Distance-Join: Pattern Match Query In a Large Graph Database [*]

Lei Zou
Huazhong University of Science and Technology
Wuhan, China
zoulei@mail.hust.edu.cn

Lei Chen
Hong Kong University of Science and Technology
Hong Kong
leichen@cse.ust.hk

M. Tamer Özsu
University of Waterloo
Waterloo, Canada
tozsu@cs.uwaterloo.ca

ABSTRACT

The growing popularity of graph databases has generated interesting data management problems, such as subgraph search, shortest-path query, reachability verification, and pattern match. Among these, a pattern match query is more flexible compared to a subgraph search and more informative compared to a shortest-path or reachability query. In this paper, we address pattern match problems over a large data graph G. Specifically, given a pattern graph (i.e., query Q), we want to find all matches (in G) that have the similar connections as those in Q. In order to reduce the search space significantly, we first transform the vertices into points in a vector space via graph embedding techniques, coverting a pattern match query into a distance-based multi-way join problem over the converted vector space. We also propose several pruning strategies and a join order selection method to process join processing efficiently. Extensive experiments on both real and synthetic datasets show that our method outperforms existing ones by orders of magnitude.

1. INTRODUCTION

Graphs have been used to model many data types in different domains, 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 subgraph search [19, 26, 27, 5, 10, 8, 28, 13, 20, 21], shortest-path query [7, 3, 16], reachability query [7, 24, 23, 4], and pattern match query [6, 22]. Among these interesting queries, a pattern match query is more flexible than a subgraph search and more informative than a simple shortest-path or reachability query. Specifically, a pattern match looks for the existences of a pattern graph in a data graph. A pattern match query is different from a 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 discuss an effective and efficient method for executing pattern match queries over a large graph database.

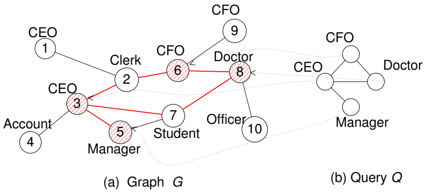
We describe a pattern match query as follows: given a data graph G, a query graph Q (with n vertices), and a parameter δ, n vertices in G can form a *match* to Q, if: (1) these n vertices in G have the same labels as the corresponding vertices in Q; and (2) for any two adjacent vertices v_i and v_j in 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 distance to measure the distance between two vertices, but our approach is not restricted to this distance function, it can be applied to other *metric* distance functions as well. We discuss two examples to demonstrate the usefulness of pattern match queries.

Example 1. Facebook Network Analysis
Figure 1(a) shows a fictitious graph model (G) of Facebook, where vertices represent active users and the edges indicate the friendship relations between two users. There are "job-title" attributes associated with vertices. We treat job-titles as vertex labels. Note that, the numbers inside vertices are vertex IDs that we introduce to simplify description of the graph. A pattern match query, Q (in Figure 1(b)), looks for friendship relations between four types of users, i.e, four types of labels: CFO, CEO, Manager and Doctor, and constraints are set up on the shortest-path distance (≤ 2) between any pair of matched labeled vertices in G. Finding such patterns may help social science researchers discover connections between a successful CEO and his/her circle of friends. In Figure 1(a), vertices (3,5,6,8) match Q, which indicates that vertices (3,5,6,8) (in G) have similar relationships as those specified in query Q.

Example 2. Biological Network Investigation
We can model a biological network as a large graph, such as a protein-protein interaction network (PPI) and a metabolic network, where vertices represent biological entities (proteins, genes and so on) and edges represent the interactions between them. Consider the following scenario: in order to study a certain disease, a scientist has constructed

[*]This work was done when the first author was visiting University of Waterloo as a visiting scholar. The first author was partially supported by National Natural Science Foundation of China under Grant 70771043. The second author was supported by Hong Kong RGC GRF 611608 and NSFC/RGC Joint Research Scheme N HKUST602 /08. The third author was supported by National Science and Engineering Research Council (NSERC) of Canada.

VLDB '09, August 24-28, 2009, Lyon, France



(a) Graph G

(b) Query Q

Figure 1: Pattern Match Query in Facebook Network

a small portion of a biological network Q based on various experimental data. The scientist is interested in predicting more biological activities about the disease. So, s/he wants to find matches of Q in a large biological network G about another well-studied disease. The matches in G have the same (or similar) pathways (i.e. shortest-path) as those in Q.

As shown in these above examples, pattern match queries are useful; however, it is non-trivial to find all matches in a large graph due to the 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, 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 the pattern Q in G is a sequence of *distance-based join* operations, which is very costly for large graphs. For example, assuming that the query Q has 6 vertices, the data graph G has 100,000 vertices, and each query vertex has 100 match vertices in G, the search space is $(100)^6 = 10^{12}$! Therefore, we need efficient pruning strategies to reduce the search space. Although many effective pruning techniques have been proposed for subgraph search, (e.g. [19, 26, 27, 5, 10, 8, 28, 13, 20, 21]), 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. 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. 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 map vertices into vectors via an embedding method and conduct distance-based multi-way join over a vector space.

2) We design an efficient distance-based join algorithm for an edge query in the converted vector space, which well utilizes the block nested loop join and hash join techniques to handle high dimensional vector space.

3) We 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.

4) Finally, we conduct extensive experiments with real and synthetic data to demonstrate the effectiveness of our solutions to answer pattern match queries.

The rest of the paper is organized as follows. We discuss the related work in Section 2. Our framework is presented in Section 3. In Section 4, we propose neighbor area pruning technique. We propose a distance-based join algorithm for an edge query and its cost model in Section 5. Section 6 presents a distance-based multi-way join algorithm for a pattern match query and join order selection method. We study our methods by experiments in Section 7. Section 8 concludes this paper.

2. RELATED WORK

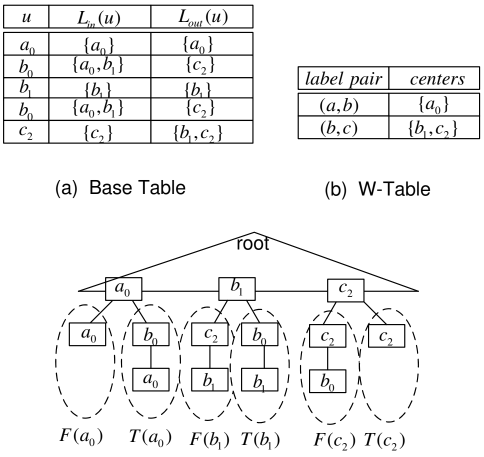
Let $G = \langle V, E \rangle$ to be a graph where V is the set of vertices and E is the set of edges. Given two vertices u_1 and u_2 in G, a *reachability query* verifies if there exists a path from u_1 to u_2, and *distance query* returns the shortest path distance between u_1 and u_2 [7]. These are well-studied problems, with a number of vertex labeling-based solutions [7]. 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: that are 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.

$$\begin{aligned} &Dist_{sp}(u_1, u_2) = \min\{Dist_{sp}(u_1, w) + Dist_{sp}(u_2, w)| \\ &w \in (L_{out}(u_1) \cap L_{in}(u_2))\} \end{aligned} \quad (1)$$

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 $O(|V(G)||E(G)|^{1/2})$ [6]. 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 $O(|E(G)|^{1/2})$.

To the best of our knowledge, there exists little work on pattern match queries over a large data graph, except for [6, 22]. In [6], 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 [6] "reachability pattern match". We first illustrate the method in [6] using Figure 2, and then discuss how it can be extended it to solve our problem and present the shortcomings of the extension.

Without loss of generality, we first assume that there is only one directed edge $e = (v_1, v_2)$ in query Q. Figure 2(a) shows a base table to store all vertex distance labels. For each center w_i, two clusters $F(w_i)$ and $T(w_i)$ of vertices are defined, where for every vertex u_1 in $F(w_i)$, it can reach every vertex u_2 in $T(w_i)$, via w_i. Then, an index structure is built based on these clusters, as shown in Figure 2c. 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 $F(w_i)$ (and $T(w_i)$). Consider a directed



u	$L_{in}(u)$	$L_{out}(u)$
a_0	$\{a_0\}$	$\{a_0\}$
b_0	$\{a_0, b_1\}$	$\{c_2\}$
b_1	$\{b_1\}$	$\{b_1\}$
b_0	$\{a_0, b_1\}$	$\{c_2\}$
c_2	$\{c_2\}$	$\{b_1, c_2\}$

(a) Base Table

label pair	*centers*
(a,b)	$\{a_0\}$
(b,c)	$\{b_1, c_2\}$

(b) W-Table

(c) Cluster-based Index

Figure 2: R-join

edge $e = (v_1, v_2)$ in query Q and assume that the labels of vertex v_1 and v_2 (in query Q) are 'a' and 'b', respectively. According to table W-Table in Figure 2b, we can find centers w_i, in which there exists at least a vertex u_1 labeled 'a' in $F(w_i)$, and there exists at least a vertex u_2 labeled 'b' in $T(w_i)$. For each such center w_i, the Cartesian product of vertices labeled 'a' in $F(w_i)$ and vertices labeled 'b' in $T(w_i)$ can form the matches of Q. This operation is called *R-join* [6]. In this example, there is only one center a_0 that corresponds to vertex label pair (a, b), as shown in Figure 2(b). According to index structure in Figure 2(c), we can find $F(a_0)$ and $T(a_0)$. 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.

We can extend the method in [6] to distance pattern match using 2-hop distance labeling instead of reachability labeling. Again, we first assume that there is only one edge $e = (v_1, v_2)$ in query Q. The vertex labels are 'a' and 'b', respectively. In order to find distance pattern matches, following the framework in [6], we also find all centers w_i, in which there exists at least a vertex u_1 labeled 'a' in $F(w_i)$ and a vertex u_2 labeled 'b' in $T(w_i)$. 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 [6], 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. Since a vertex u may exist in different clusters $F(w_i)$ and $T(w_i)$, the computational cost of this straightforward extension is far larger than $|R_1| \times |R_2|$.

As discussed in Section 1, the challenge in our distance pattern match problem is the huge search space. Simply extending the method proposed in [6] will not resolve the efficiency issue. 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. Furthermore, our method is independent of 2-hop graph labeling techniques.

The best-effect algorithm [22] returns K matches with large scores. Based on some heuristic rules, the algorithm first finds the most promising match vertex u (in data graph G) for one vertex in query Q (called *Seed-Finder*). Then, it extends the vertex to match other vertices in Q (called *Neighbor-Expander*). After that, it finds a "good" path to connect two match data vertices if they are required to be connected according to query Q (called *Bridge*). The query can be repeated with another seed node, until the user receives all k matches that are requested. This 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 the algorithm cannot guarantee the completeness of results. In [9], authors propose ranked twig queries over a large graph, however, a "twig pattern" is a directed graph, not a general graph.

Besides reachability, distance, and pattern match queries, there are a lot of works on subgraph search over graph databases, such as [19, 26, 27, 5, 10, 8, 28, 13, 20, 21], none of which can be applied to pattern match queries, since all these pruning techniques are based on the necessary condition of subgraph isomorphism.

Table 1: Meanings of Symbols Used

G	data graph	Q	Query Graph
$V(G)/V(Q)$	Vertex set of G/Q	v_i	a vertex in Q
$E(G)/E(Q)$	Edge set of G/Q	u_i	a vertex in G

3. FRAMEWORK

In this section, we give the formal definition of pattern match queries over a graph and present the general framework of our proposed solution. As discussed in Section 1, in this work, we study search over a large *vertex-labeled* and *edge-weighted* undirected graph. In the following, unless otherwise specified, all uses of the term "graph" refer to a vertex-labeled and edge-weighted graph. The common symbols used in this paper are given in Table 1.

DEFINITION 3.1. ***Match***. *Consider a data graph G, a connected query graph Q that has n vertices $\{v_1, ..., v_n\}$, and a parameter δ. A set of n distinct vertices $\{u_1, ..., u_n\}$ in G is said to be a* match *of Q, if and only if the following conditions hold:*
1) $L(u_i) = L(v_i)$, where $L(u_i)(L(v_i))$ denotes u_i's (v_i's) label; and
2) If there is an edge between v_i and v_j in Q, the shortest path distance between u_i and u_j in G is no larger than δ, that is, $Dist_{sp}(u_i, u_i) \leq \delta$.

Given an edge (v_i, v_j) in Q and its match (u_i, u_j), the shortest path between u_i and u_j in G is said to be a *match path* of the edge (v_i, v_j) in Q.

DEFINITION 3.2. ***Pattern Match Query***. *Given a large data graph G, a connected query graph Q with n vertices $\{v_1, ..., v_n\}$, and a parameter δ, a* pattern match query *reports all matches of Q in G according to Definition 3.1.*

According to Definition 3.2, 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 query 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. Given a pattern match query Q that has n vertices, according to vertex label predicates associated with each vertex v_i, we first obtain 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. We say list R_i corresponds to a vertex v_i in Q. Then, we need to perform a shortest path distance-based multi-way join over these lists. 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 3 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.

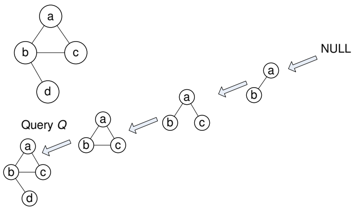


Figure 3: A Join-Order

As in left-deep join processing in relational systems, we always perform a shortest path distance-based two-way join to answer an edge query. We call this two-way join *Distance-Join* (*D-join* for short), which is expressed by Equation 2, in which R_1 and R_2 are two lists of vertices in graph G, and u_1 and u_2 are two vertices in the two lists, respectively.

$$RS = R_1 \underset{\mathrm{Dist}_{\mathrm{sp}}(u_1,u_2)\le\delta}{\bowtie} R_2 \tag{2}$$

According to Definition 3.1, 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 $O(|V(G)||E(G)|^{1/2})$ [15]. Thus, we adopt graph labeling technique instead of *Pre-compute* method to perform shortest-path distance computation.

The key problem in naive join processing is its large number of distance computations, which is $|R_1| \times |R_2|$. In order to speed up the query performance, we need to address two issues: how to reduce the number of distance computations; and, finding a distance computation method to find all candidate matches that is more efficient than shortest path distance computation.

In order to address these issues, we utilize *LLR embedding* technique [17, 18] 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 3.1). 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 termed as *similarity join*) in vector space [1, 2]; we focus on L_∞ distance in the converted space simply because we use L_∞ distance to find candidate matches.

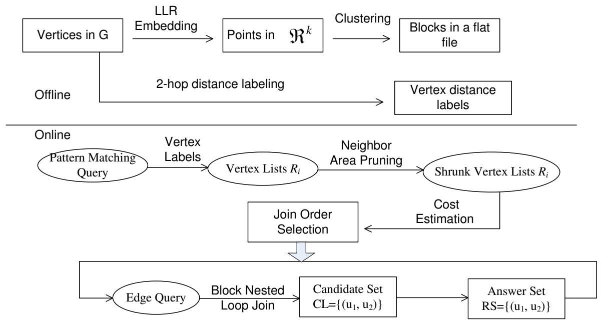


Figure 4: Framework of Pattern Match Query

Figure 4 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 [11] 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 compute graph distance label for each vertex to enable fast shortest path distance computation [7, 15]. 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 5.1); 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.

According to LLR embedding technique [17, 18], 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 random selected vertices in $V(G)$. We define

$$D(u, S_{n,m}) = min_{u' \in S_{n,m}}\{Dist_{sp}(u, u')\} \tag{3}$$

that is, $D(u, S_{n,m})$ is the distance from u to its closest neighbor in $S_{n,m}$.

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 $k = \kappa\beta = O(log^2|V(G)|)$. Each subset $S_{n,m}$ ($1 \le n \le \beta$, $1 \le m \le \kappa$) in R has 2^n vertices in $V(G)$.

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

$$E(u) = [D(u, S_{1,1}), ..., D(u, S_{1,\kappa}), ..., D(u, S_{\beta,1}), ..., D(u, S_{\beta,\kappa})] \tag{4}$$

where $\beta\kappa = k$.

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

$$L_\infty(E(u_1), E(u_2)) = max_{n,m}|D(u_1, S_{n,m}) - D(u_2, S_{n,m})| \tag{5}$$

where $D(u_1, S_{n,m})$ is defined in Equation 3, and $E(u_1)$ is the corresponding point (in $\Re^k$ space) with regard 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 3.1 establishes L_∞ distance over $\Re^k$ as the lower bound of the shortest path distance over G.

Theorem 3.1. *[18] 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,*

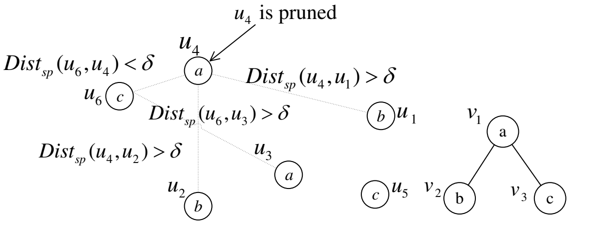
$$L_\infty(E(u_1), E(u_2)) \leq Dist_{sp}(u_1, u_2) \tag{6}$$

Note that shortest path distance and L_∞ distance are both metric distances [11]; thus they satisfy triangle inequality.

4. NEIGHBOR AREA PRUNING

As a result of LLR embedding, all vertices in G have been mapped into points in $\Re^k$. We use a relational table $T(ID, I_1, ..., I_k, L)$ 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 answer a pattern match query, we conduct a multiway join over the converted vector space, not the original graph space. Similarly, each D-join step is conducted over the vector space as well. Thus, to reduce the cost of multiway join, the first step is to remove all the points that do not qualify for D-join (i.e., they don't satisfy join condition in Equation 2) as early as possible. In this section, we propose an efficient pruning strategy called *neighbor area* pruning.


u_4 is pruned
u_4
$Dist_{sp}(u_6,u_4) < \delta$
a
$Dist_{sp}(u_4,u_1) > \delta$
u_6
c
$Dist_{sp}(u_6,u_3) > \delta$
b
u_1
v_1
a
$Dist_{sp}(u_4,u_2) > \delta$
u_3
a
u_2
b
c
u_5
v_2
b
v_3
c

(a) Shortest Path Distances in graph G (b) Query Q

Figure 5: Area Neighbor Pruning

We first illustrate the rationale behind neighbor area pruning using Figure 5. Consider a query Q in Figure 5. If a vertex u labeled 'a' (in G) can match v_1 (in Q) according to Definition 3.1, 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 5, 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 step, until convergence is reached (i.e., no vertices in each 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, for vertex u_4 in Figure 5, if there exists no vertex u' labeled with 'b' where $L_\infty(u_4, u') \leq \delta$, u_4 can also be pruned safely. 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 4.1. *Given a vertex v_i in query Q and its corresponding list R_i in data graph G, for a point u_i in R_i, we define* vertex neighbor area *to be $Area(u_i) = ([(u_i.I_1 - \delta, u_i.I_1 + \delta), ..., (u_i.I_k - \delta, u_i.I_k + \delta)])$, where u_i is a point in $\Re^k$ space. The* list neighbor area *of R_i is defined as $Area(R_i) = \bigcup_{u_i \in R_i} Area(u_i)$.*

Definition 4.2. *Given a list R_i and a vertex u_j, $u_j \in Area(R_i)$, if and only if, for any dimension I_n, $u_j.I_n \in Area(R_i).I_n$, where $Area(R_i).I_n$ is the nth dimension of $Area(R_i)$.*

Theorem 4.1. *Consider a vertex v_i in query Q and assume that v_i has m neighbor vertices v_j (i.e. (v_i, v_j) is an edge), $j = 1, ..., m$, and for each vertex v_j, its corresponding list is R_j in G. If $\exists j, u_i \notin Area(R_j)$, u_i can be safely pruned from the list R_i.*

Proof. (sketch) If $\exists j, u_i \notin Area(R_j)$, there is no vertex u_j labeled as the same as v_j, where $L_\infty(u_i, u_j) \leq \delta$. □

Algorithm 1 Neighbor Area Pruning

```
Require: 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 4.1, Algorithm 1 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 iteration steps (Line 1). The total time complexity of Algorithm 1 is $O(\sum_i |R_i|)$. In the worst case, D-join processing needs $O(\prod_i |R_i|)$. Thus, it is desirable to perform *neighbor area* pruning before join processing.

5. EDGE QUERY PROCESSING

After neighbor area pruning, we obtain n "shrunk" lists, $R_1, \ldots, R_n$, each corresponding to a vertex v_i in query Q. According to the framework in Figure 4, at each step, we need to answer an edge query. In this section, we propose an efficient D-join edge query algorithm.

We first use L_∞ distance in the converted vector space $\Re^k$ to find a candidate match set CL (Definition 5.1):

$$CL = R_1 \underset{L_\infty(u_1,u_2) \leq \delta}{\bowtie} R_2 \tag{7}$$

.

Each candidate match in G 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 a graph labeling technique to obtain the exact shortest path distance $Dist_{sp}(u_i, u_j)$ [7, 15]. All pairs (u_i, u_j) where $Dist_{sp}(u_i, u_j) \leq \delta$ are collected to form the final result RS. Theorem 5.1 proves that the above process guarantees *no false negatives.*

Definition 5.1. *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) $L_\infty(u_1, u_2) \le \delta$.

THEOREM 5.1. *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 computed according to Formula 7, and RS denote the set of all matches of Q_e. Then, $RS \subseteq CL$.*

PROOF. Straightforward from Theorem 3.1. □

Essentially, a D-join is a similarity join over vector space. Existing similarity join algorithms (such as RSJ [2] 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 [2]) cannot work well due to the *dimensionality curse* [14]. 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 a novel data structure to reduce both I/O and CPU costs (Section 5.1). Then, we propose triangle inequality pruning and hash-join to further reduce CPU cost (Section 5.2).

5.1 Data Structures and D-join Algorithm

Due to drawbacks of index-based access in high-dimensional space, we adopt nested loop join strategy for a D-join processing. However, a naive nested loop algorithm to join two lists R_1 and R_2 has serious performance issues: a) High I/O cost: Assume that table T is stored into N disk pages, the total number of I/O in join processing is N^2; b) High CPU cost: The number of distance computations is $|R_1| \times |R_2|$.

In order to perform an efficient D-join for edge query, we propose cluster-based *block nested loop join.* The converted high dimensional space $\Re^k$ is not uniformly distributed; there exist some clusters in the $\Re^k$ space. Inspired by iDistance [14] that answers NN queries in high dimensional space, we first utilize existing cluster algorithms to find clusters in $\Re^k$. In our implementation, we use K-medoids algorithm [11] to find clusters. Note that the clustering algorithm is orthogonal to our D-join algorithm. 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 in $\Re^k$ are given. For each cluster C_i, we find its *cluster center* c_i as a *pivot.* 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 6 depicts our method, where we Euclidean distance is used as the distance function for demonstration; the actual distance function is still L_∞.

We need to perform sequential scan in the nested loop join. To facilitate sequential scan during join processing, we define a total order of the clusters. According to this order, we link all corresponding blocks B_i to form a flat file. We delay the discussion on the total order until the end of this subsection, since it is related to our D-join algorithm.

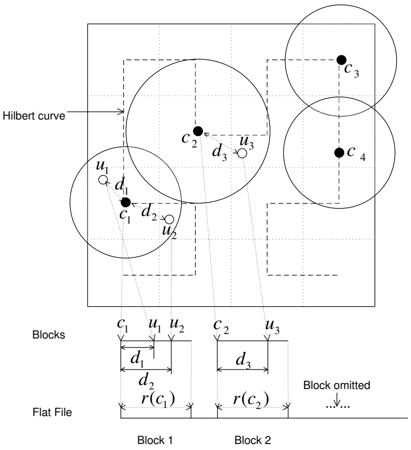


Figure 6: Cluster in $\Re^k$

We adopt block nested loop strategy in D-join algorithm. Given an edge query $Q_e = (v_1, v_2)$, let R_1 and R_2 to 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. D-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 scan R_2 to load a "promising" block B_2 into memory in the inner loop. Then, we perform memory join algorithm between B_1 and B_2. Theorem 5.2 shows the necessary condition that B_2 is a promising block with regard to B_1.

THEOREM 5.2. *Given two blocks B_1 and B_2 (the "outer" and "inner" join operands, respectively), the necessary condition that D-join between B_1 and B_2 produces a non-empty result is:*

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

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. Proven according to triangle inequality. □

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 graph labeling to compute the shortest path distance between u_1 and u_2, that is, $Dist_{sp}(u_1, u_2)$. If $Dist_{sp}(u_1, u_2) \le \delta$, (u_1, u_2) will be inserted into answer set RS. The detailed steps of D-join Algorithm are shown in Algorithm 2.

Now, we discuss the total order for clusters. In Algorithm 2, in each inner loop, we sequentially scan R_2 to load promising blocks into memory with regard to B_1 (the "outer" join operand). Consider two promising blocks B_2 and B_3 with regard to B_1 with corresponding clusters C_2, C_3 and C_1, respectively. According to triangle inequality, $|L_\infty(c_1, c_2) - L_\infty(c_1, c_3)| \le L_\infty(c_2, c_3) \le |L_\infty(c_1, c_2) + L_\infty(c_1, c_3)|$. This means that clusters C_2 and C_3 are near each other in $\Re^k$ space.

All clusters that need to be joined with B_1 should be near each other in $\Re^k$ space. If their corresponding blocks are also adjacent to each other in flat file F, we only need to scan a portion of file F (instead of scanning the whole file) in the inner loop. Due to good locality-preserving behavior, an Hilbert curve is often used in multidimensional databases.

We define the total order for different clusters according to Hilbert order. 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) 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. According to the total order, we can link all corresponding blocks to form a flat file.

Algorithm 2 D-join Algorithm

```
Require: Input: An edge $e = (l_1, l_2)$ 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 $\delta$. $R_1$, 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 $Dist_{sp}(u_1, u_2) \leq \delta$.
Initialize candidate set $CL$ and answer set $RS$.
for each cluster $C_1$ in flat file $F$ do
  if $C_1 \cap R_1 \neq \phi$ then
    Load $C_1$ into memory
    According to Theorem 5.2, find all promising clusters $C_2$ w.r.t $C_1$ in flat file $F$ to form cluster set $PC$.
    Order all clusters $C_2$ in $PC$ according to physical position in flat file $F$.
    for each promising cluster $C_2$ in $PC$ do
      Load cluster $C_2$ into memory.
      Perform memory-based D-Join algorithm on $C_1$ and $C_2$ to find candidate set $CL_1$ (call Algorithm 3).
      Insert $CL_1$ into $CL$.
for each candidate match $(u_1, u_2)$ in $CL$ do
  Compute $Dist_{sp}(u_1, u_2)$ by graph labeling techniques.
  if $Dist_{sp}(u_1, u_2) \leq \delta$ then
    Insert $(u_1, u_2)$ into answer set $RS$
Report $RS$
```

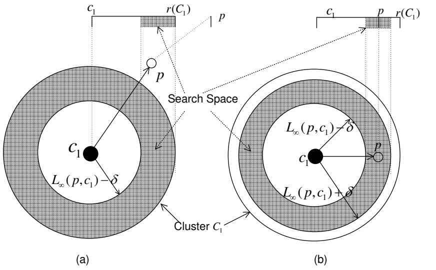

c_1
$r(C_1)$
p
p
Search Space
c_1
$L_\infty(p, c_1) - \delta$
Cluster C_1
(a)
c_1
p
$r(C_1)$
$L_\infty(p, c_1) - \delta$
c_1
p
$L_\infty(p, c_1) + \delta$
(b)

Figure 7: Theorem 5.3

5.2 Memory Join Algorithm

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.

5.2.1 Triangle Inequality Pruning

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

THEOREM 5.3. *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 (C_1 (C_2) is the cluster corresponding to B_1 (B_2)):*

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

PROOF. Directly follows from triangle inequality since L_∞ is metric. □

Figure 7 visualizes the search space in cluster C_1 with regard to point p in C_2 after pruning according to Theorem 5.3.

5.2.2 Hash Join

Hash join in a well-known join algorithm with good performance. The classical hash join does not work for D-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 one 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 5.4. *Given a point p in block B_2 (inner join operand), according to hash function $H(p) = n_1 = \lfloor \frac{p.I_1}{\delta} \rfloor$, p is located at the n_1^{th} bucket. It is only necessary to perform join processing between p and all points of B_1 located in the n_1^{th} bucket. The candidate search space for point p is, $Can_1(p) = b_{n_1}$, where b_{n_1} denotes all points in the n_1^{th} bucket.*

PROOF. It can be proven using L_∞ distance definition. □

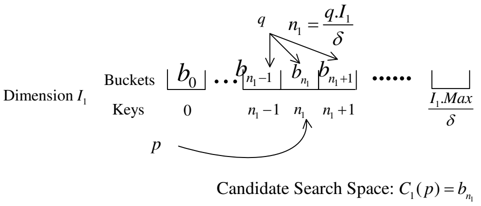


Figure 8: Hash Join

Figure 8 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 5.5 establishes the final search space of p using hash join.

THEOREM 5.5. *The overall search space for vertex p is $Can(p) = Can_1(p) \cap Can_2(p) ... \cap Can_k(p)$, where $Can_i(p)$ ($i = 1, ..., k$) is defined in Theorem 5.4.*

Theorem 5.6 shows that, for a join pair (q, p) (p from B_1 and q from B_2, respectively), if $Dist_\infty(q, p) > 2 \times \delta$, the join pair (q, p) can be safely pruned by the hash join.

THEOREM 5.6. *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 5.4 and 5.5. □

According to two pruning techniques in Theorem 5.3 and join hash, respectively, we propose Memory D-join in Algorithm 3.

Algorithm 3 Memory D-Join Algorithm

```
Require: Input: An edge $e = (v_1, v_2)$ in query $Q$. Two clusters
    are $C_1$ and $C_2$. The distance constraint is $\delta$. $R_1$ 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 $RS = \{(u_1, u_2)\}$, where $L(u_1) = L(v_1)$
    AND $L(u_2) = L(v_2)$ AND $Dist_{sp}(u_1, u_2) \le \delta$.
for each vertex $p$ in $C_2$ do
    if $p \in R_2$ then
        According to Theorem 5.3, find search space in $C_1$ with
        regard to $p$, denoted as $SP(p)$.
        Using hash join in Theorem 5.5, find search space
        $Can(p)$.
        Final search space with regard to $p$ is $SP(p) = SP(p) \cap$
        $Can(p)$.
        for each point $q$ in the search space $SP(p)$ do
            if $L_\infty(q, p) \le \delta$ then
                Insert $(q, p)$ into candidate set $CL$
Report $CL$.
```

6. PATTERN MATCH QUERY

According to the framework in Figure 4, a pattern match query is transformed into a shortest path distance-based multi-way join problem, called *MD-join*. Thus, we first give the detailed steps to answer a multi-way join query in Section 6.1, then we present the cost function (Section 6.2) that drives join order selection discussed in Section 6.3.

6.1 MD-Join Algorithm

In the following discussion, we assume that the join order is specified. As discussed in Section 3, a join order of MD-join corresponds to a traversal order in query graph Q. According to given traversal order (in Q), we visit one edge $e = (v_i, v_j)$ (in Q) from vertex v_i 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*; and if v_j has been visited before, e is called a *backward edge*. The processing of a forward edge query and that of a backward edge query are different. Essentially, forward edge processing is performed by a D-join algorithm (as discussed in Section 5.1), while backward edge processing is a selection operation, which will be discussed shortly.

MD-join is similar to traditional multi-join operation in relational databases and XML databases [25]. Thus, following the same conventions, we define the concept of "status".

Definition 6.1. *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 MD-join algorithm (Algorithm 4) performs a sequential move from the initial status NULL to final status Q, as shown in Figure 3. 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 D-join processing (Line 4 in Algorithm 4). 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 4). We can obtain the 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 D-join algorithm (Algorithm 2), 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 can 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 $Dist_{sp}(u_i, u_j) > \delta$ (we can compute $Dist_{sp}(u_i, u_j)$ by graph labeling technique). 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: $MR(Q'_{i+1}) = \sigma_{(Dist_{sp}(r.v_i, r.v_j) \le \delta)}(MR(Q'_i))$.

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

Algorithm 4 Multi-Distance-Join Algorithm (MD-join)

```
Require: Input: A query graph $Q$ that has $n$ vertices and a pa-
    rameter $\delta$ 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 D-join algorithm (call Algorithm 2).
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:        $MR(e) = \prod_{(R_i.ID, R_j.ID)} ( R_i \bowtie_{\mathrm{Dist_{sp}}(r_i, r_j) \le \delta} R_j )$ (call Algo-
           rithm 2)
11:        $MR(Q'_{i+1}) = MR(Q'_i) \bowtie_{v_i} MR(e)$
12:    else
13:        $MR(Q'_{i+1}) = \sigma_{(Dist_{sp}(r.v_i, r.v_j) \le \delta)}(MR(Q'_i))$
14: Report $MR(Q)$.
```

6.2 Cost Model

It is well-known that different join orders in MD-join algorithm will lead to different performances. The join order selection is based on the cost estimation of edge query. In this section, we discuss the cost of D-join algorithm that answers edge query, which has three components: the cost of block nested loop join (Lines 2-10 in Algorithm 2), 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 [25], we also assume that intermediate results should be stored in a temporary table in disk. We use a set of factors to normalize the cost of D-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 cost; f_S: the average cost of shortest path distance computation cost; f_{IO}: the average cost of storing one match into 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 D-join algorithm can be computed as follows:

$$\begin{aligned}&Cost(e) = \\ &|B_1| \times |B_2| \times \gamma_1 \times f_R + |R_1| \times |R_2| \times \gamma_2 \times f_D + \\ &|CL| \times f_S + |CL| \times \gamma_3 \times f_{IO}\end{aligned} \tag{8}$$

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

$$\gamma_1 = \frac{|AccessedBlocks|}{|B_1| * |B_2|}, \gamma_2 = \frac{|DisComp|}{|R_1| * |R_2|}, \gamma_3 = \frac{|RS|}{|CL|} \tag{9}$$

and where $|AccessedBlocks|$ is the number of accessed blocks in Algorithm 2; $|DisComp|$ is the number of L_∞ distance computations and $|RS|$ (and $|CL|$) is 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 the γ_1, γ_2 and γ_3 for vertex label pair (l_1, l_2) and the distance constraint δ in the ith interval $[(i-1)d, i * d]$ $(1 \le i \le z)$, 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 D-join algorithm, and compute γ_1, γ_2 and γ_3 using Equation 9.

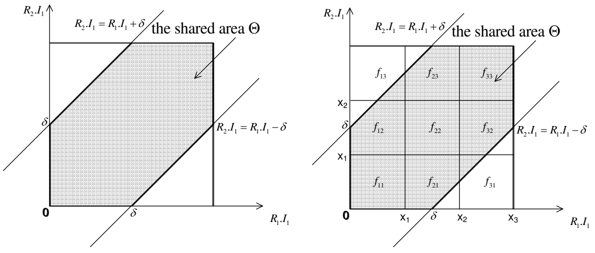
2) Online: Given an edge query $Q_e = (v_1, v_2)$, we look up the estimates for γ_1, γ_2 and γ_3 that were computed offline using the vertex label $(L(v_1), L(v_2))$ and δ.

Next, we discuss how to estimate $|CL|$. Let us first assume that $k = 1$, given an edge query $Q_e = (v_1, v_2)$, the cardinality of candidate match set CL can be denoted as $|CL| = |R_1| \times |R_2| \times \theta$ where θ is the selectivity of D-join based on L_∞ distance. We can regard $R_1.I_1$ 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 $z \le \delta$. Figure 9(a) visualizes the joint random variable z and the area Θ between two curves $y = x + \delta$ and $y = x - \delta$. We can use the following equation to compute selectivity θ.

$$\theta = Pr(z \le \delta) = \int\!\!\int_{|x-y| \le \delta} f(x,y)d(x,y) = \int\!\!\int_{(x,y) \in \Theta} f(x,y)d(x,y)$$

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 9(b). Similar to other histogram methods, we also assume that the distribution in each bucket is uniform. Then, we use a systematic sampling technique [12] to estimate density function in each bucket.

The basic idea of systematic sampling is the following [12]: Given a relation R with N tuples that can be accessed in ascending/desceding 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 [12]. The relations here are R_1 and R_2, and the join attributes are $R_1.I_1$ and $R_2.I_1.R_1$ 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/desceding 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.



(a) (b)

Figure 9: 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

$$f(A) = \frac{|A|}{|SR_1| \times |SR_2|}. \tag{10}$$

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

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

where $area(A \cap \Theta)$ denotes the area of intersection between A and Θ and $area(A)$ denotes the area of A.

We adopt Monte-Carlo methods to estimate $\frac{area(A \cap \Theta)}{area(A)}$. 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 Θ is b. Then, we estimate $\frac{area(A \cap \Theta)}{area(A)}$ to be $\frac{a}{b}$.

Therefore, we have

$$|CL| = \sum\nolimits_{ij} |A_{ij} \cap \Theta| = |R_1| \times |R_2| \times \sum\nolimits_{ij} (f(A_{ij}) \times \frac{area(A_{ij} \cap \Theta)}{area(A_{ij})})$$

The selectivity of θ can be estimated as follows

$$\theta = Pr(z \le \delta) = \sum\nolimits_{ij} |A_{ij} \cap \Theta| = \sum\nolimits_{ij} (f(A_{ij}) \times \frac{area(A_{ij} \cap \Theta)}{area(A_{ij})}) \tag{11}$$

where $f(A_{ij})$ is estimated by Equation 10.

If $k > 1$, according to Theorem 5.1, we have

$$CL = R_1 \underset{Max_{1 \le i \le k}(|R_1.I_i - R_2.I_i| \le \delta)}{\bowtie} R_2$$

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

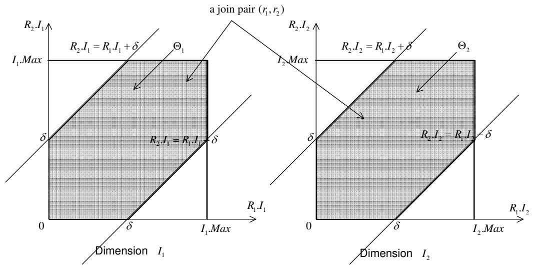


Figure 10: Multi-Dimension Selectivity Estimation

x_i and y_i. Let $z_i = |x_i - y_i|$ denote the joint random variable.

$$\theta = Pr(Max(z_1, ..., z_k) \leq \delta)) = Pr((z_1 \leq \delta) \wedge ... \wedge (z_k \leq \delta)) \quad (12)$$

To compute Equation 12, we propose two techniques: dimension-independence assumption and sampling-based method.

1) Dimension-Independence Assumption

We assume that every dimension I_i in vector space $\Re^k$ is independent of each other. Thus, we have

$$Pr((z_1 \leq \delta) \wedge ... \wedge (z_k \leq \delta)) = Pr(z_1 \leq \delta) \times ... \times Pr(z_k \leq \delta). \quad (13)$$

where $Pr(z_i \leq \delta)$ $(i = 1, ..., k)$ can be computed using Equation 12. Experiments indicate that Equation 13 cannot provide accurate selectivity estimation. since dimensions in $\Re^k$ space are correlated. In order to obtain more accurate estimation, we propose sampling.

2) Sampling-based Method

Consider two lists R_1 and R_2 to be joined. Assume, for simplicity, $k = 2$. In Figure 10, $Pr(Max(z_1, z_2) \leq \delta))$ is the probability that a vertex pair falls into both shared areas Θ_1 and Θ_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 $Max(|u_1.I_i - u_2.I_i|) \leq \delta$, $(1 \leq i \leq k)$, $Pr(Max(z_1, ..., z_k) \leq \delta) = \frac{M}{|SR_1| * |SR_2|}$. 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 the equal probability of being chosen as a sample.

6.3 Join Order Selection

The join order selection can be performed by adopting the traditional dynamic programming algorithm [25] using the cost model introduced in the previous section. However, this solution is inefficient due to 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. 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 8) is minimum of all forward edges.

7. 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 Network".

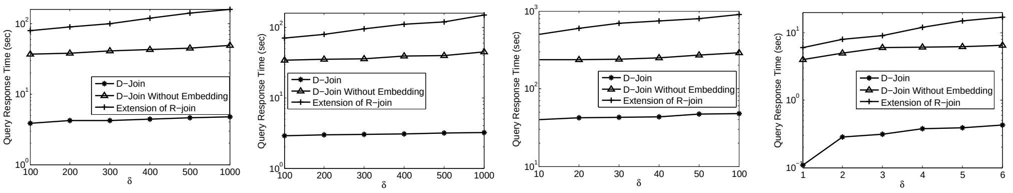
b) Scale-Free Model: We use the graph generator *gengraphwin* (www.cs.sunysb.edu/ algorith/implement/viger/distrib/). We generate a large graph G with 100K vertices satisfying power-law distribution. Default value of parameter α is set to 2.5. There are 89198 vertices and 115526 edges in the maximal connected component of G. We can sequentially perform our method in each connected component of G. This dataset is denoted "SF Network".

In the above two datasetes, 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. For an edge $e = (u_1, u_2)$ in G, its weight 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.

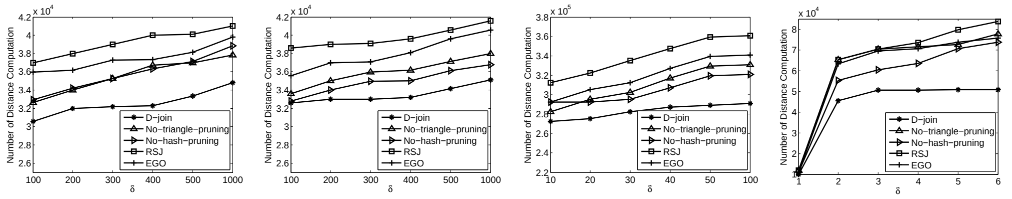
d) Yeast. This is a protein-to-protein interaction network in budding yeast (http://vlado.fmf.uni-lj.si/pub/networks/data/). Each vertex denotes a protein and an edge denotes the interaction between two corresponding proteins. We delete 'self-loop' edges in the original dataset. There are 13 types of 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.

Exp.1 We first evaluate the performance of LLR embedding technique. In this experiment, we consider D-join algorithm to answer edge query. For clustering, we use the k-medoids algorithm. The value of the cluster number depends on the available memory size for join processing. We choose two alternative methods for performance comparison: the extension of R-join algorithm [6] 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 11, which shows



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

Figure 11: Evaluating Embedding Technique



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

Figure 12: Number of Distance Computation

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 (we do not report pruning power here due to the space limitation). Finally, the extension of R-join cannot work well for our problem, since no pruning techniques are introduced to reduce the search space.

Exp.2 In this experiment, 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 12 and 13, 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 Exp.4. In 'No-triangle-pruning' (see Figure 13) method, we do not utilize the triangle inequality pruning technique, and only use the hash join technique. In 'No-hash-pruning' method, we do not utilize the hash join pruning technique, and only use triangle inequality pruning. We also compare our techniques with two alterative similarity join algorithms: RSJ [2] and EGO [1]. Figure 12 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 12 and 13, 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 13(d), since Yeast dataset has only about 2000 vertices.

Exp.3 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 technique can capture data distribution in $\Re^k$ space, thus, it can provide better estimation, as shown in Figure 14.

Exp.4 In this experiment, 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 6. 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-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 15. 'No join order selection' is much slower than MD-join algorithm. Figure 15 also demonstrates that randomly defining join order cannot work as well as MD-join algorithm.

8. CONCLUSIONS

In this paper, we propose a novel pattern match problem over a large graph G. We transform vertices in G into points in a vector space via graph embedding methods, coverting a pattern match query into a distance-based multi-way join problem over vector space. Several pruning techniques are developed to reduce the search space significantly, such as neighbor area pruning, triangle inequality pruning and hash join. We also design a cost estimation technique to find a cheap query plan (i.e., join order).

9. REFERENCES

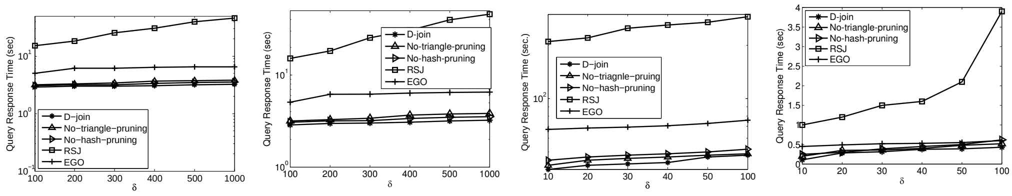
[1] C. Böhm, B. Braunmüller, F. Krebs, and H.-P. Kriegel. Epsilon grid order: An algorithm for the similarity join on massive high-dimensional data. In *SIGMOD*, 2001.

[2] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Efficient processing of spatial joins using r-trees. In *SIGMOD*, 1993.

[3] E. P. F. Chan and H. Lim. Optimization and evaluation of shortest path queries. *VLDB J.*, 16(3), 2007.

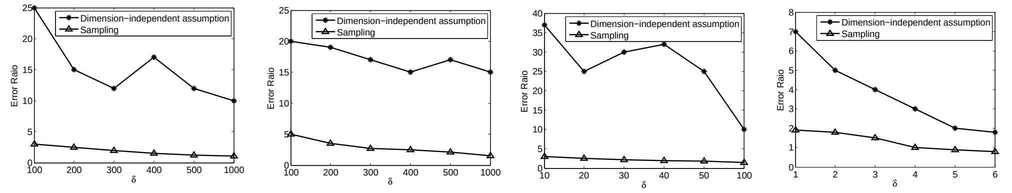
[4] Y. Chen and Y. Chen. An efficient algorithm for answering graph reachability queries. In *ICDE*, 2008.

[5] J. Cheng, Y. Ke, W. Ng, and A. Lu. fg-index: Towards verification-free query processing on graph



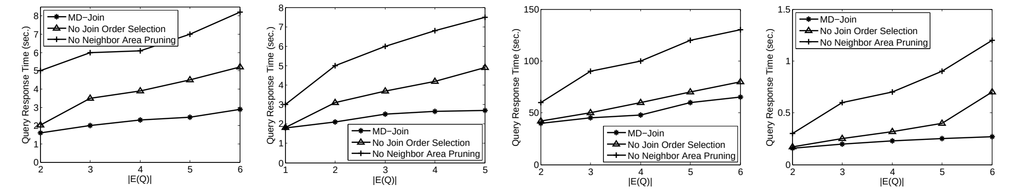
(a) ER Network (b) SF Network (c) Citeseer Network (d) Yeast Network

Figure 13: Edge Query Response Time



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

Figure 14: Cost Estimation



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

Figure 15: Pattern Match Query Response Time VS. $|E(Q)|$

databases. In *SIGMOD*, 2007.

[6] J. Cheng, J. X. Yu, B. Ding, P. S. Yu, and H. Wang. Fast graph pattern matching. In *ICDE*, 2008.

[7] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. *SIAM J. Comput.*, 32(5), 2003.

[8] J. H. D.W. Williams and W. Wang. Graph database indexing using structured graph decomposition. In *ICDE*, 2007.

[9] G. Gou and R. Chirkova. Efficient algorithms for exact ranked twig-pattern matching over graphs. In *SIGMOD*, 2008.

[10] P. Y. H. Jiang, H. Wang and S. Zhou. Gstring: A novel approach for efficient search in graph databases. In *ICDE*, 2007.

[11] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers, 2000.

[12] B. Harangsri, J. Shepherd, and A. H. H. Ngu. Selectivity estimation for joins using systematic sampling. In *DEXA Workshop*, 1997.

[13] H. He and A. K. Singh. Closure-tree: An index structure for graph queries. In *ICDE*, 2006.

[14] H. V. Jagadish, B. C. Ooi, K.-L. Tan, C. Yu, and R. Zhang. idistance: An adaptive b+-tree based indexing method for nearest neighbor search. *ACM Trans. Database Syst.*, 30(2), 2005.

[15] C. Jiefeng and J. X. Yu. On-line exact shortest distance query processing. In *EDBT*, 2009.

[16] 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.

[17] N. Linial, E. London, and Y. Rabinovich. The geometry of graphs and some of its algorithmic applications. *Combinatorica*, 15(2), 1995.

[18] 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), 2003.

[19] D. Shasha, J. T.-L. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In *PODS*, 2002.

[20] 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.

[21] Y. Tian and J. M. Patel. Tale: A tool for approximate large graph matching. In *ICDE*, pages 963–972, 2008.

[22] H. Tong, C. Faloutsos, B. Gallagher, and T. Eliassi-Rad. Fast best-effort pattern matching in large attributed graphs. In *KDD*, 2007.

[23] S. Trißl and U. Leser. Fast and practical indexing and querying of very large graphs. In *SIGMOD*, 2007.

[24] H. Wang, H. He, J. Yang, P. S. Yu, and J. X. Yu. Dual labeling: Answering graph reachability queries in constant time. In *ICDE*, 2006.

[25] Y. Wu, J. M. Patel, and H. V. Jagadish. Structural join order selection for xml query optimization. In *ICDE*, 2003.

[26] X. Yan, P. S. Yu, and J. Han. Graph indexing: A frequent structure-based approach. In *SIGMOD*, 2004.

[27] S. Zhang, M. Hu, and J. Yang. Treepi: A novel graph indexing method. In *ICDE*, 2007.

[28] P. Zhao, J. X. Yu, and P. S. Yu. Graph indexing: Tree + delta >= graph. In *VLDB*, 2007.