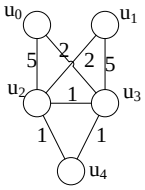


Fig. 3 Framework of Pattern Match Query



(a) Graph G

vertex	2-hop Labels
u_0	$L_{in}(u_0)=\{(w_0,2),(w_1,3)\}$;
u_1	$L_{in}(u_1)=\{(w_0,2),(w_1,3)\}$;
u_2	$L_{in}(u_2)=\{(w_0,0),(w_1,1)\}$;
u_3	$L_{in}(u_3)=\{(w_0,1),(w_1,1)\}$;
u_4	$L_{in}(u_4)=\{(w_0,1),(w_1,1)\}$;

(b) 2-hop labeling with centers $\{w_0=u_2, w_1=u_3\}$

vertex	2-hop Labels
u_0	$L_{in}(u_0)=\{(w_0,0),(w_1,5),(w_2,3)\}$;
u_1	$L_{in}(u_1)=\{(w_0,2),(w_1,5),(w_2,3)\}$;
u_2	$L_{in}(u_2)=\{(w_0,3),(w_1,2),(w_3,1)\}$;
u_3	$L_{in}(u_3)=\{(w_0,2),(w_1,3),(w_2,1)\}$;
u_4	$L_{in}(u_4)=\{(w_0,3),(w_1,3),(w_2,0)\}$;

(c) 2-hop labeling with centers $\{w_0=u_0, w_1=u_1, w_2=u_4\}$

Fig. 4 2-hop Labeling With Different 2-hop Centers

vertices have higher betweenness value than those that do not.

Definition 3 [10] Given a graph G and a vertex u, if vertex u occurs on the shortest distance path between u_1 and u_2 ($u_1 \neq u_2 \neq u$), we say that u *covers* the shortest distance path between u_1 and u_2. The *betweenness* of v is defined as follows:

$$Betweenness(u) = \sum_{u_1 \neq u \neq u_2 \in V} \frac{\sigma_{u_1 u_2}(u)}{\sigma_{u_1 u_2}} \quad (3)$$

where $\sigma_{u_1 u_2}(u)$ denotes the number of different shortest distance paths between u_1 and u_2 that are covered by u; and $\sigma_{u_1 u_2}$ denotes the number of different shortest distance paths between u_1 and u_2.

This measure fits our requirements best. In order to guarantee the *completeness* (defined in Section 2) of 2-hop labeling, we need to select some 2-hop centers to cover all pairwise shortest paths. Meanwhile, in order to minimize the space cost of 2-hop labeling, the number of 2-hop centers should be minimized. Thus, it is better to select some vertices that can cover a large number of shortest paths as 2-hop centers, i.e., selecting some vertices that have higher betweenness as 2-hop centers in our method.

Given an undirected graph G in Figure 4a, u_2 and u_3 are two vertices with the highest betweenness, where $Betweenness(u_2) = Betweenness(u_3) = 0.7$. If we select u_2 and u_3 as 2-hop centers, all shortest paths in G are covered by u_2 or u_3. Figure 4b shows the corresponding 2-hop labeling for G. However, if we select other 2-hop centers, such as u_0, u_1, u_4, Figure 4c shows the corresponding 2-hop labeling. Obviously, the size of the former 2-hop labeling is smaller than the latter.

Motivated by the above observation, we propose a betweenness-based method to compute 2-hop labeling for a graph G in Algorithm 1. Firstly, we select some vertices with high betweenness values in G as 2-hop centers (Line 1 in Algorithm 1). It is very expensive to compute betweenness in a large graph, since it has the same time complexity as computing all pairwise shortest paths. Therefore, we propose to adopt a sampling approach to estimate betweenness [2]. Specifically, we first randomly select 1% of the vertices G as "pivots". The experimental results show that the random sampling method has precision similar to some carefully-designed sampling approaches [2]. Thus, we adopt the random sampling method. More details about "pivot selection" and "betweenness estimation" can be found in [2].

According to the shortest paths between these pivots and Equation 4, we can estimate betweenness for each vertex in G as follows.

$$ESTBetweeness(u) = \sum_{u_1 \neq u \neq u_2, u_1, u_2 \in V'} \frac{\sigma_{u_1 u_2}(u)}{\sigma_{u_1 u_2}} \quad (4)$$

where V' denotes the set of pivot vertices, $\sigma_{u_1 u_2}(u)$ denotes the number of different shortest distance paths between u_1 and u_2 that are covered by u; and $\sigma_{u_1 u_2}$ denotes the number of different shortest distance paths between u_1 and u_2.

We select the top-k vertices with the highest estimated betweenness values as 2-hop centers, denoted as $W_b = \{w_b\}$. As discussed earlier, we focus on a *connected undirected* graph G. For each center w_b in W_b, the ancestor and descendant clusters (denoted as A_{w_b} and D_{w_b}) contain all vertices in a G, since all vertices can reach the center w_b and these vertices are also reachable from w_b in G.

We then remove all vertices (including their adjacent edges) in W_b from G to obtain G', and partition G' into n segments. The set of node separators is denoted as $W_s = \{w_s\}$ (Lines 2-3 in Algorithm 1). We require that the size of each partition is small enough to be processed by the method in [9]. We employ METIS algorithm [19] to perform graph partitioning. Since METIS

performs an edge separator-based partition, we adopt an approach similar to [13] to convert it into a node separator-based partition. Considering one edge separator $e = (u_1, u_2)$ that connects two partitions P_1 and P_2, we shift the partition boundary so that one of u_1 and u_2 is on the boundary and the other one is in the partition. The node on the boundary is called a *node separator*. The choice of node separator depends on the size of partitions P_1 and P_2. In order to balance the size of each partition, we always make the node from the large partition a node separator.

We remove all vertices in $W_b \cup W_s$ and their adjacent edges from G to obtain $G^\diamond$ (Line 4). Assume that there are m connected components in $G^\diamond$. For purposes of presentation, we assume that each connected component is called a *subgraph*, denoted S_i, $i = 1, ..., m$. It is clear that $m \geq n$, and each subgraph S_i is small enough to be processed by the method in [9], according to the graph partition method in the second step.

All pairwise shortest paths in G can be classified into two categories, denoted as $PathSet_1$ and $PathSet_2$, where $PathSet_1$ contains all paths that cross at least two subgraphs, and $PathSet_2$ contains the paths that are constrained in each subgraph.

In order to guarantee the completeness of 2-hop labeling, the process should consist of two parts, namely *skeleton 2-hop labeling* and *local 2-hop labeling*. The former covers all paths in $PathSet_1$, and the latter covers all paths in $PathSet_2$. For each subgraph S_i, we employ the method in [9] to compute local 2-hop labeling (Lines 5-6). The key issue is how to compute skeleton 2-hop labeling. Obviously, all paths in $PathSet_1$ are covered by at least one vertex in $W_b \cup W_s$. For each vertex $w \in W_b \cup W_s$, we perform Dijkstra's algorithm to obtain the shortest path tree with root w. The ancestor and descendant clusters for w (denoted as A_w and D_w) contain all vertices in G and their corresponding distances. However, the space cost of this straightforward approach is quite high. Some pruning methods based on previously identified clusters are proposed in [7]. However, the key problem of this technique is that all previously identified clusters need to be kept in memory for further checking. Otherwise, frequent swaps will affect the performance dramatically, making it prohibitive in a large graph.

An interesting observation is that a large fragment of shortest paths in $PathSet_1$ are covered by W_b, but $|W_b| = k << |W_s|$. Therefore, we propose the following method to reduce redundancies in 2-hop clusters: For each vertex $w_b \in W_b$, A_{w_b} (and D_{w_b}) contains all vertices in G and their corresponding distances (Lines 7-9). However, it is not necessary for A_{w_s} and D_{w_s} ($w_s \in W_s$) to contain all vertices in G, since most paths have been covered by some 2-hop centers in W_b.

Theorem 1 *Given a 2-hop center $w_s \in W_s$ and a vertex u in G, if the shortest path between w_s and u (denoted as $\overline{uw_s}$) passes through some vertex $w_b \in W_b$, u can be filtered out from A_{w_s} and D_{w_s} without affecting the completeness of 2-hop labeling.*

Proof See [37].

According to Theorem 1, we have the following steps. For each 2-hop center $w_b \in W_b$, we perform Dijkstra's algorithm to obtain the shortest path tree with root w_b. The 2-hop cluster for $w_b \in W_b$ contains all vertices in G and the corresponding distances (Lines 7-9). For each 2-hop center $w_s \in W_s$, we also perform Dijkstra's algorithm to obtain the shortest path tree with root w_s (Line 11). If the shortest path between w_s and some vertex u (denoted as $\overline{w_s u}$) passes through another 2-hop center w_b, u can be filtered out from A_{w_s} and D_{w_s} (Lines 15-16). According to the 2-hop clusters, it is straightforward to obtain the skeleton 2-hop labels (Line 17).

As mentioned earlier, 2-hop label of each vertex contains two parts, a local 2-hop label and a skeleton 2-hop, which can be obtained in Lines 5-6 and Lines 7-17 of Algorithm 1, respectively. Finally, for each vertex, we combine the local 2-hop labels and the skeleton 2-hop labels together (Line 18).

5 Offline Processing For Pattern Match Query

5.1 Graph Embedding Technique

According to LLR embedding technique [20, 23], we have the following embedding process to map all vertices in G into points in a vector space $\Re^k$, where k is the dimensionality of the vector space:

1) Let $S_{n,m}$ be a subset of randomly selected vertices in $V(G)$. We define the distance from u to its closest neighbor in $S_{n,m}$ as follows:

$$Dist(u, S_{n,m}) = min_{u' \in S_{n,m}}\{Dist_{sp}(u, u')\} \quad (5)$$

2) We select $k = O(log^2|V(G)|)$ subsets to form the set $R = \{S_{1,1}, ..., S_{1,\kappa}, ..., S_{\beta,1}, ..., S_{\beta,\kappa}\}$. where $\kappa = O(log|V(G)|)$ and $\beta = O(log|V(G)|)$ and $k = \kappa \cdot \beta = O(log^2|V(G)|)$. Each subset $S_{n,m}$ ($1 \leq n \leq \beta$, $1 \leq m \leq \kappa$) in R has 2^n vertices in $V(G)$.

3) The mapping function $E : V(G) \rightarrow \Re^k$ is defined as follows:

$$E(u) = [Dist(u, S_{1,1}), ..., Dist(u, S_{1,\kappa}), ..., Dist(u, S_{\beta,1}), ..., Dist(u, S_{\beta,\kappa})]$$

Algorithm 1 **B**etwneenness **e**stimation-based 2-hop Labeling Computing (BE for short)

Input: **Input**: graph G **Output**: 2-hop labeling for G.

1: Select k vertices in G with the highest estimated betweenness as 2-hop centers, denoted as W_b.
2: Remove W_b from G to form G'.
3: Partition G' into n segments by a set of node-separators, denoted as W_s. The size of each partition is small enough to be handled by the method in [9].
4: Remove $W_b \cup W_s$ from G to form $G^\diamond$. Each connected component is called a subgraph S_i.
5: **for** each S_i in G **do**
6: Employ the method in [9] to compute the local 2-hop labeling.
7: **for** each vertex $w \in W_b$ **do**
8: Perform Dijkstra's algorithm to find the shortest distance path tree rooted at w.
9: The 2-hop cluster A_w and D_w contain all vertices in G and the corresponding distance.
10: **for** each vertex w in W_s **do**
11: Perform Dijkstra's algorithm to find the shortest distance path tree rooted at w.
12: **for** each vertex u in G **do**
13: **if** the shortest distance path between w and u does not pass through $w' \in W_b$ **then**
14: Insert u into the cluster A_w and D_w.
15: Generate skeleton 2-hop labeling according to 2-hop clusters.
16: Combine the local 2-hop labels and the skeleton 2-hop labels together.

(6)

where $\kappa \cdot \beta = k$.

In the converted vector space $\Re^k$, we use L_∞ metric as distance function in $\Re^k$, which is defined as follows:

$$L_\infty(E(u_1), E(u_2)) = max_{n,m}|Dist(u_1, S_{n,m}) - Dist(u_2, S_{n,m})|$$

where $E(u_1)$ is the point (in $\Re^k$ space) corresponding to the vertex u_1 in graph G. For notational simplicity, we also use u_1 to denote the point in $\Re^k$ space, when the context is clear. Theorem 2 establishes L_∞ distance over $\Re^k$ as the lower bound of the shortest path distance over G.

Theorem 2 *[23] Given two vertices u_1 and u_2 in G, L_∞ distance between two corresponding points in the converted vector space $\Re^k$ is the lower bound of the shortest path distance between u_1 and u_2; that is $L_\infty(E(u_1), E(u_2)) \leq Dist_{sp}(u_1, u_2)$*

The time complexity and space cost of offline processing are $O(|E(G)|\ |V(G)| + |V(G)|^2 log|V(G)|)$ and $O(|V(G)||E(G)|^{\frac{1}{2}} + |V(G)|\ log^2|V(G)|)$, respectively. Please refer to [37] for detailed analysis.

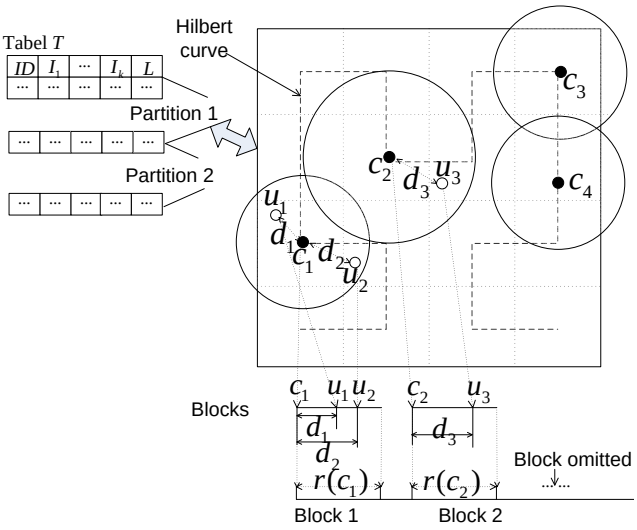


Fig. 5 Table Format of The Converted Vector Space

5.2 Data Structures

Note that shortest path distance and L_∞ distance are both metric distances [12]; thus they satisfy the triangle inequality. After LLR embedding, all vertices are mapped into points in high dimensional space $\Re^k$. We use a relational table $T(ID, I_1, ..., I_k, L)$ (as shown in Figure 5) to store all points in $\Re^k$. The first ID column is the vertex ID, columns I_1, ..., I_k are k dimensions of a mapped point in $\Re^k$, and the last column L denotes the vertex label. To avoid the sequential scan over table T, we organize all points in T by the following method: First, we divide the whole table into different partitions according to vertex labels. The orders for partitions are randomly defined, but we record the partition start and end offsets in a secondary file. Second, for each partition, we group all points in this partition into different clusters. For each cluster C_i, we find its *cluster center* c_i. For each point u in cluster C_i whose center is c_i, according to distance $L_\infty(u, c_i)$ (c_i is cluster center of C_i), u is mapped into 1-dimensional block B_i. Clearly, different clusters are mapped into different blocks. We define *cluster radius* $r(C_i)$ as the maximal distance between center c_i and vertex u in cluster C_i. Figure 5 depicts our method, where Euclidean distance is used as the distance function for demonstration (the actual distance function is L_∞, but that is harder to show). We define the total order for different clusters according to Hilbert order in the high dimensional space. Consider two clusters C_1 and C_2 whose cluster centers are c_1 and c_2 respectively. Assuming c_1 and c_2 are in two different cells S_1 and S_2 (in $\Re^k$ space, as shown in Figure 5) respectively, if cell S_1 is ahead of S_2 in Hilbert order,

cluster C_1 is larger than C_2. If c_1 and c_2 are in the same cell, the order of C_1 and C_2 is arbitrarily defined.

Note that the maximal size for one cluster (i.e., one block) is specified according to the allocated memory size. In our implementation, we use K-medoids algorithm [12] to find clusters. Also note that the clustering algorithm is orthogonal to our approach. How to find an optimal clustering in $\Re^k$ is beyond the scope of this paper. In the following discussion, we assume that clustering results are given.

6 Neighbor Area Pruning

To answer a distance pattern match query, we conduct a distance-based join over the converted vector space, not the original graph space. Thus, to reduce the cost of join processing, the first step is to remove all the points that do not qualify for join condition as early as possible. In this section, we propose an efficient pruning strategy called *neighbor area* pruning.

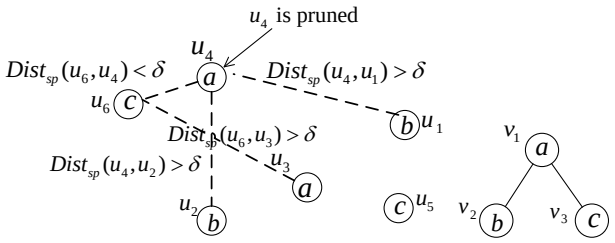


Fig. 6 Area Neighbor Pruning

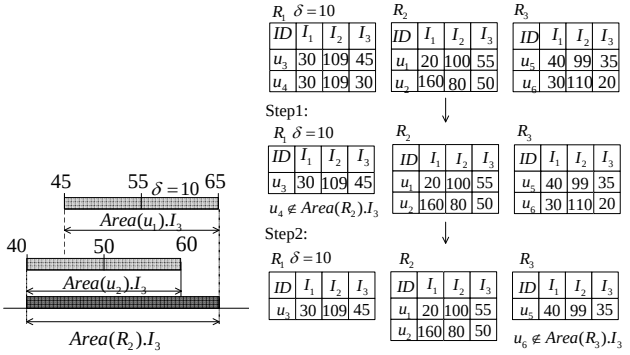
We first illustrate the rationale behind neighbor area pruning using Figure 6. Consider query Q in Figure 6. If a vertex u labeled 'a' (in G) can match v_1 (in Q) according to Definition 1, there must exist another vertex u' labeled 'b' (in G), where $Dist_{sp}(u, u') \leq \delta$, since v_1 has a neighbor vertex labeled 'b' in query Q. For vertex u_4 in Figure 6, there exists no vertex u' labeled with 'b', where $Dist_{sp}(u_4, u') \leq \delta$; thus, u_4 can be pruned safely. Vertex u_6 has label 'c', thus, it is a candidate match to vertex v_3 in query Q. Although there exists a vertex u_4 labeled 'a', where $Dist_{sp}(u_6, u_4) < \delta$, pruning vertex u_4 in the last step will lead to pruning u_6 as well. In other words, neighbor area pruning is an iterative process that continues until convergence is reached (i.e., no vertices in any list can be further pruned).

As a result of LLR embedding, all vertices in G have been mapped into points in $\Re^k$. Therefore, we want to conduct neighbor area pruning over the converted space. Since L_∞ distance is the lower bound for the shortest path distance, vertex u_4 can be pruned safely, if there exists no vertex u' labeled with 'b' where $L_\infty(u_4, u') \leq \delta$. However, it is inefficient to check each vertex one-by-one. Therefore, we propose the *neighbor area* pruning to reduce the search space in $\Re^k$.

Definition 4 Given a vertex v in query Q and its corresponding list R in data graph G, for each vertex u_i in R, according to the LLR embedding technique, each vertex u_i is mapped to a point in $\Re^k$ space, denoted as $u_i^{\Re^k}$. We define *vertex neighbor area* of u_i as follows: $Area(u_i) = ([(u_i^{\Re^k}.I_1 - \delta, u_i^{\Re^k}.I_1 + \delta), ..., (u_i^{\Re^k}.I_k - \delta, u_i^{\Re^k}.I_k + \delta)])$, where $u_i^{\Re^k}$ is a point in $\Re^k$ space, and I_i is the i-th dimension at $\Re^k$ space. The *list neighbor area* of R_i is defined as follows: $Area(R) = [(Area(u_1).I_1 \cup ... \cup Area(u_n).I_1), ..., (Area(u_1).I_k \cup ... \cup Area(u_n).I_k)]$, where $u_1, ..., u_n \in R$. Note that, $Area(R).I_j = Area(u_1).I_j \cup ... \cup Area(u_n).I_j$, $j = 1, ..., k$.

For notational simplicity, we also use "u_i" to denote the corresponding point $u_i^{\Re^k}$ in $\Re^k$ space, when the context is clear.

Definition 5 Given a list R and a vertex u, $u \in Area(R)$ if and only if, for every dimension I_j, $u.I_j \in Area(R).I_j$, where $Area(R).I_j$ is defined in Definition 4.



(a) An Example of $Area(R_2).I_3$ (b) An Example of Algorithm 2

Fig. 7 Running Examples

Given a data graph G in Figure 6, assume that the converted space $\Re^k$ is given in Figure 7(b) and list R_2 contains all vertices labeled 'b'. Since there are two vertices u_1 and u_2 in R_2, $Area(u_1).I_3$ and $Area(u_2).I_3$ are shown in Figure 7(a). $Area(R_2).I_3 = (Area(u_1).I_3 \cup Area(u_2).I_3)$ is also shown in Figure 7(a). Obviously, $u_3.I_3 \in Area(R_2).I_3$, since $45 \in [40, 65]$. However, $u_4.I_3 \notin Area(R_2).I_3$.

Theorem 3 *Given a vertex v in query Q, its corresponding list is R in G. Assume that v has m neighbor*

vertices v_j (i.e., (v, v_j) is an edge), $j = 1, ..., m$, and for each vertex v_j, its corresponding list is R_j in G. Given a vertex $u_i \in R$, if $\exists j, u_i \notin Area(R_j)$, then u_i can be safely pruned from the list R.

In Figure 7(b), according to Theorem 3, no vertices can be filtered out from R_2 and R_3. However, for vertex u_4 in R_1, $u_4.I_3 \notin Area(R_2).I_3 = [40, 60] \cup [45, 65] = [40, 65]$. Thus, u_4 is pruned safely from R_1 in the first step. The more interesting thing is that pruning R_1 leads to shrinking of $Area(R_1).I_3$. In this way, u_6 is pruned in the second step from R_3. Now, R_1, R_2 and R_3 converge.

Algorithm 2 Neighbor Area Pruning

Input: Query Q that has n vertices v_i; and each v_i has a corresponding list R_i.
Output: n lists R_i after pruning.
1: **while** numLoop < MAXNUM **do**
2: **for** each list R_i **do**
3: Scan R_i to find $Area(R_i)$.
4: **for** each list R_i **do**
5: Scan R_i to filter out false positives by $Area(R_j)$, where v_j is a neighbor vertex w.r.t v_i.
6: **if** all list R_i has not been change in this loop **then**
7: Break

Based on Theorem 3, Algorithm 2 lists the steps to perform pruning on each list R_i. Notice that, as discussed above, the pruning process is iterative. Lines 2-5 are repeated until either the convergence is reached (Lines 6-7), or iteration step exceeds the maximal number of iterations (Line 1).

Time Complexity. It is straightforward that the time complexity of Lines 2-5 is linear with respect with the number of vertices in R_i, i.e., $O(\sum_i |R_i|)$. According to Line 1 in Algorithm 2, the algorithm iterates until that the convergence is reached, or at most $MAXNUM$ times. In each iteration, there is at least one vertex to be pruned. There are $(\sum_i |R_i|)$ vertices in total. Therefore, the total time complexity of Algorithm 2 is $O((\sum_i |R_i|)^2)$. Note that, in practice, the complexity is far less than the complexity in the worst case.

7 Answering Pattern Query via Edge Join

In this section, we propose "edge join" and answer a pattern match query by a series of edge joins. To complete this task, we need to define a join order. In fact, a join order in our problem corresponds to a traversal order in Q. In each traversal step, the subgraph induced by all visited edges (in Q) is denoted as Q'. We can find all matches of Q' in each step. Figure 2 shows a join order (i.e., traversal order in Q) of a sample query Q. In the first step, there is only one edge in Q', thus, the pattern match query degrades into an *edge query*. After the first step, we still need to answer an edge query for each new encountered edge.

7.1 Edge Join

As discussed above, at each step, we need to answer an edge query. In this subsection, we propose an efficient edge join algorithm. We first use L_∞ distance in the converted vector space $\Re^k$ to find a candidate match set CL, where *candidate match* is defined as follows:

Definition 6 Given an edge query $Q_e = (v_1, v_2)$ over a graph G and a parameter δ, vertex pair (u_1, u_2) is a *candidate match* of Q_e if and only if:
(1) $L(v_1) = L(u_1)$ and $L(v_2) = L(u_2)$ where $L(u_i)$ $(L(v_i))$ indicates label of u_i (v_i); and
(2) $L_\infty(u_1, u_2) \leq \delta$.

Each candidate match in CL is a pair of vertices (u_i, u_j) $(i \neq j)$, where $L_\infty(u_i, u_j) \leq \delta$. Then, for each candidate (u_i, u_j), we utilize 2-hop distance labels to compute the exact shortest path distance $Dist_{sp}(u_i, u_j)$ [9,7]. All pairs (u_i, u_j) where $Dist_{sp}(u_i, u_j) \leq \delta$ are collected to form the final result RS. Theorem 4 proves that the above process guarantees *no false negatives*.

Theorem 4 *Given an edge query $Q_e = (v_1, v_2)$ over a graph G, and a parameter δ, let CL denote the set of candidate matches of Q_e, and RS denote the set of all matches of Q_e. Then, $RS \subseteq CL$.*

Proof Straightforward from Theorem 2.

Essentially, an edge join is a similarity join over vector space. Existing similarity join algorithms (such as RSJ [3] and EGO [1]) can be utilized to find candidate matches over the vector space $\Re^k$. However, there are two important issues to be addressed in answering an edge query. First, the converted space $\Re^k$ is a high dimensional space, where $k = O(log^2|V(G)|)$. In our experiments, we choose 15-30 dimensions when $|V(G)| = 10K \sim 100K$. R-tree based similarity join algorithms (such as RSJ [3]) cannot work well due to the *dimensionality curse* [16]. Second, although some high-dimensional similarity join algorithms have been proposed, they are not optimized for L_∞ distance, which we use to find candidate matches. To address these key issues, we first propose the following edge join algorithm to reduce both I/O and CPU costs.

We adopt block nested loop join strategy in edge join algorithm. Given an edge query $Q_e = (v_1, v_2)$, let R_1 and R_2 be the lists of candidate vertices (in G) that satisfy vertex label predicates associated with v_1 and

v_2, respectively. Let R_1 be the "outer" and R_2 be the "inner" join operand. Edge join algorithm reads one block B_1 from R_1 in each step. In the inner loop, it is not necessary to perform join processing between B_1 and all blocks in R_2. We only load "promising" blocks B_2 into memory in the inner loop. Then, we perform memory join algorithm between B_1 and B_2. Theorem 5 shows the necessary condition that B_2 is a promising block with regard to B_1. Due to memory size constraint, we only load one promising block in each step.

Theorem 5 *Given two blocks B_1 and B_2 (the "outer" and "inner" join operands, respectively), edge join between B_1 and B_2 produces a non-empty result only if*

$$L_\infty(c_1, c_2) \leq r(C_1) + r(C_2) + \delta$$

where C_1 (C_2) is the corresponding cluster of block B_1 (B_2), c_1 (c_2) is C_1's (C_2)'s cluster center, and $r(C_1)$ ($r(C_2)$) is C_1's (C_2's) cluster radius.

Proof See [37].

After the nested loop join, we can find all candidate matches for edge query. Then, for each candidate match (u_1, u_2), we use 2-hop distance labeling technique to compute the shortest path distance between u_1 and u_2, that is, $Dist_{sp}(u_1, u_2)$. If $Dist_{sp}(u_1, u_2) \leq \delta$, (u_1, u_2) will be inserted into answer set RS. The detailed steps of edge join algorithm are shown in Algorithm 3.

Algorithm 3 Distance-based *Edge Join* Algorithm

Input: : An edge $e = (v_1, v_2)$ in query Q, where $L(v_1)$ (and $L(v_2)$) denotes the vertex label of vertex v_1 (and v_2). The distance constraint is δ. R_1, the set of candidate vertices for matching v_1 in e. R_2, the set of candidate vertices for matching v_2 in e.
Output: Answer set $RS = \{(u_1, u_2)\}$, where $L(u_1) = L(v_1)$ AND $L(u_2) = L(v_2)$ AND $Dist_{sp}(u_1, u_2) \leq \delta$.

```
1: Initialize candidate set CL and answer set RS.
2: for each cluster C1 in R1 do
3:    Load C1 into memory
4:    According to Theorem 5, find all promising clusters w.r.t
      C1 in flat file F to form cluster set PC.
5:    Order all clusters in PC according to physical position in
      table T.
6:    for each promising cluster C2 in PC do
7:       Load cluster C2 into memory.
8:       Perform memory-based edge join algorithm (call Algo-
         rithm 4) on C1 and C2 to find candidate set CL1.
9:       Insert CL1 into CL.
10: for each candidate match (u1, u2) in CL do
11:    Compute Dist_sp(u1, u2) by graph labeling techniques.
12:    if Dist_sp(u1, u2) ≤ δ then
13:       Insert (u1, u2) into answer set RS
14: Report RS
```

For a pair of blocks B_1 and B_2 that are loaded in memory, we need to perform a join efficiently. We achieve this by pruning using triangle inequality and by applying hash join.

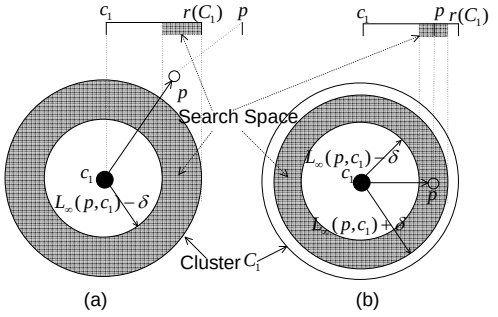


Fig. 8 Search Space after pruning based on Theorem 6

7.1.1 Triangle Inequality Pruning

The following theorem specifies how the number of distance computations can be reduced based on triangle inequality.

Theorem 6 *Given a point p in block B_2 (the inner join operand) and a point q in block B_1 (the outer join operand), the distance between p and q needs to be computed only when the following condition holds:*

$$Max(L_\infty(p, c_1) - \delta, 0) \leq L_\infty(q, c_1) \leq Min(L_\infty(p, c_1) + \delta, r(C_1))$$

where C_1 (C_2) is the cluster corresponding to B_1 (B_2), c_1 (c_2) is the center of C_1 (C_2), and $r(C_1)$ is the radius of C_1.

Proof See [37].

Figure 8 visualizes the search space in cluster C_1 with regard to point p in C_2 after pruning according to Theorem 6.

7.1.2 Hash Join

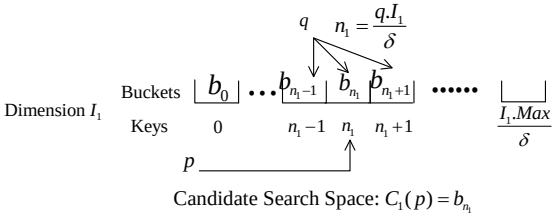


Fig. 9 Hash Join

Hash join in a well-known join algorithm with good performance. The classical hash join does not work for edge join processing, since it can only handle equi-join.

Consider two blocks B_1 and B_2 (the outer and inner join operands). For purposes of presentation, we first assume that there is only one dimension (I_1) in $\Re^k$, i.e., $k = 1$. The maximal value in I_1 is defined as $I_1.Max$. We divide the interval $[0, I_1.Max]$ into $\lceil \frac{I_1.Max}{\delta} \rceil$ buckets for dimension I_1. Given a point q in block B_1 (the outer operand), we define hash function $H(q) = n_1 = \lfloor \frac{q.I_1}{\delta} \rfloor$. Then, instead of hashing q into a single bucket, we put q into *three* buckets, $(n_1 - 1)^{th}$, n_1^{th}, and $(n_1 + 1)^{th}$ buckets. To save space, we only store q's ID in different buckets. Based on this revised hashing strategy, we can reduce the search space, which is described by the following theorem.

Theorem 7 *Given a point p in block B_2 (inner join operand), according to hash function $H(p) = n_i = \lfloor \frac{p.I_i}{\delta} \rfloor$, p is located at the n_i^{th} bucket. It is only necessary to perform join processing between p and all points of B_i located in the n_i^{th} bucket. The candidate search space for point p is $Can_i(p) = b_{n_i}$, where b_{n_i} denotes all points in the n_i^{th} bucket.*

Proof It can be proven using L_∞ distance definition.

Figure 9 demonstrates our proposed hash join method. When $k > 1$ (i.e., higher dimensionality), we build buckets for each dimension I_i ($i = 1, ..., k$). Consider a point p (the inner join operand) from block B_2 and obtain candidate search space $Can_i(p)$ in dimension I_i, $i = 1, ..., k$. Theorem 8 establishes the final search space of p using hash join.

Theorem 8 *The overall search space for vertex p is $Can(p) = Can_1(p) \cap Can_2(p) \cap ... \cap Can_k(p)$, where $Can_i(p)$ ($i = 1, ..., k$) is defined in Theorem 7.*

Theorem 9 shows that, for a join pair (q, p) (p from B_1 and q from B_2, respectively), if $Dist_\infty(q, p) > 2\delta$, the join pair (q, p) can be safely pruned by the hash join.

Theorem 9 *Consider two blocks B_1 and B_2 (the outer and inner join operands) to be joined in memory. For any point p in B_2, the necessary and sufficient condition that a point q is in p's search space (i.e., $q \in C(p)$) is $L_\infty(p, q) \leq 2\delta$.*

Proof It can be proven according to Theorems 7 and 8.

The memory edge join algorithm is proposed in Algorithm 4, which uses triangle inequality pruning and hash join to reduce join space.

Algorithm 4 Memory edge join Algorithm

Input: : An edge $e = (v_1, v_2)$ in query Q. Two clusters are C_1 and C_2. The distance constraint is δ. R_1 is the set of candidate vertices that match v_1; R_2 is the set of candidate vertices that match v_2.
Output: Answer set $CL = \{(u_1, u_2)\}$, where $L(u_1) = L(v_1)$ AND $L(u_2) = L(v_2)$ AND $L_\infty(u_1, u_2) \leq \delta$.

```
1: for each vertex p in C2 do
2:   if p ∈ R2 then
3:     According to Theorem 6, find search space in C1 with
       regard to p, denoted as SP(p).
4:     Using hash join in Theorem 8, find search space Can(p).
5:     Final search space with regard to p is SP(p) = SP(p) ∩
       Can(p).
6:     for each point q in the search space SP(p) do
7:       if L∞(q, p) ≤ δ then
8:         Insert (q, p) into candidate set CL
9: Report CL.
```

7.2 Pattern Match Query via Edge Join

We start with the assumption that the join order is specified; we relax this in the next section. As discussed in Section 3, a join order corresponds to a traversal order in query graph Q. According to given traversal order, we visit one edge $e = (v_i, v_j)$ in Q in each step. If vertex v_j is the new encountered vertex (i.e., v_j has not been visited yet), edge $e = (v_i, v_j)$ is called a *forward edge*; if v_j has been visited before, e is called a *backward edge*. The processing of a forward edge query and a backward edge query are different. Essentially, forward edge processing is performed by an edge join algorithm (i.e., Algorithm 3), while backward edge processing is a selection operation, which will be discussed shortly.

Definition 7 Given a query graph Q, a subgraph Q' induced by all visited edges in Q is called a *status*. All matches of Q (and Q') are stored in a relational table $MR(Q)$ (and $MR(Q')$), in which columns correspond to vertices v_i in Q (and Q').

The pattern match query via edge join algorithm (Algorithm 5) performs a sequential move from the initial status NULL to final status Q, as shown in Figure 2. Consider two adjacent statuses Q'_i and Q'_{i+1}, where Q'_i is a subgraph of Q'_{i+1} and $|E(Q'_{i+1})| - |E(Q'_i)| = 1$. Let $e = (Q'_{i+1} \setminus Q'_i)$ denote an edge in Q'_{i+1} but not in Q'_i. If e is the first edge to be visited in query Q, we can get the matches of e (denoted as $MR(e)$) by edge join processing (Line 4 in Algorithm 5). Otherwise, there are two cases to be considered.

Forward edge processing: If $e = (v_i, v_j)$ is a forward edge, we can obtain $MR(Q'_j)$ as follows: 1) we first project table $MR(Q')$ over column v_i to obtain list R_i (Line 9 in Algorithm 5). We can obtain list R_j (by scanning the original table T before joining processing

in Line 1) that corresponds to vertex v_j, according to v_j's label. Note that, R_j is a shrunk list after neighbor area pruning (Line 2); 2) According to the edge join algorithm (Algorithm 3), we find the matches for edge e, denoted as $MR(e)$ (Line 10); 3) We perform traditional natural join over $MR(Q'_i)$ and $MR(e)$ to obtain $MR(Q'_j)$ based on column v_i (Line 11).

Backward edge processing: If $e = (v_i, v_j)$ is a backward edge, we scan the intermediate table $MR(Q'_i)$ to filter out all vertex pairs (u_i, u_j), where u_i and u_j correspond to vertices v_i and v_j in query Q, and $Dist_{sp}(u_i, u_j) > \delta$. After filtering $MR(Q'_i)$, we obtain the matches of Q'_{i+1}, i.e., $MR(Q'_{i+1})$. Essentially, it is a selection operation based on the distance constraint (Line 13), defined as follows: $MR(Q'_{i+1}) = \sigma_{(Dist_{sp}(r.v_i, r.v_j) \leq \delta)}(MR(Q'_i))$.

The above steps are iterated until the final status Q is reached (Lines 6-13).

7.3 Cost Model

It is well-known that each join order in Algorithm 5 will lead to a different performance. The join order selection is based on the cost estimation of edge query. The cost of edge join algorithm has three components: the cost of block nested loop join (Lines 2-10 in Algorithm 3), the cost of computing the exact shortest path distance (Lines 12-14), and the cost of storing answer set RS (Line 15). Note that the matches of an edge query are intermediate results for graph pattern query. Therefore, similar to cost analysis for structural join in XML databases [32], we also assume that intermediate results are stored in a temporary table on disk. We use a set of factors to normalize the cost of edge join algorithm. These factors are f_R: the average cost of loading one block into memory; f_D: the average cost of L_∞ distance computation; f_S: the average cost of shortest path distance computation; and f_{IO}: the average cost of storing one match on disk. Given an edge query $Q_e = (v_1, v_2)$ and a parameter δ, R_1 (R_2) is the list of candidate vertices for matching v_1 (v_2). All vertices in R_1 (R_2) are stored in $|B_1|$ ($|B_2|$) blocks in a flat file F. The cost of Algorithm 3 can be computed as follows:

$$Cost(e) = |B_1| \times |B_2| \times \gamma_1 \times f_R + |R_1| \times |R_2| \times \gamma_2 \times f_D + |CL| \times f_S + |CL| \times \gamma_3 \times f_{IO} \quad (7)$$

where γ_1, γ_2, and γ_3 are defined as follows.

$$\gamma_1 = \frac{|AccessedBlocks|}{|B_1| \times |B_2|}, \gamma_2 = \frac{|DisComp|}{|R_1| \times |R_2|}, \gamma_3 = \frac{|RS|}{|CL|} \quad (8)$$

and where $|AccessedBlocks|$ is the number of accessed blocks in Algorithm 3; $|DisComp|$ is the number of L_∞ distance computations and $|RS|$ (and $|CL|$) is the cardinality of answer set RS (and candidate set CL). We use the following methods to estimate γ_1, γ_2 and γ_3.

1) Offline: We pre-compute γ_1, γ_2 and γ_3. Notice that γ_1, γ_2 and γ_3 are related to *vertex labels* and the *distance constraint* δ. Thus, according to historical query logs, the maximal value of δ is $\overline{\delta}$. We partition $[0, \overline{\delta}]$ into z intervals, each with width $d = \lceil \frac{\overline{\delta}}{z} \rceil$. In order to compute the statistics γ_1, γ_2 and γ_3 for vertex label pair (l_1, l_2) and the distance constraint δ in the ith interval $[(i-1)d, i*d]$ $(1 \leq i \leq z)$, we set $\delta = (i - 1/2)d$, and there is only one edge $e = (v_1, v_2)$ in query graph Q where $L(v_1) = l_1$ and $L(v_2) = l_2$. We perform edge join and compute γ_1, γ_2 and γ_3 using Equation 8.

2) Online: Given an edge query $Q_e = (v_1, v_2)$, we look up the estimates for γ_1, γ_2 and γ_3 that were computed offline using the vertex label $(L(v_1), L(v_2))$ and δ.

Algorithm 5 Multiway Distance-Join Algorithm (MD-Join)

Input: Input: A query graph Q that has n vertices and a parameter δ and a large graph G and a table T for the converted vector space $\Re^k$, and the join order MDJ.
Output: $MR(Q)$: All matches of Q in G.

1: for each vertex v_i in query Q, find its corresponding list R_i, according to v_i's label.
2: Obtain Shrunk lists R_i $(i = 1, ..., n)$ by neighbor area pruning.
3: Set $e = (v_1, v_2)$.
4: Obtain $MR(e)$ by edge join algorithm (call Algorithm 3).
5: set $Q'_i = e$.
6: **while** $Q'_i != Q$ **do**
7: According to join order MDJ, e is the next traversal edge.
8: **if** e is forward edge, denoted as $e = (v_i, v_j)$ **then**
9: $R_i = \sigma_{t.ID \in (\prod_{v_i} MR(Q'_i))}(T)$.
10: $MR(e) = \prod_{(R_i.ID, R_j.ID)} (R_i \underset{\mathrm{Dist_{sp}}(r_i, r_j) \leq \delta}{\bowtie} R_j)$ (call Algorithm 3)
11: $MR(Q'_{i+1}) = MR(Q'_i) \underset{v_i}{\bowtie} MR(e)$
12: **else**
13: $MR(Q'_{i+1}) = \sigma_{(Dist_{sp}(r.v_i, r.v_j) \leq \delta)}(MR(Q'_i))$
14: Report $MR(Q)$.

Given an edge query $Q_e = (v_1, v_2)$, the cardinality of candidate match set CL can be computed as $|CL| = |R_1| \times |R_2| \times \theta$ where θ is the selectivity of D-join based on L_∞ distance. Therefore, the key issue is how to estimate θ. We propose two methods, one based on probability-model and the other on sampling.

7.4 Probability Model For Estimating θ

Given an edge query $Q_e = (v_1, v_2)$, according to vertex label, we can find two vertex lists R_1 and R_2. For the purposes of presentation, let us first assume that $k = 1$, i.e., there is one dimension in the converted space. We can regard $R_1.I_1$ and $R_2.I_1$ as two random variables x and y. Let $z = |x-y|$ denote the joint random variable. Selectivity θ equals to the probability of $z \leq \delta$. Figure 10(a) visualizes the joint random variable z and the area Ω between two curves $y = x + \delta$ and $y = x - \delta$. We can compute selectivity θ as follows:

$$\theta = Pr(z \leq \delta) = \iint\limits_{|x-y|\leq\delta} f(x,y)d(x,y) = \iint\limits_{(x,y)\in\Omega} f(x,y)d(x,y)$$

where $f(x, y)$ denotes z's density function. We use two-dimensional histogram method to estimate $f(x, y)$. Specifically, we use equi-width histograms that partition (x, y) data space into t^2 regular buckets (where t is a constant called the histogram resolution), as shown in Figure 10(b). Similar to other histogram methods, we also assume that the distribution in each bucket is uniform. Then, we use a systematic sampling technique [14] to estimate the density function in each bucket.

The basic idea of systematic sampling is the following: Given a relation R with N tuples that can be accessed in ascending/descending order on the join attribute of R, we select n sample tuples as follows: select a tuple at random from the first $\lceil \frac{N}{n} \rceil$ tuples of R and every $\lceil \frac{N}{n} \rceil$th tuple thereafter [14]. The relations here are R_1 and R_2, and the join attributes are $R_1.I_1$ and $R_2.I_1$, where R_1 and R_2 are both from table T. We assume that there exists a B^+-tree index on each dimension I_i in table T, allowing tuples to be accessed in ascending/descending order. We select $|R_1| \times \lambda$ vertices from R_1, and all these selected vertices are collected to form subset SR_1, where λ is a sampling ratio. The same is done for subset SR_2 from the list R_2.

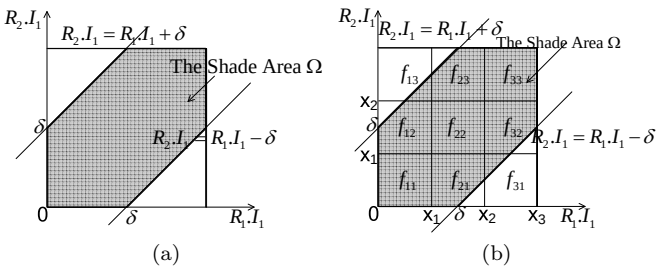


Fig. 10 Selectivity Estimation

We map $SR_1 \times SR_2$ into different two-dimensional buckets. For each bucket A, we use $|A|$ to denote the number of points (from $SR_1 \times SR_2$) that fall into bucket A. The joint density function of points in bucket A is denoted as

$$f(A) = \frac{|A|}{|SR_1| \times |SR_2|}. \quad (9)$$

Some buckets are partially contained in the shared area Ω. The number of points (from $R_1 \times R_2$) that fall into both bucket A and the shared area Ω (denoted as $|A \cap \Omega|$) can be estimated as:

$$|A \cap \Omega| = R_1 \times R_1 \times f(A) \times \frac{area(A \cap \Omega)}{area(A)}$$

where $area(A \cap \Omega)$ denotes the area of intersection between A and Ω and $area(A)$ denotes the area of A.

We adopt Monte-Carlo methods to estimate $\frac{area(A\cap\Omega)}{area(A)}$. Specifically, we first randomly generate a set of points in bucket A (the number of generated records is a). The number of points that fall in Ω is b. Then, we estimate $\frac{area(A\cap\Omega)}{area(A)}$ to be $\frac{a}{b}$.

Therefore, we have $|CL| = \sum_{ij} |A_{ij} \cap \Omega| = |R_1| \times |R_2| \times \sum_{ij} (f(A_{ij}) \times \frac{area(A_{ij}\cap\Omega)}{area(A_{ij})})$

The selectivity of θ can be estimated as follows

$$\theta = Pr(z \leq \delta) = \sum_{ij} |A_{ij} \cap \Omega| = \sum_{ij} (f(A_{ij}) \times \frac{area(A_{ij}\cap\Omega)}{area(A_{ij})}) \quad (10)$$

where $f(A_{ij})$ is estimated by Equation 9.

If $k > 1$, according to Theorem 4, we have

$$CL = R_1 \bowtie_{Max_{1\leq i\leq k}(|R_1.I_i - R_2.I_i| \leq \delta)} R_2$$

The cardinality of $|CL|$ is $|CL| = |R_1| \times |R_2| \times \theta$,where θ is the selectivity of D-join based on L_∞ distance. We can regard $R_1.I_i$ and $R_2.I_i$ $(i = 1, ..., k)$ as random variables x_i and y_i. Let $z_i = |x_i - y_i|$ denote the joint random variable.

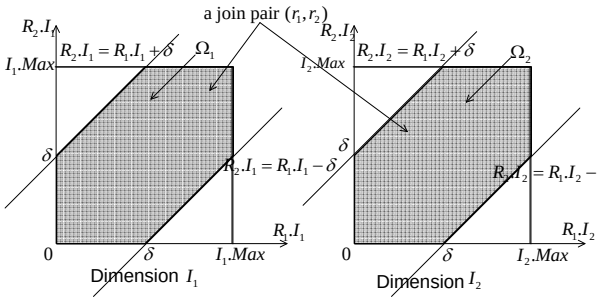


Fig. 11 Multi-Dimension Selectivity Estimation

$$\theta = Pr(Max(z_1, ..., z_k) \leq \delta)) = Pr((z_1 \leq \delta) \wedge ...(z_k \leq \delta))$$

(11)

In order to compute Equation 11, we assume that every dimension I_i in vector space $\Re^k$ is independent of each other. Thus, we have

$$Pr((z_1 \leq \delta) \wedge ... \wedge (z_k \leq \delta)) = Pr(z_1 \leq \delta) \times ... \times Pr(z_k \leq \delta). \quad (12)$$

where $Pr(z_i \leq \delta)$ $(i = 1, ..., k)$ can be computed using Equation 11.

7.5 Sampling-based Method

Consider two lists R_1 and R_2 to be joined. Assume, for simplicity, $k = 2$. In Figure 11, $Pr(Max(z_1, z_2) \leq \delta))$ is the probability that a vertex pair falls into both shared areas Ω_1 and Ω_2. We adopt sampling-based methods to estimate $Pr(Max(z_1, ..., z_k) \leq \delta))$. For example, we have two sample sets SR_1 and SR_2 from two sets R_1 and R_2, respectively. If there are M join pairs (u_1, u_2) such that $Max(|u_1.I_i - u_2.I_i|) \leq \delta, (1 \leq i \leq k)$, $Pr(Max(z_1, ..., z_k) \leq \delta) = \frac{M}{|SR_1| \times |SR_2|}$. The specific technique for computing the optimal sampling technique in high-dimensional space is beyond the scope of this paper. Without loss of generality, we choose random samples, i.e., each point has equal probability of being chosen as a sample.

7.6 Join Order Selection

The join order selection can be performed by adopting the traditional dynamic programming algorithm [32] using the cost model introduced in the previous section. However, this solution is inefficient due to the very large solution space, especially when $|E(Q)|$ is large. Therefore, we propose a simple yet efficient greedy solution to find a good join order. As mentioned earlier, one join order corresponds to one traversal order in query graph Q, according to which, one edge (of Q) is visited in each step. In each step, the subgraph Q' that is formed by collecting all edges that have been visited is called a *status* (Definition 7). Obviously, status Q' moves from NULL to Q according to the specified join order. There are two important heuristic rules in our join order selection.

1) Given a status Q'_i, if there is a backward edge e attached to Q'_i, the next status is $Q'_{i+1} = Q'_i \cup e$, i.e., we perform back edge processing as early as possible. If there are more than one backward edges attached to Q'_i, we perform all back edge processing simultaneously, which will reduce the I/O cost.

The intuition behind this heuristic rule is similar to "selection push-down" in relational query optimization. Performing back edge query will reduce the cardinality of intermediate join results.

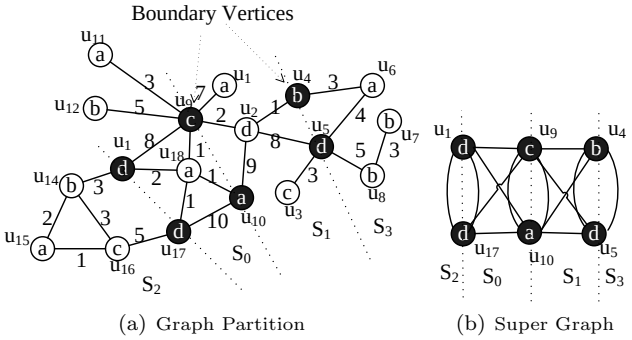
2) Given a status Q'_i, if there is no backward edge attached to Q'_i, the next status is $Q'_{i+1} = Q'_i \cup e$, where e is a forward edge and $Cost(e)$ (defined in Equation 7) is minimum of all forward edges.

8 Bi-Level Distance Join

In this section, we will address the scalability of our method by graph partitioning. Given a graph G, we first partition G into several connected subgraphs $\{S_i\}$ by node separators, which are called *boundary nodes*. We use the same partition strategy given in Lines 1-4 of Algorithm 1, and select all 2-hop centers in $W_b \cup W_s$ as boundary nodes. Then, for each subgraph S_i, we perform graph embedding according to the method in Section 5.1. Consequently, each subgraph S_i is converted into a high dimensional space $\Re_i^k$.

At runtime, given a query Q, we can first employ Algorithm 3 to find all matches in each subgraph S_i. Obviously, some matches may cross several subgraphs. In order to find such matches, we build a *super graph* that contains all boundary nodes. The details are given in Section 8.1.

Figure 12(a) shows an example of graph partition G, where the letters inside vertices are vertex labels and the letters beside the vertices are vertex IDs that we introduce to simplify description of the graph and the black vertices are boundary vertices, and the numbers beside the edges are edge weights.



(a) Graph Partition (b) Super Graph

Fig. 12 Bi-Level Method

8.1 Offline Processing

Algorithm 6 depicts the whole offline processing framework. First, we partition G into some subgraphs $\{S_i\}$ (Line 1 in Algorithm 6). Then, for each subgraph S_i, we employ graph embedding to convert S_i into a multi-dimensional space $\Re_i^k$ (Line 3). According to the method in Algorithm 1, we can compute 2-hop labeling for each subgraph S_i (Line 4). In order to address the matches that cross several subgraphs, we build a super graph G^s, that will be discussed shortly. We also employ graph embedding to convert G^s into the converted space $\Re_{super}^k$, and compute the 2-hop labeling for G^s (Lines 6-7).

We employ the method in [4] to construct a *super graph* G^s. Specifically, a super graph consists of all boundary nodes as vertices and two vertices are connected by an edge directly when they are in the same subgraph S_i. The edge in a super graph is called a *super edge* whose weight is the shortest path distance between two vertices in the subgraph involved. Given the graph partition in Figure 12(a), Figure 12(b) shows the corresponding super graph G^s, which contains all boundary nodes. In each subgraph S_i, we introduce super edges between all pairwise boundary nodes, where the edge weight denotes the corresponding pairwise shortest path distance in S_i. For example, since vertices u_9 and u_{10} are boundary vertices in segment S_0, we introduce an edge $e_1 = (u_9, u_{10})$ in G^s, whose weight is the shortest path distance between u_9 and u_{10} in subgraph S_0. Note that, u_9 and u_{10} are also boundary nodes in subgraph S_2, thus, we introduce another edge $e_2 = (u_9, u_{10})$ in G^s, whose weight denotes the shortest path distance between u_9 and u_{10} in subgraph S_2. Therefore, there are two edges with different weights between u_9 and u_{10} in G^s, meaning that G^s is a multigraph. Actually, if there is more than one edge between two vertices, we can keep only the edge (u_9, u_{10}) with the minimal weight and remove others by postprocessing. Therefore, in the following discussion, we can still assume that G^s is a graph, not a multigraph. Then, we also utilize graph embedding technique to convert G^s into a multi-dimensional space $\Re_{super}^k$.

In summary, at the end of offline processing, we have the following pre-computed results: 1) For each subgraph S_i, there is a table T_i that stores all multi-dimensional points in the converted space $\Re_i^k$; 2) There is a table T_{super} to store all multi-dimensional points that in the converted space $\Re_{super}^k$; 3) For each subgraph S_i and the super graph G^s, there is also a table to store 2-hop labels for S_i and G^s.

Algorithm 6 Offline Processing in Bi-Level Method

Input: Input:Graph G.
Output: $T_i(T_{super})$: the converted multi-dimensional space for each subgraph S_i and G^s; $2HopLabel(S_i)$ and $2HopLabel(G^s)$: the 2-hop labeling for each subgraph S_i and G^s.
1: Partition G into m subgraphs $\{S_1, ..., S_m\}$.
2: **for** each subgraph S_i **do**
3: Employ graph embedding to map S_i into the converted space T_i.
4: Employ Algorithm 1 to compute 2-hop labeling for S_i, denoted as $2HopLabel(S_i)$.
5: Build the super graph G^s.
6: Employ graph embedding to map G^s into the converted space T_{super}.
7: Employ Algorithm 1 to compute 2-hop labeling for G_s, denoted as $2HopLabel(G^s)$.

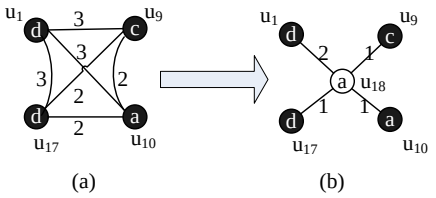


Fig. 13 Optimization For Super Graph Construction

8.2 Optimization for Super Graph Construction

The key problem in the above method is that the super graph G^s is very dense. Although $|V(G^s)|$ is small compared with $|V(G)|$, $|E(G^s)|$ is quite large. As discussed in Section 5.1, we need to employ Dijkstra's algorithm to perform graph embedding. If $|E(G^s)|$ is too large to be cached in memory, it will affect graph embedding performance dramatically, even though we can employ DiskSP algorithm [4] for shortest path distance computation. Actually, most super edges in G^s can be removed, since there are lots of redundancy in G^s. Figure 13 shows a fragment of super graph for partition S_0. There are 4 boundary nodes in S_0, thus, we introduce $C_4^2 = 6$ super edges to connect each pairwise boundary nodes, as shown in Figure 13a. Since vertex u_{18} covers all super edges (i.e., the pairwise shortest paths between boundary nodes), we can introduce u_{18} into the super graph and connect u_{18} with each boundary vertex. The number of super edges is 4 in Figure 13b rather than 6 in Figure 13a.

Motivated by the above example, we propose Algorithm 7 to construct G^s. First, we set $V(G^s) = Boundary(G)$, which means that G^s contains all boundary vertices in G. Then, we consider each subgraph S_i separately. Given a subgraph S_i, the boundary vertices in S_i are denoted as $Boundary(S_i)$. We pre-compute all pairwise shortest paths between vertices in $Boundary(S_i)$ in subgraph S_i, denoted as $D_{Boundary(S_i)}$. Then, we se-

lect a vertex u (in S_i) to maximize $|D_{Boundary(S_i,u)}|$, where $D_{Boundary(S_i,u)}$ denotes all paths in $D_{Boundary(S_i)}$ that pass through u. We introduce edges between u and each path endpoint in $D_{Boundary(S_i,u)}$. We remove $D_{Boundary(S_i,u)}$ from $D_{Boundary(S_i)}$. The above process is iterated until $|D_{Boundary(S_i)}| < 2 \times |D_{Boundary(S_i,u)}|$. The intuition behind the stop condition is as follows. Assume that one vertex u is selected to maximize $|D_{Boundary(S_i,u)}|$ in some iteration step. We need to introduce extra $2\times |D_{Boundary(S_i,u)}|$ edges to connect u and path endpoints $D_{Boundary(S_i,u)}$ in G^s. If $|D_{Boundary(S_i)}| < 2\times|D_{Boundary(S_i,u)}|$, it means that the number of the uncovered pairwise shortest paths is less than the number of introduced extra edges. Thus, we can introduce an edge between two path endpoints in $D_{Boundary(S_i)}$ directly, and the edge weight is the corresponding shortest path distance. Finally, if there are multi-edges between two vertices in G^s, we only keep one edge with the minimal weight.

Algorithm 7 Optimization For Subgraph Graph Construction

Input: Input: $Boundary(G)$: the boundary vertices in G; S_i: each subgraph in G.
Output: The subgraph G'.
1: set $V(G') = Boundary(G)$.
2: **for** each subgraph S_i **do**
3: Set $D_{Boundary(S_i)} = \{\overline{u_1u_2} | u_1, u_2 \in Boundary(S_i)\}$, where $\overline{u_1u_2}$ denotes the shortest path between u_1 and u_2.
4: **repeat**
5: Select a vertex u in S_i to maximize $|D_{Boundary(S_i,u)}|$, where $D_{Boundary(S_i,u)} = \{\overline{u_1u_2} | (u_1, u_2 \in Boundary(S_i)) \wedge (u \in \overline{u_1u_2})\}$.
6: Introduce edges between u and endpoints in $D_{Boundary(S_i,u)}$.
7: $D_{Boundary(S_i)} = D_{Boundary(S_i)} - D_{Boundary(S_i,u)}$.
8: Introduce an edge between u_1 and u_2, where $\overline{u_1u_2} \in D_{Boundary(S_i)}$.
9: **until** $|D_{Boundary(S_i)}| < 2 \times |D_{Boundary(S_i,u)}|$
10: **for** each vertex pair (u_1, u_2), where $u_1, u_2 \in Boundary(G)$ **do**
11: **if** there are multi edges between u_1 and u_2 **then**
12: Only keep the edge with the minimal weight.

8.3 Online Processing

At run time, given a query graph Q, we also employ the framework described in Section 3 to find matches of Q, i.e., answering Q by a series of edge queries. Consequently, the key problem in answering Q is how to answer the edge query efficiently. To distinguish from the method in Section 7, we call the following algorithm *bi-level distance join (bD-join for short)* algorithm.

Algorithm 9 shows a nested-loop join algorithm to answer edge query. In each loop, we join vertices from

Algorithm 8 bD-Join Algorithm

Input: Input: an edge query $e = (v_0, v_1)$ and a parameter δ and two subgraph S_1 and S_2.
Output: Answer set $RS(S_1, S_2)$.
1: According to vertex labels, we find two vertex lists R_1 (from S_1) and R_2 (from S_2) that corresponds to v_1 and v_2, respectively.
2: Employ Algorithm 3 to find $RS_{1\alpha_1} = R_1 \underset{\text{Dist}_{\text{sp}}(u_1,u_2)\leq\delta}{\bowtie} Boundary(P_1)$, where $Boundary(P_1)$ denotes all boundary vertices in P_1.
3: Employ Algorithm 3 to find $RS_{\alpha_2 2} = Boundary(P_2) \underset{\text{Dist}_{\text{sp}}(u_1,u_2)\leq\delta}{\bowtie} R_2$.
4: Employ Algorithm 3 to find $RS_{\alpha_1\alpha_2} = Boundary(S_1) \underset{\text{Dist}_{\text{sp}}(u_1,u_2)\leq\delta}{\bowtie} Boundary(S_2)$.
5: $RS(S_1, S_2) = \sigma_{(RS_{1\alpha_1}.dist+RS_{\alpha_1\alpha_2}.dist+RS_{\alpha_2 2}.dist\leq\delta)}(RS_{1\alpha_1} \bowtie RS_{\alpha_1\alpha_2} \bowtie RS_{\alpha_2 2})$.
6: Return $RS(S_1, S_2)$.

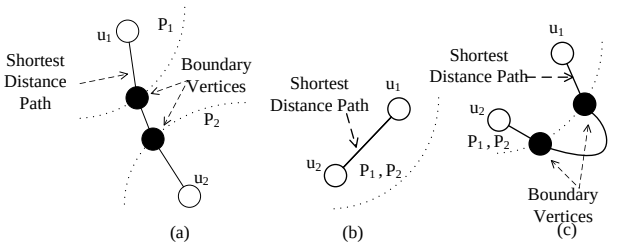


Fig. 14 Shortest Distance Path

S_1 and S_2. Specifically, according to vertex labels, we can find two vertex lists R_1 (from S_1) and R_2 (from S_2) that correspond to v_1 and v_2, respectively. There are two cases namely, $S_1 \neq S_2$ and $S_1 = S_2$.

1) $S_1 \neq S_2$: We first propose how to join R_1 (from S_1) and R_2 (from S_2) in Algorithm 8, where $S_1 \neq S_2$. Considering two vertices u_1 and u_2 in S_1 and S_2, respectively, the shortest path between u_1 and u_2 must go through a boundary vertex in $Boundary(S_1)$ and a boundary vertex in $Boundary(S_2)$. Based on distance constraint δ, we can employ Algorithm 3 to join R_1 and $Boundary(S_1)$ to obtain $RS_{1\alpha_1} = R_1 \bowtie_{Dist_{sp}(u_1,u_2)\leq\delta} Boundary(S_1)$ (Line 2 in Algorithm 8). We also join $Boundary(S_2)$ and R_2 to obtain $RS_{\alpha_2 2}$, where $RS_{\alpha_2 2} = Boundary(S_2) \bowtie_{Dist_{sp}(u_1,u_2)\leq\delta} R_2$ (Line 3). In the super graph G^s, we also employ Algorithm 3 to join $Boundary(S_1)$ and $Boundary(S_2)$, i.e., $RS_{\alpha_1\alpha_2} = Boundary(S_1) \bowtie_{Dist_{sp}(u_1,u_2)\leq\delta} Boundary(S_2)$ (Line 3). The three temporary table formats ($RS_{1\alpha_1}$, $RS_{\alpha_1\alpha_2}$ and $RS_{\alpha_2 2}$) are shown in Figure 15. Finally, we perform natural joins between three tables to obtain final matches for edge query $e = (v_1, v_2)$ by the following SQL query (Line 5).

Select Distinct($RS_{1\alpha_1}.u_1$, $RS_{\alpha_2 2}.u_2$)
From $RS_{1\alpha_1}$,$RS_{\alpha_2 2}$, $RS_{2\alpha_2}$
Where $RS_{1\alpha_1}.\alpha_1 = RS_{\alpha_1\alpha_2}.\alpha_1$
AND $RS_{\alpha_1\alpha_2}.\alpha_2 = RS_{2\alpha_2}.\alpha_2$ *AND* ($RS_{1\alpha_1}.dist$+ $RS_{\alpha_1\alpha_2}.dist + RS_{\alpha_2 2}.dist <= \delta$)

2) $S_1 = S_2$: Next, we discuss how to join two vertex lists R_1 and R_2 that are from the same subgraph. The shortest path between two vertices u_1 and u_2 in the same subgraph may be contained in the subgraph (see Figure 14(b)), or the path also goes through some boundary vertices (see Figure 14(c)). We need to consider both cases. In the former case, we can employ Algorithm 3 to find matches for edge query e. In the latter one, we adopt the same method in Algorithm 8. Specifically, we first employ Algorithm 3 to obtain $RS_{1\alpha_1} = R_1 \bowtie_{Dist_sp(u_1,u_2)\leq\delta} Boundary(S_1)$ and $RS_{\alpha_2 2} = Boundary(S_2) \bowtie_{Dist_sp(u_1,u_2)\leq\delta} R_2$. Then, we also employ Algorithm 3 to join $Boundary(S_1)$ and $Boundary(S_2)$ in G^s. Finally, we perform natural joins between $RS_{1\alpha_1}$ and $RS_{\alpha_2 2}$ and $RS_{\alpha_1\alpha_2}$ to obtain final matches. Note that S_1 is the same as S_2 in this case.

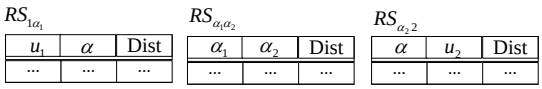


Fig. 15 Three temporary table formats

Algorithm 9 Bi-Level Edge Query

Input: **Input**: an edge query $e = (v_0, v_1)$ and a parameter δ.
Output: Answer set $RS(e)$.
1: **for** each subgraph S_1 **do**
2: **for** each subgraph S_2 **do**
3: According to vertex labels, we find two vertex lists R_1 (from S_1) and R_2 (from S_2) that corresponds to v_1 and v_2, respectively.
4: **if** $S_1 \neq S_2$ **then**
5: Employ Algorithm 8 to find $RS_{(S_1S_2)}$.
6: Insert $RS_{(S_1S_2)}$ into RS.
7: **else**
8: Employ Algorithm 3 to find $RS_{(S_1S_2)} = S_1 \bowtie_{\mathrm{Dist_{sp}}(u_1,u_2)\leq\delta} S_2$ in subgraph S_1.
9: Insert $RS_{(S_1S_2)}$ into RS.
10: Employ Algorithm 8 to find $RS_{S_1S_2}$.
11: Insert $RS_{(S_1S_2)}$ into RS.
12: Report RS.

In order to answer a graph query Q, we can perform a series of edge queries, which is exactly the same as the proposed method in Section 7.2.

9 Approximate Subgraph Search

Given a query graph Q and a data graph G, *subgraph search* locates all subgraph isomorphisms of Q over G. We can simply set distance constraint $\delta = 1$ and employ distance-join algorithm to find all (distance-based) matches of Q over G. Then, for each match, we check its subgraph isomorphism to Q. We omit the details, which are straightforward. Furthermore, distance-join algorithm can also support "approximate subgraph search". In this section, we discuss how to use distance-based join algorithm to answer approximate subgraph query.

Definition 8 (Approximate Subgraph Match). Consider a data graph G, a connected query graph Q that has n vertices $\{v_1, ..., v_n\}$ and a parameter Δ. A *connected* subgraph S that has n distinct vertices $\{u_1, ..., u_n\}$ in G is said to an *approximate subgraph match* of Q, if and only if the following conditions hold:
1) $\forall i, L(u_i) = L(v_i)$, where $L(u_i)$ $(L(v_i))$ denotes u_i's (v_i's) label; and
2)$|\{(v_i, v_j)|(v_i, v_j) \in E(Q) \wedge (u_i, u_j) \notin E(S)\}| \leq \Delta$
3)$|\{(v_i, v_j)|(v_i, v_j) \notin E(Q) \wedge (u_i, u_j) \in E(S)\}| = 0$.

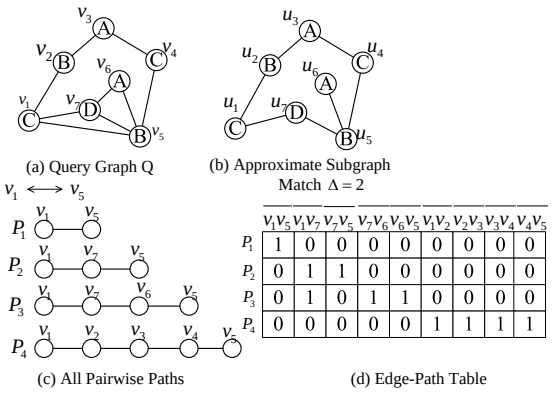


Fig. 16 Approximate Subgraph Search

Definition 9 (Approximate Subgraph Query). Consider a data graph G, a connected query graph Q that has n vertices $\{v_1, ..., v_n\}$, and a parameter Δ, *an approximate subgraph query* finds all approximate subgraph matches (Definition 8) of Q over G.

Given a query graph Q in Figure 16(a) and $\Delta = 2$, Figure 16(b) shows an approximate subgraph match S of Q. The rationale of our proposed solution is that: given two adjacent vertices v_{i_1} and v_{i_2} in Q, removing Δ edges in Q may enlarge the distance between v_{i_1} and v_{i_2}. For example, $Dist_{sp}(v_1, v_5) = 1$ in Figure 16(a). If edge (v_1, v_5) is deleted, $Dist_{sp}(v_1, v_5) = 2$, as shown in Figure 16. Therefore, if we set up distance constant $\delta = 2$, S (in Figure 16(b)) is also a (distance-based) match of Q according to Definition 1.

Given a query graph Q and a parameter Δ, for each edge (v_{i_1}, v_{i_2}) in query Q, we first compute an upper bound for the shortest path distance between v_{i_1} and v_{i_2} (denoted as $Dist_{max}(v_{i_1}, v_{i_2})$), if Δ edges are

deleted from Q. Then, for each edge (v_{i_1}, v_{i_2}) in Q, we introduce a distance constraint $\delta_{ij} = Dist_{max}(v_i, v_j)$. Based on distance constraints, we employ Algorithm 5 to find all (distance-based) matches of Q. Finally, for each (distance-based) match, we check whether it is an approximate subgraph match of Q within G.

Given a query Q in Figure 16(a), for edge (v_1, v_5), if $\Delta = 1$, $Dist_{max}(v_1, v_5) = 2$. Analogously, if $\Delta = 2$, $Dist_{max}(v_1, v_5) = 4$. We propose a set-cover based solution to evaluate Δ. Initially, we precompute and rank all pairwise paths between v_1 and v_5. We use $SP_k(v_1, v_5)$ to denote all the k-th shortest distance paths between v_1 and v_5. Let $Dist_{SP_k}(v_1, v_2)$ be the corresponding distance. For example, $SP_1(v_1, v_5) = \{\overline{v_1 v_5}\}$ and $SP_2(v_1, v_5) = \{\overline{v_1 v_7 v_5}\}$. $Dist_{SP_1}(v_1, v_5) = 1$ and $Dist_{SP_2}(v_1, v_5) = 2$. Then, we build an edge-path table, as shown in Figure 16(d). Given Δ, we need to find k, where $SP_{1,k-1}(v_1, v_5) = \{SP_1(v_1, v_5) \cup ... \cup SP_{k-1}(v_1, v_5)\}$ can be covered by Δ edges, but $SP_{1,k}(v_1, v_5) = \{SP_1(v_1, v_5) \cup ... \cup SP_k(v_1, v_5)\}$ cannot be covered by Δ edges. For example, in Figure 16, if $\Delta = 2$, we can find $k = 4$, since all paths in $SP_{1,3}(v_1, v_5)$ can be covered by 2 edges, but $SP_{1,4}(v_1, v_5)$ cannot be covered by 2 edges. Therefore, $Dist_{max}(v_1, v_5) = Dist_{SP_4}(v_1, v_5) = 4$. We can find the value of k by enumerating all possible combinations of Δ edges in Q. Obviously, there are $\binom{|E(Q)|}{\Delta}$ possible combinations. In order to avoid exponential time complexity, we propose a solution based on the greedy set-cover method.

Given a path set SP, we first find an edge that can cover the maximal number of paths in SP. Then, we remove these paths from SP. The process is iterated Δ times. Finally, there are W_{greedy} paths that are covered by Δ edges. Let W_{max} be the maximal number of paths that can be covered by Δ edges.

Theorem 10 *Given a path set SP, let W_{greedy} be the number of paths covered by Δ edges following the greedy solution. If $W_{max} = 1.6 \times W_{greedy} < |SP|$, SP cannot be covered by Δ edges.*

Proof (Sketch) $W_{max} = 1.6 \times W_{greedy}$ [35]. If $W_{max} < |SP|$, it means that Δ edges cannot cover all paths in SP.

Considering an edge (v_1, v_5) in Figure 16, we need to find the minimal value of k', such that $SP_{1,k'}(v_1, v_5)$ cannot be covered by Δ edges according to Theorem 10 (Lines 3-6 in Bound function in Algorithm 10). Then, we set distance constraint $\delta_{v_1 v_5} = Dist_{max}(v_1, v_5) = Dist_{SP_{k'}(v_1, v_5)}$ (Line 2 in Algorithm 10). Let us consider an extreme case. If $\Delta = 3$, all paths between v_1 and v_5 can be covered by Δ edges. According to Definition 8, it is not allowed to disconnect query Q by removing Δ edges. Therefore, we set up $\delta_{v_1 v_5} = Max(v_1, v_5) = 4$, where $Max(v_1, v_5)$ denotes the longest path distance between v_1 and v_5 in Q (Line 8 in Bound function of Algorithm 10). Then, based on distance constraints, we employ Algorithm 5 to find all matches of Q over G (Line 3 in Algorithm 10). Finally, for each match, we check whether it is an approximate subgraph match (Lines 4-7).

Algorithm 10 Approximate Subgraph Query

Input: : Query graph Q and Data graph G and a parameter Δ.
Output: All approximate subgraph matches.
1: **for** each edge $e_i = (v_{i_1}, v_{i_2})$ **do**
2: $\delta_i = Dist_{max}(v_{i_1}, v_{i_2})$=$Bound(e_i, \Delta)$.
3: Call Algorithm 5 to find all (distance-based) matches in G.
4: **for** each (distance-based) match M **do**
5: **if** M is approximate subgraph isomorphism to Q **then**
6: insert M into answer set RS
7: Report RS

Bound$(e_i = (v_{i_1}, v_{i_2}), \Delta)$
1: Compute and rank all paths between v_{i_1} and v_{i_2} in Q.
2: Let $SP_{k'}(v_{i_1}, v_{i_2})$ denote all the k'-th pairwise paths between v_{i_1} and v_{i_2}.
3: **if** all paths cannot be covered by Δ edges according to Theorem 10 **then**
4: **for** $k' = 1, ...$ **do**
5: **if** all paths in $SP_1(v_{i_1}, v_{i_2}) \cup ... \cup SP_{k'}(v_{i_1}, v_{i_2})$ cannot be covered by Δ edges according to Theorem 10 **then**
6: Return k'
7: **else**
8: Return $Max(v_{i_1}, v_{i_2})$

10 Experiments

We evaluate our methods using both synthetic and real data sets. All of the methods have been implemented using standard C++. The experiments are conducted on a P4 3.0GHz machine with 1G RAM running Windows XP.

Synthetic Datasets *a) Erdos Renyi Model*: This is a classical random graph model. It defines a random graph as N vertices connected by M edges, chosen randomly from the $N(N-1)/2$ possible edges. We set $N = 100K$ and $M = 500K$. This graph is connected, and it is denoted as "ER data".
b) Scale-Free Model: We use the graph generator *gengraph win* (http://fabien.viger.free.fr/liafa/generation/)[28]. We generate a large graph G with 100K vertices satisfying power-law distribution. Usually, $2 < \gamma < 3$ [21]. Thus, default value of parameter α is set to 2.5 in this work. There are 89198 vertices and 115526 edges in the maximal connected component of G. We can sequentially perform our method in each connected component of G. This dataset is denoted "SF data".

Table 1 Performance of BE Method Over Real Unweighted Datasets

Graph	|V(G)|	|E(G)|	Offline Performance								Online Performance (milliseconds)				
			Index Time (sec)				Labeling Size (MB)								
			BE	SC	GP	TEDI	BE	SC	GP	TEDI	BE	SC	GP	TEDI	Dijkstra
Arxiv	27400	352021	1556	#	3523	65	104	#	211	10.3	2.00	#	2.51	0.53	300
As-Rel	22442	91100	304	#	1203	32	9.66	#	12.4	5.78	0.10	#	0.2	0.02	54
California	5925	15770	35.74	1035	327	21	8.25	6.55	11.8	3.2	0.23	0.21	0.31	0.01	14
Epa	4253	8897	48.7	2050	421	5	5.08	4.32	6.85	1.9	0.12	0.1	0.13	0.01	8
Erdo	6927	11850	33.5	1205	230	4	5.95	3.26	7.56	3.21	0.16	0.12	0.2	0.03	12
Eva	4475	4652	10.625	560	216	3	3.6	3	4.02	0.9	0.17	0.12	0.18	0.02	5
UspowerGrid	4941	6594	23.76	230	319	2	4.21	2.5	3.5	2.89	0.10	0.1	0.12	0.02	7
DBLP (unweighted)	592983	591977	22606	#	42560	2530	588	#	1230	129	2.10	#	3.15	0.9	520

#: SC cannot work when |V(G)|>10K

Table 2 $|R_i|$ in Different Datasets

ER	200	SF	200	Citeseer	388
Yeast	46 - 586	DBLP	197		

In the above two datasets, the edge weights in G satisfy a random distribution between $[1, 1000]$. Vertex labels are randomly assigned between $[1, 500]$.

Real Datasets *c) Citeseer:* We generate co-author network G from citeseer dataset (http://cs1.ist.psu.edu/public/oai/). We generate co-author network G as follows: We treat each author as a vertex u in G and introduce an edge to connect two vertices if and only if there is at least one paper co-authored by the two corresponding authors. We assign vertex labels and edge weights as follows: according to text clustering algorithms, we group all author affiliations into 1000 clusters. For each author, we assign the cluster ID as its vertex label. The weight of edge $e = (u_1, u_2)$ in G is assigned as $\frac{100}{co(u_1,u_2)}$, where $co(u_1, u_2)$ denotes the number of co-authored papers between authors u_1 and u_2. There are 387954 vertices and 1143390 edges in the generated G. There are 273458 vertices and 1021194 edges in the maximal connected component of G.

d) Yeast. This is a protein-to-protein interaction network in budding yeast (http://vlado.fmf.uni-lj.si/pub/networks/data/). Each vertex denotes a protein and an edge denotes the interaction between two corresponding proteins. We delete 'self-loop' edges in the original dataset. There are 13 types of protein clusters in this dataset. Therefore, we assign vertex labels based on the corresponding protein clusters. The edge weights are all set to '1'. There are 2361 vertices and 6646 edges in G. There are 2223 vertices and 6608 edges in the maximal connected component of G.

e) DBLP. In order to test the scalability of our method, we utilize the DBLP dataset [30]. The dataset is generated from the DBLP N-triple dump (www4.wiwiss.fu-berlin.de/bizer/d2rq/benchmarks). It contains all the "inproceeding" records and all "proceedings" records, with all their persons, publishers, series and relations between persons and inproceedings. There are about 592983 vertices and 591977 edges in graph G. There are 353779 vertices and 427092 edges in the maximal connected component of G. Since the dataset has no vertex labels and edge weights, we perform the following method to assign them. According to text clustering algorithms, we group all vertices into 3000 clusters. We assign the same vertex label to all vertices in a given cluster. The edge weights in G satisfy a random distribution between $[1, 1000]$.

Query Graphs We randomly generate connected query graphs, which are tree-like structures. It means that there are a few backward edges but many forward edges. As discussed in Section 7.2, forward edge processing is performed by an edge join algorithm (Algorithm 3), but backward edge processing is a selection operation, which is much cheaper than forward edge processing. Given $|E(Q)|$ edges, it is easy to know that there are fewer forward edges in clique-like graphs than that in tree-like graphs. Thus, for our approach, it is much cheaper to answer clique-like graph queries than tree-like graphs. Therefore, in order to test the performance of our approach when query graphs have more forward edges, we prefer using "tree-like" queries. In our experiments, we vary $|E(Q)|$ from 1 to 20, as shown in Figure 25.

As discussed earlier, $|R_i|$ denotes the number of vertices that have the same label as v_i in query Q. We report $|R_i|$ for different datasets in Table 2. The value of distance constraints (δ) will be provided in different experiments.

10.1 Evaluating 2-hop Distance Labeling Computation

We first test the method proposed in Section 4 for 2-hop distance-aware labeling computation, i.e., Algo-

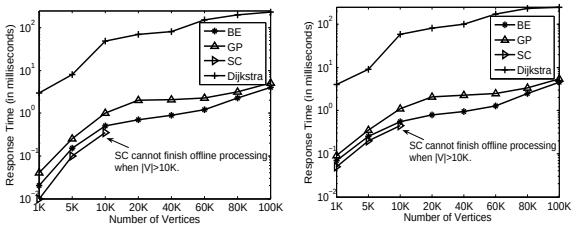
rithm 1, denoted as BE (**b**etweenness **e**stimation-based method). As discussed in Section 2, there exists little work for 2-hop distance-aware labeling computation except for [9] and [7]. We compare BE with these, and denote them as SC (**s**et **c**over-based method) [9] and GP(**g**raph **p**artition-based method)[7]. Since GP can only work on directed graphs, we implement it as follows: We remove the first step in GP, and adopt the graph partition method (the flexible strategy in Section 6.2 of [7]) to compute 2-hop labeling and reduce the 2-hop clusters according to Theorem 1 in Section 5.2 of [7]. Two other methods [30,33] can only work on an un-weighted graph, and do not use the 2-hop labeling strategy, therefore, we do not use them in the performance comparisons.

Figure 17(a) and Figure 17(b) show the running time of SC, GP and BE on synthetic datasets (SF and ER models) varying $|V(G)|$ from 1K to 100K. When $|V(G)|$ >10K, SC runs out of memory. Our method (BE) is significantly faster than GP, as shown in Figures 17(a) and 17(b). The reasons are: 1) the number of selected 2-hop centers in our method is small than that in GP; 2) We employ a light but effective redundancy checking method. The maximum memory usage in BE method is about 200M bytes when $|V(G)| = 100K$, which confirms the good scalability of BE method.

Figure 17(c) and Figure 17(d) show the labeling size of different methods on synthetic datasets (SF and ER modes) varying $|V(G)|$ from 1K to 100K. Although SC has the minimal labeling size, it cannot work when $|V(G)|$ >10K. Since our method has smaller 2-hop centers than that in GP, the labeling size in our method is also smaller than that in GP.

Comparing Figure 17(a) and Figure 17(b), and Figure 17(c) and Figure 17(d), we can make some interesting observations: The performance of all three methods over SF data are much better than that over ER data; they have faster running times and smaller labeling sizes. The reason behind that is as follows: The vertex degrees in SF data follow the power-law distribution. A few vertices have very large degrees, and they cover a large fragment of pairwise shortest paths. Thus, these vertices can be selected as the 2-hop centers to minimize space labeling size. However, we cannot find such vertices in ER data, since the vertices in ER have similar vertex degrees. Thus, the number of 2-hop centers in ER data is much larger than that in SF data.

In order to test online response time for the shortest path distance query, we randomly generate 1000 queries. The reported results in Figure 18 are the average online response times for a single query. We implement Dijkstra's algorithm using C++ Boost library (www.boost.org). For shortest path query using 2-hop



(a) SF Data (b) ER Data

Fig. 18 The Online Performances of Shortest Path Query

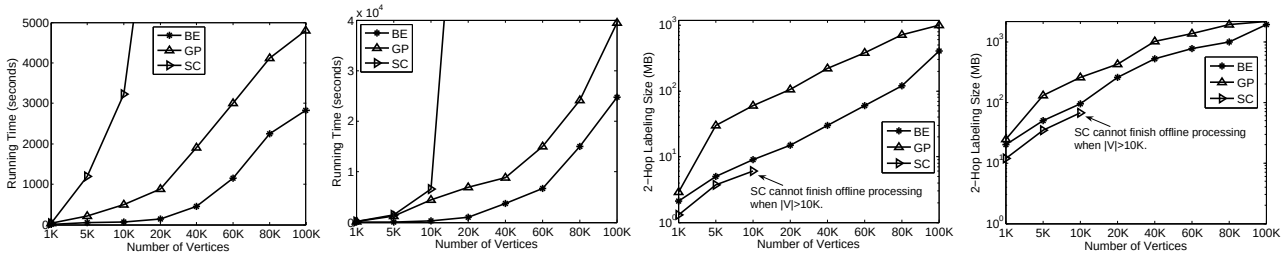
labeling, we assume that all 2-hop labels are retained on disk. This means that the online response time includes I/O cost (loading the corresponding 2-hop labels into memory) and CPU cost (evaluating Equation 5). Figure 18 shows that BE, SC and GP have similar online performance, since they are all 2-hop labeling. The average speed up by 2-hop labeling is 50∼100 over Dijkstra's algorithm using SF and ER data.

In order to further study the performance of our method GP, we compare it with TEDI [30], the existing state-of-the-art technique. Since TEDI can only support unweighted graphs, we only evaluate them in real unweighted graphs that are provided by authors of [30] together with the codes of TEDI in Table 1. Although GP has much better performance than other 2-hop labeling techniques, it is not as good as TEDI. TEDI uses "tree-decomposition" to reduce the search space, thus, it is faster than our 2-hop labeling method. In the unweighted graphs, TEDI uses breath-first search (BFS) to find the shortest paths. However, BFS cannot be used to compute shortest paths in weighted graphs. Therefore, TEDI can only support unweighted graphs, but GP is a general solution for shortest path distance queries. Furthermore, our solution focuses on weighted graphs, thus, GP algorithm is still preferable for distance evaluation in Lines 11-14 in Algorithm 3. In fact, if the graphs are unweighted graphs, we can use TEDI in the verification step to speed up the query processing.

10.2 Evaluating Single-Level Method

10.2.1 Offline Processing

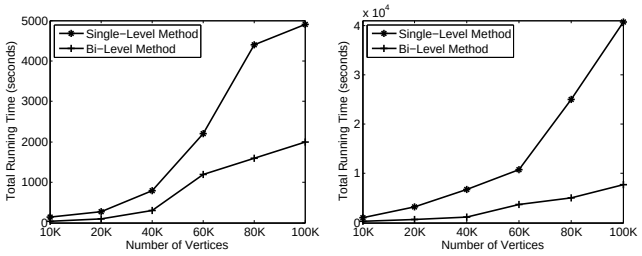
Figure 19 reports the total offline processing time on SF and ER data varying the vertex number from 10K to 100K. Figure 20 shows the total index sizes, including the converted multi-dimensional space, the 2-hop labeling and the data structure that is discussed in Section 5.2. In this experiment, we consider D-join algorithm to answer an edge query. For clustering, we use the



(a) Running Time on SF Data (b) Running Time on ER Data (c) Labeling Size on SF Data (d) Labeling Size on ER Data

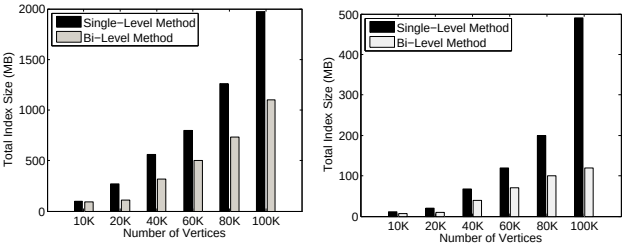
Fig. 17 The Performances of Different 2-Hop Labeling Computing Methods

k-medoids algorithm. The value of the cluster number depends on the available memory size for join processing.



(a) ER Network (b) SF Network

Fig. 19 Total Offline Processing Time



(a) ER Network (b) SF Network

Fig. 20 Total Index Size

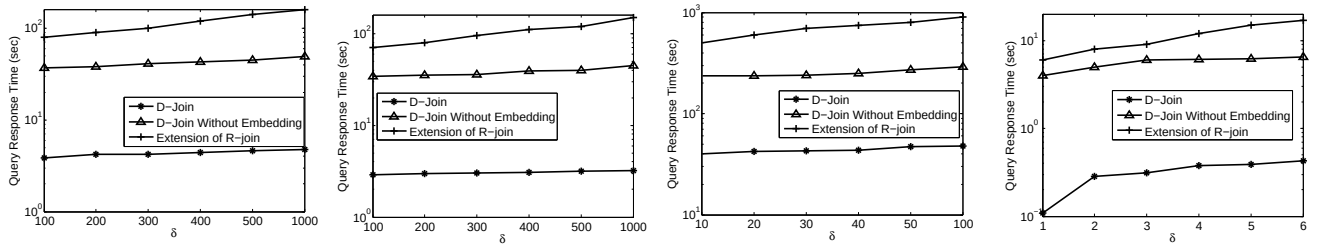
10.2.2 Online Processing

We first evaluate the effectiveness of LLR embedding technique. We choose two alternative methods for performance comparison: the extension of R-join algorithm [8] and the D-join without embedding. In D-join without embedding method, we conduct distance-based joins directly over the graph, rather than first performing join processing over converted space and verifying candidate matches. We use cluster-based block nested loop join and triangle pruning, but no 'hash join' pruning. We report query response time in Figure 21, which shows that the response time of D-join is lower than 'D-Join without Embedding' by orders of magnitude. This is because of two reasons: first, L_∞ distance computation is faster than shortest path distance computation by 3 orders of magnitude in our tests; second, LLR embedding can filter out about 90% of search space. Finally, the extension of R-join cannot work well for our problem, since no pruning techniques are introduced to reduce the search space.

Then, we evaluate the effectiveness of the proposed pruning techniques for the D-join algorithm. We report the number of distance computations (after pruning) and query response time in Figures 22 and 23, respectively. In order to evaluate the pruning power of different pruning strategies, we do *not* utilize neighbor area pruning that shrinks the two lists before join processing. Neighbor area pruning is evaluated in Figure 23.

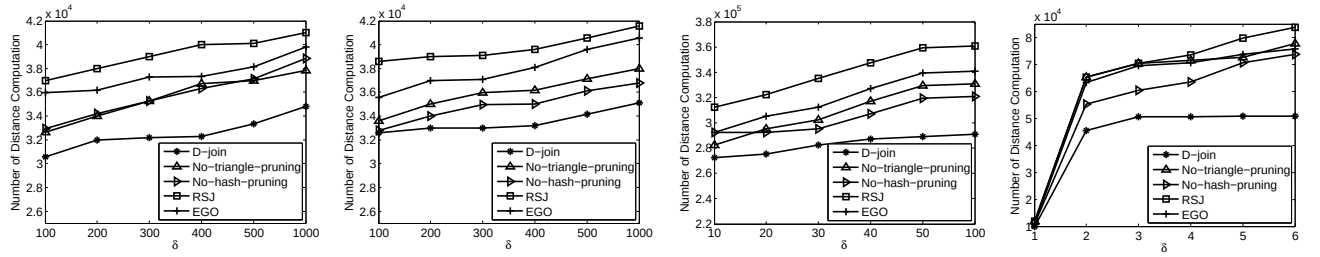
In 'no-triangle-pruning' (see Figure 23) method, we only use the hash join technique. Consequently, in 'no-hash pruning', we only use triangle inequality pruning. We also compare our techniques with two alterative similarity join algorithms: RSJ [3] and EGO [1]. Figure 22 shows that using two pruning techniques (triangle pruning and hash join) together can provide better pruning power, since they are orthogonal to each other. Furthermore, since the dimensionality of the converted vector space is large, R-tree based RSJ cannot work well due to the dimensionality curse. As shown in Figures 22 and 23, D-join with both pruning methods outperforms EGO significantly, because EGO algorithm is not optimized for L_∞ distance. Note that, the difference between the running time in D-join and EGO is not clear in Figure 23(d), since Yeast dataset has only about 2000 vertices.

We test the two cost estimation techniques. Estimation error is defined as $\frac{||CL'|-|CL'||}{|CL|}$, where $|CL|$ is the actual candidate size and $|CL'|$ is estimation size. Since there are some correlations in $\Re^k$ space, dimension independence assumption does not hold. Sampling-based



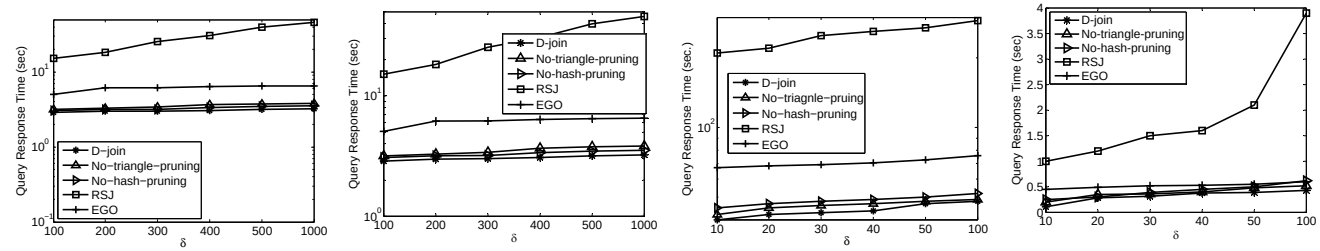
(a) ER Network (b) SF Network (c) Citeseer Network (d) Yeast Network

Fig. 21 Evaluating Embedding Technique



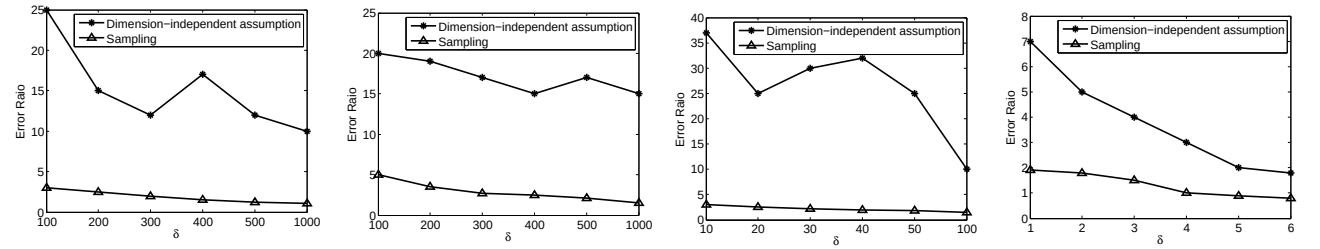
(a) ER Network (b) SF Network (c) Citeseer Network (d) Yeast Network

Fig. 22 Number of Distance Computation



(a) ER Network (b) SF Network (c) Citeseer Network (d) Yeast Network

Fig. 23 Edge Query Response Time

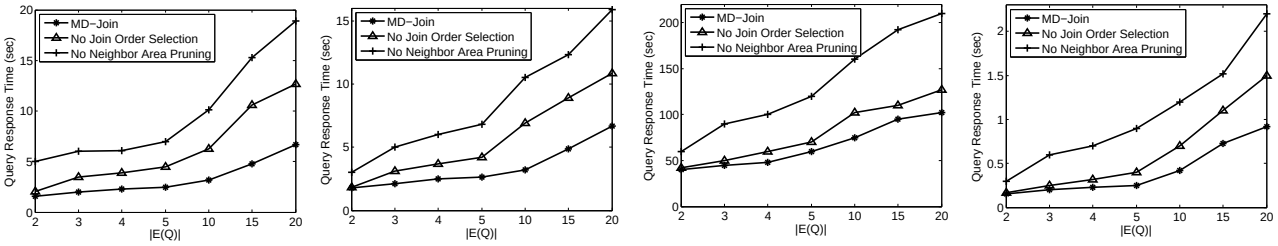


(a) ER Network (b) SF Network (c) Citeseer Network (d) Yeast Network

Fig. 24 Cost Estimation

technique can capture data distribution in $\Re^k$ space, thus, it can provide better estimation, as shown in Figure 24.

We test the performance of MD-join algorithm. We also evaluate the effectiveness of neighbor-area pruning technique and join order selection method. In this experiment, we fix the distance constraint δ, and vary $|E(Q)|$ from 2 to 20. In 'without neighbor area pruning', we do not reduce the search space by neighbor area pruning, but we still use join order selection method to select a cheap query plan. In 'no join order selection', we randomly define the join order for the MD-



(a) ER Network, $\delta = 200$ (b) SF Network, $\delta = 200$ (c) Citeseer Network, $\delta = 20$ (d) Yeast Network, $\delta = 2$

Fig. 25 Pattern Match Query Response Time VS. $|E(Q)|$

join processing, but we use neighbor area pruning. We use both techniques in MD-join algorithm. Without neighbor area pruning, the search space is much larger than in MD-join algorithm using neighbor area pruning, which is confirmed by the experimental results shown in Figure 25. 'No join order selection' is much slower than MD-join algorithm. Figure 25 also demonstrates that randomly defining join order cannot work as well as MD-join algorithm.

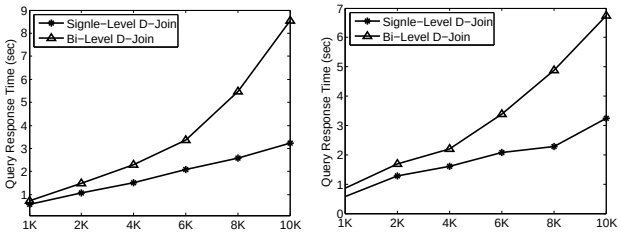
10.3 Evaluating Bi-Level Method

When $|V(G)|$ is very large, the offline processing is very expensive, especially in the graph embedding process (recall Figures 19(a) and 19(b)). We evaluate bi-level offline processing time on SF and ER data varying $|V(G)|$ from 10K to 100K. During the implementation, we partition G into 10 subgraphs. Figures 19(a) and 19(b) show that bi-level offline processing is much faster than the single-level method. Furthermore, the index size in the second-level method is also smaller than that in the single-level method. Actually, the single-level method cannot finish offline processing in 48 hours on DBLP data. However, the bi-level method only spends 5 hours on the same datasets.

However, the online performance of bi-level method is not as good as the single-level version, as shown in Figure 26. We only test bD-Join algorithm (Algorithm 9) over DBLP data in Figure 27, since the single-level method cannot finish offline processing on such large data. Figure 28(a) shows the query response time for varying δ from 200 to 1000. We also fix $\delta = 200$ and vary query graph size from 2 to 6 on DBLP data in Figure 28(b).

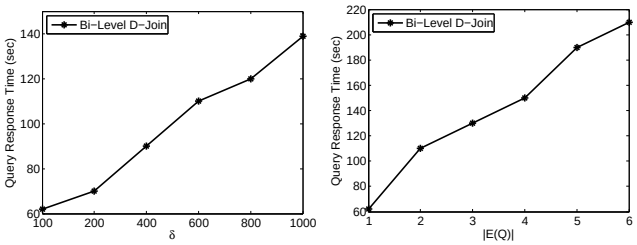
10.4 Evaluating Approximate Subgraph Search

In this section, we evaluate Algorithm 10 for approximate subgraph search. We fix $|E(Q)| = 20$ and vary relax ratio Δ from 1 to 5 in Figure 28, which shows



(a) ER Network (b) SF Network

Fig. 26 Edge Query Response Time, $\delta = 200$



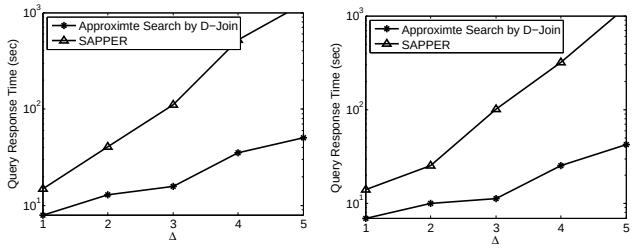
(a) Edge Query (b) Pattern Query

Fig. 27 Query Response Time Over DBLP data

that our method is faster than SAPPER by orders of magnitude, especially when $\Delta > 3$. The key reason is that SAPPER always transforms an approximate subgraph query allowing for missed edges to all possible exact subgraph queries. Obviously, the number of possible exact subgraph queries is exponential with respect to Δ. Thus, SAPPER cannot work well when Δ is large.

11 Conclusions

In this paper, we propose a novel pattern matching problem over a large graph G by graph embedding technique. In order to address the complexity of offline processing, we design the bi-level version (i.e., bD-Join) of D-Join algorithm. We also propose a solution to answer approximate subgraph queries.



(a) ER Graphs (b) SF Graphs

Fig. 28 Approximate Subgraph Queries

Acknowledgement

Lei Zou and Dongyan Zhao were supported by NSFC under Grant No.61003009 and RFDP under Grant No. 20100001120029. Lei Chen's work was supported by RGC NSFC Joint Project Under Grant No. N_HK UST 61_2/09,NSFC under Grant No. 60970112, 60003047, and NSF of Zhejiang Province under Grant No. Z11008 22. M. Tamer Özsu's work was supported by NSERC of Canada.

References

1. C. Böhm, B. Braunmüller, F. Krebs, and H.-P. Kriegel. Epsilon grid order: An algorithm for the similarity join on massive high-dimensional data. In *SIGMOD*, pages 379–388, 2001.
2. U. Brandes and C. Pich. Centrality estimation in large networks. *I. J. Bifurcation and Chaos*, 17(7):2303–2318, 2007.
3. T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Efficient processing of spatial joins using r-trees. In *SIGMOD*, pages 237–246, 1993.
4. E. P. F. Chan and N. Zhang. Finding shortest paths in large network systems. In *ACM-GIS*, pages 160–166, 2001.
5. Y. Chen and Y. Chen. An efficient algorithm for answering graph reachability queries. In *ICDE*, pages 893–902, 2008.
6. J. Cheng, Y. Ke, W. Ng, and A. Lu. fg-index: Towards verification-free query processing on graph databases. In *SIGMOD*, pages 857–872, 2007.
7. J. Cheng and J. X. Yu. On-line exact shortest distance query processing. In *EDBT*, pages 481–492, 2009.
8. J. Cheng, J. X. Yu, B. Ding, P. S. Yu, and H. Wang. Fast graph pattern matching. In *ICDE*, 2008.
9. E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. *SIAM J. Comput.*, 32(5):937–946, 2003.
10. L. Freeman. A set of measures of centrality based upon betweenness. *Sociometry*, 40:35–41, 1977.
11. G. Gou and R. Chirkova. Efficient algorithms for exact ranked twig-pattern matching over graphs. In *SIGMOD*, 2008.
12. J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers, 2000.
13. J. Y. P. S. Y. Hao He, Haixun Wang. Blinks: Ranked keyword searches on graphs. In *SIGMOD*, pages 305–316, 2007.
14. B. Harangsri, J. Shepherd, and A. H. H. Ngu. Selectivity estimation for joins using systematic sampling. In *DEXA Workshop*, pages 384–389, 1997.
15. H. He and A. K. Singh. Closure-tree: An index structure for graph queries. In *ICDE*, pages 38–40, 2006.
16. H. V. Jagadish, B. C. Ooi, K.-L. Tan, C. Yu, and R. Z. 0003. idistance: An adaptive b^+-tree based indexing method for nearest neighbor search. *ACM Trans. Database Syst.*, 30(2):364–397, 2005.
17. H. Jiang, H. Wang, P. S. Yu, and S. Zhou. Gstring: A novel approach for efficient search in graph databases. In *ICDE*, pages 566–575, 2007.
18. N. Jing, Y.-W. Huang, and E. A. Rundensteiner. Hierarchical encoded path views for path query processing: An optimal model and its performance evaluation. *IEEE Trans. Knowl. Data Eng.*, 10(3), 1998.
19. G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, 1998.
20. N. Linial, E. London, and Y. Rabinovich. The geometry of graphs and some of its algorithmic applications. *Combinatorica*, 15(2):215–245, 1995.
21. Réka Albert and Albert-László Barabási. Statistical mechanics of complex networks. *Reviews of Mordern Physics*, 74:47–97, 2002.
22. G. Sabidussi. The centrality index of a graph. *Psychometrika*, 31:581–603, 1966.
23. C. Shahabi, M. R. Kolahdouzan, and M. Sharifzadeh. A road network embedding technique for k-nearest neighbor search in moving object databases. *GeoInformatica*, 7(3):255–273, 2003.
24. D. Shasha, J. T.-L. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In *PODS*, pages 39–52, 2002.
25. Y. Tian, R. C. McEachin, C. Santos, D. J. States, and J. M. Patel. Saga: a subgraph matching tool for biological graphs. *Bioinformatics*, 23(2), 2007.
26. H. Tong, C. Faloutsos, B. Gallagher, and T. Eliassi-Rad. Fast best-effort pattern matching in large attributed graphs. In *SIGKDD*, pages 737–746, 2007.
27. S. Trißl and U. Leser. Fast and practical indexing and querying of very large graphs. In *SIGMOD*, pages 845–856, 2007.
28. F. Viger and M. Latapy. Efficient and simple generation of random simple connected graphs with prescribed degree sequence. In *COCOON*, 2005.
29. H. Wang, H. He, J. Yang, P. S. Yu, and J. X. Yu. Dual labeling: Answering graph reachability queries in constant time. In *Proceedings of International Conference on Data Engineering*, pages 75–89, 2006.
30. F. Wei. Tedi: Efficient shortest path query answering on graphs. In *SIGMOD*, pages 99–110, 2010.
31. D. W. Williams, J. Huan, and W. Wang. Graph database indexing using structured graph decomposition. In *ICDE*, pages 976–985, 2007.
32. Y. Wu, J. M. Patel, and H. V. Jagadish. Structural join order selection for xml query optimization. In *ICDE*, pages 443–454, 2003.
33. Y. Xiao, W. Wu, J. Pei, W. Wang, and Z. He. Efficiently indexing shortest paths by exploiting symmetry in graphs. In *EDBT*, 2009.
34. X. Yan, P. S. Yu, and J. Han. Graph indexing: A frequent structure-based approach. In *SIGMOD*, pages 335–346, 2004.
35. X. Yan, P. S. Yu, and J. Han. Substructure similarity search in graph databases. pages 766–777, 2005.
36. P. Zhao, J. X. Yu, and P. S. Yu. Graph indexing: Tree + delta >= graph. In *VLDB*, pages 938–949, 2007.
37. L. Zou, L. Chen, M. T. Özsu, and D. Zhao. Answering pattern match queries in large graph databases via graph embedding. Technical report, Peking University, 2011.