2018 International Conference on Computational Science and Computational Intelligence (CSCI)

On Evaluation of Graph Pattern Matching in Large Databases

[1]Yangjun Chen, [2]Guo Bin, and [3]Xinyue Huang
Dept. Applied Computer Science,
515 Portage Ave. University of Winnipeg, Canada
[1]y.chen@uwinnipeg, [2]guo-b75@webmail.uwinnipeg.ca, [3]huangxy19910321@163.com

***Abstract*—Recently, graph databases have been received much attention in the research community due to their extensive applications in practice, such as social networks, biological networks and World Wide Web, which bring forth a lot of challenging data management problems including subgraph search, shortest-path queries, reachability verification, pattern matching, and so on. Among them, the graph pattern matching is to find all matches in a data graph *G* for a given pattern graph *Q* and is more general and flexible compared with other problems mentioned above. In this paper, we address a kind of graph matching, the so-called *graph matching with δ*, by which an edge in *Q* is allowed to match a path of length ≤ δ in *G*. In order to reduce the search space when exploring *G* to find matches, we propose a new index structure and a novel pruning technique to eliminate a lot of unqualified vertices before join operations are carried out. Extensive experiments have been conducted, which show that our approach makes great improvements in running time compared to existing ones.**

Keywords—graph matching, δ-transitive-closures, triangle consistency, join ordering

I. Introduction

Nowadays, in numerous applications, including social networks, biological networks, and WWW networks, as well as geographical networks, data are normally organized into graphs with vertices for objects and edges for their relationships. The burgeoning size and heterogeneity of networks have inspired extensive interests in querying a graph in different ways, such as *subgraph search* [1], *shortest-path queries* [2], *reachability queries* [3, 4], and *pattern matching queries*. Among them, the pattern matching is very challenging, by which we are asked to look for all matches of a certain pattern graph *Q* in a data graph *G*, each of which is isomorphic to *Q* or satisfies certain conditions related to *Q*. As a key ingredient of many advanced applications in large networks, the graph matching is conducted in many different domains: (1) in the traditional relational database research, a schema is often represented as a graph. By matching of data instances we are required to map a schema graph to part of a data graph to check any updating of data for consistency; (2) in a large metabolic network, it is desirable to find all protein substructures that contain an α-β-barrel motif, specified as a cycle of β strands embraced by an α-helix cycle; (3) in the computer vision, a scene is naturally represented as a graph $G(V, E)$, where a feature is a vertex in V and an edge in E stands for a geographical adjacency of two features. Then, a scene recognition is just a matching of a graph standing for part of a scene to another stored in databases.

The first two applications mentioned above are typically exact matching, by which the graph isomorphism checking, or subgraph isomorphism is required. In other words, the mapping between two graphs must be both vertex-label preserving and edge preserving in the sense that if two vertices in the first graph are linked by an edge, they are mapped to two vertices in the second by an edge as well. It is well-known that the subgraph isomorphism checking is *NP*-complete. A lot of work has been done on this problem, but most of them are for special kinds of graphs, such as [6] for planar graphs and [7] for valence graphs, or by establishing indexes, or uses some kinds of heuristics to speed up the working process. The third application is a kind of inexact matching. First, two matching features from two graphs may disagree in some way due to different observation of a same object. Secondly, between two adjacent features in a graph may there be some more features in another graph figured out by a more meticulous description. This leads to a new kind of queries, called *pattern matching with δ* (or *graph matching with δ*), where δ is a number, by which an edge in a query graph is allowed to match a path of length ≤ δ in a data graph. More specifically, two adjacent vertices v and v' in a query graph Q can match two vertices u and u' in a data graph G with $label(v) = label(u)$ and $label(v') = label(u')$ if the distance between u and u' is ≤ δ. Here, the distance between u and u' is defined to be the weight of the shortest path connecting these two vertices, denoted as $dist(u, u')$. Note that when δ = 1, the problem reduces to the normal subgraph isomorphism.

In this paper, we address this problem and propose a new method to evaluate pattern matching with δ based on a new concept of δ-transitive closure of G, used as an index, as well as a filtering method to remove useless data before a join is conducted. Besides, the bit mapping technique is also integrated into our filtering method to speed up computation.

II. Problem Statements

In this section, we give a formal definition of the pattern matching queries with δ over directed weighted graphs G. First of all, G should be a *Weakly Connected Component* (*WCC*) (i.e., the undirected version of G is connected); otherwise, we can decompose G into a collection of *WCC*s and perform pattern matching over each *WCC* in turn. Secondly, we will use the shortest path length to measure the distance between two vertices. However, our approach is not restricted to this distance function and it can also be applied to other metrics without any difficulty.

Definition 2.1 (*Data Graph G*) A data graph $G = (V(G), E(G), \Sigma)$ is a vertex-labeled, directed, weighted, and weakly connected graph. Here, $V(G)$ is a set of labeled vertices, $E(G)$

978-1-7281-1360-9/18/$31.00 ©2018 IEEE
DOI 10.1109/CSCI.2018.00239

is a set of edges (ordered pairs) each with a weight represented as a nonnegative number, and Σ is a set of vertex labels. Each vertex $u \in V(G)$ is assigned a label $l \in \Sigma$, denoted as $label(v) = l$.

Definition 2.2 (*Query Graph Q*) A query Q is a vertex-labeled and directed graph, $Q = (V(Q), E(Q))$. Here, $V(Q)$ is a set of labeled vertices, and $E(Q)$ is a set of edges. Each vertex $v \in V(Q)$ is also assigned a label $l \in \Sigma$.

Definition 2.3 (*List* $\boldsymbol{D}[l]$) Given a data graph G, we use $\boldsymbol{D}[l]$ to represent a list that includes all those vertices u in G whose labels are $l \in \Sigma$, i.e., $label(u) = l$ for each $u \in \boldsymbol{D}[l]$.

Sometimes we also use the notation $\boldsymbol{D}$, instead of $\boldsymbol{D}[l]$, if its label l is clear from the context. Let v be a vertex in Q with $label(v) = l$. We also call $\boldsymbol{D}[l]$ the domain of v.

Definition 2.4 (*Edge Query*) Given a data graph G, an edge $e = (v_i, v_j)$ in a query graph Q and a parameter δ, the evaluation of e reports all matching pairs $<u_i, u_j>$ in G if the following conditions hold:

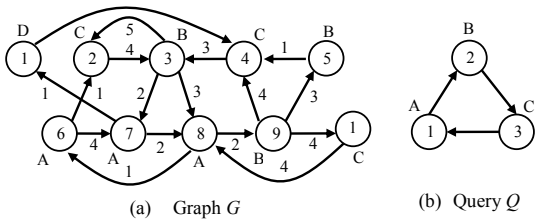
1) $label(u_i) = label(v_i)$ and $label(u_j) = label(v_j)$;
2) The distance from u_i to u_j in G is not larger than δ. That is, $Dist(u_i, u_j) \leq \delta$.

Definition 2.5 (*Pattern Matching Query with δ*) Given a data graph G, a query graph Q with n vertices $\{v_1, ..., v_n\}$ and a parameter δ, the evaluation of Q reports all matching results $<u_1, \ldots u_n>$ in G if the following conditions hold:

1) $label(u_i) = label(v_i)$ for $1 \leq i \leq n$;
2) For any edge $(v_i, v_j) \in Q$, the distance between u_i and u_j in G is no larger than δ, i.e., $Dist(u_i, u_j) \leq \delta$.

In Fig. 1(a), we show a simple graph, in which the numbers inside the vertices are their IDs and the letters attached to them are their labels. There are altogether 4 labels: $\Sigma = \{A, B, C, D\}$. Each edge is also associated with a number, representing its weight. In Fig. 1(b), we show a simple query Q.

Since Q contains three vertices labeled with $A, B, C \in \Sigma$, respectively, three lists: $\boldsymbol{D}[A]$, $\boldsymbol{D}[B]$, $\boldsymbol{D}[C]$ from G will be constructed. For example, $\boldsymbol{D}[A] = \{u_6, u_7, u_8\}$.



(a) Graph G (b) Query Q

Fig. 1: Examples of a data graph G and a query Q

In addition, the query Q also has 3 query edges, $\{(v_1, v_2), (v_2, v_3), (v_3, v_1)\}$. If the parameter $\delta = 5$, the pairs matching a query edge (v_1, v_2) (with labels (A, B)) are $\{<u_6, u_3>, <u_7, u_9>, <u_8, u_9>, <u_8, u_5>\}$. The matching result of the whole query Q is $\{<u_8, u_9, u_{10}>, <u_7, u_9, u_4>\}$. If $\delta = 6$, the pairs matching (v_1, v_2) is the same as $\delta = 5$. But the matching result of the whole query Q will be augmented by adding $\{<u_8, u_9, u_4>\}$.

The common symbols used in this paper are summarized in Table 1.

Table 1. Meaning of used symbols

Directed data graph G		Directed query graph Q	
$V(G)$	vertex set of G	$V(Q)$	vertex set of Q
$E(G)$	edge set of G	$E(Q)$	edge set of Q
u_i	a vertex in G	v_i	a vertex in Q
$Label(u_i)$	label of u_i	n	number of vertices in Q
$Dist(u_i, u_j)$	distance between-u_i and u_j	m	number of edges in Q

In a similar way, we can also define the problem for undirected, weighted graphs, by simply removing directions of edges when calculating the distance between two vertices.

Finally, we should notice that if G contains negative-weight cycles the shortest path from a vertex to another may not be well defined. It is because by a negative-weight cycle C we have some edges associated with negative weights and all the weights of the edges on C can sum up to a negative value. Thus, we can always find a path with lower weight by following such a cycle. In this paper, negative-weight cycles will not be considered.

III. Construction of Indexes

In this section, we begin to describe our method. In a nutshell, our algorithm comprises three steps. In the first step, we will construct off-line a δ-transitive closure for a certain δ-value as an index over G, by which a set of binary relations as illustrated in Fig. 2(b) will be constructed. (In general, how large δ is set is determined according to historical query logs.) Thus, when a query Q is submitted, for each edge $e = (v_i, v_j)$ in Q, a relation corresponding to $<label(v_i), label(v_j)>$, referred to as R_{ij}, will be loaded from hard disk to main memory. In the second step, we will do a relation filtering. Finally, in the third step, a series of *equi-joins* on the reduced relations will be conducted to get the final results.

In the subsequent discussion, we mainly concentrate on the index construction. The description of the second and third steps will be shifted to Section IV.

A. Constructing δ-transitive closures

As with [5], we will construct an index for graphs to speed up computation. However, instead of 2-hop covers, we will construct *δ-transitive closures* for them.

Definition 3.1 (*δ-transitive closures*) Let $G = (V, E, \Sigma)$ be a directed, weighted graph. Let $\delta > 0$ be a positive number. A δ-transitive closure of G, denoted G^δ, is a graph such that $V(G^\delta) = V$, and there is an edge $<u, u'> \in E(G^\delta)$ if and only if $dist(u, u') \leq \delta$.

Example 1 Consider the graph shown in Fig. 1(a) again. For $\delta = 5$, its δ-transitive closure G^5 is a graph as shown in Fig. 2(a), which can be stored as a set of binary relations as shown in Fig. 2(b), in which each tuple is a pair of vertices, but associated with their corresponding shortest distance ≤ 5 for computation convenience.

This concept is motivated by an observation that δ tends to be small in practice. For instance, in a social network where each vertex u represents a person and each edge represents a relationship like *parent-of*, *brother-of*, *sister-of*, *uncle-of*, and so on, if the weight of each edge is set to be 1, then to check whether a person is a relative of somebody else, setting $\delta = 6$ or larger is not so meaningful. As another example, we consider an application in forensics and assume that a detective may want to investigate an individual who is related to a known criminal through some

relationships, such as money laundry, human trafficking, and so on. Then, setting $\delta > 5$ may not be quite helpful for the detective to find some significant evidence.

Clearly, for different applications, we can adjust the values of δ to precompute G^{δ}. One choice for this task is to use an algorithm for finding all-pairs shortest-paths. However, since such an algorithm needs to store the output in a matrix, it is not so suitable for very large graphs. In this case, we can utilize a single-source shortest-paths algorithm and run it to generate G^{δ}. Specifically, the following two steps will be conducted:

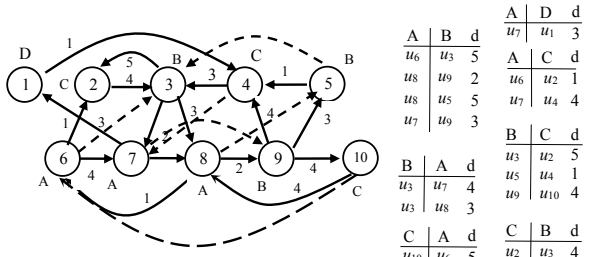
i) Remove all those edges from $G(V, E)$, whose weight is $> \delta$.

ii) Run the single-source shortest-paths algorithm $|V|$ times each with a different vertex as a source.

If all edge weights are nonnegative, the best algorithm for this problem uses *Fibonacci heap* and needs only $O(|V|^2 \lg\delta + |V||E|)$ time.

In comparison with 2-hop covers, the main advantage of δ-transitive closures is three-folds:

- Less space is required to store a δ-transitive closure than a 2-hop cover if δ is not set so large.
- Much less time is needed to create a δ-transitive closure of G than a 2-hop cover of G. It is because to generate a 2-hop cover the entire transitive closure of G has to be first created, which requires $O(|V|^3)$ time in the worst case, much higher than the time complexity for creating G^{δ}.
- No on-line time is needed to form R_{ij}'s while by the methods with 2-hop covers R_{ij}'s have to be generated on-the-fly when a query Q arrives. This is done by using the 2-hop data structures for the relevant edges in Q. Obviously, the response time to queries can be greatly delayed.



(a)

A	B	d
u_6	u_3	5
u_8	u_9	2
u_8	u_5	5
u_7	u_9	3

A	D	d
u_7	u_1	3

A	C	d
u_6	u_2	1
u_7	u_4	4

B	C	d
u_3	u_2	5
u_5	u_4	1
u_9	u_{10}	4

B	A	d
u_3	u_7	4
u_3	u_8	3

C	A	d
u_{10}	u_6	5
u_{10}	u_8	4
u_4	u_7	5

C	B	d
u_2	u_3	4
u_4	u_3	3

A	A	d
u_6	u_7	4
u_7	u_8	2
u_8	u_6	1

B	B	d
u_3	u_5	5
u_5	u_3	4
u_9	u_5	3

D	C	d
u_1	u_4	1

(b)

Fig. 2: An example of δ-transitive closures

B. Relation Signatures and Vertex Counters

For each relation in a δ-transitive closure, we can also establish two extra data structures for efficiency: one is a set of *bit string pairs* with each associated with an R_{ij}, called the *relation signature* of R_{ij}; and the other is a set of counters each for a single vertex in G.

a) *Relation signatures*: Consider $G(V, E, \Sigma)$. Let $\Sigma = \{l_1, \ldots, l_k\}$. We will divide all the vertices into $|\Sigma|$ disjoint lists, denoted as $\boldsymbol{D}[l_1], \ldots, \boldsymbol{D}[l_k]$ such that $\boldsymbol{D}[l_1] \cup \ldots, \cup \boldsymbol{D}[l_k] = V$, $\boldsymbol{D}[l_i] \cap \boldsymbol{D}[l_j] = \phi$ for $i \neq j$, and all the vertices in a $\boldsymbol{D}[l_i]$ have the same label l_i. Then, we sort all vertices in $\boldsymbol{D}[l_i]$ in ascending order by their vertex IDs and refer to each vertex by its order number. For example, for the graph shown in Fig. 1(a), we have $\boldsymbol{D}[\mathrm{A}] = \{u_6, u_7, u_8\}$. Thus, u_6 is the first vertex, u_7 the second, and u_8 the third in $\boldsymbol{D}[\mathrm{A}]$. In this way, a vertex in G can be referred to as a pair (l, i), where l is a label, and i is the order number of the vertex in $\boldsymbol{D}[l]$. For example, u_6 can be represented as (A, 1). Let R be a relation in G^{δ} corresponding to a pair of vertex labels (l, l'). Denote by $R[1]$ and $R[2]$ all vertices in the first and the second column, respectively. Then, all the vertices in $R[1]$ ($R[2]$) can be represented by a bit string s of length $|\boldsymbol{D}[l]|$ (resp. $|\boldsymbol{D}[l']|$) such that $s[i] = 1$ if the ith vertex of $\boldsymbol{D}[l]$ (resp. $\boldsymbol{D}[l']|$) appears in $R[1]$ (resp. $R[2]$). Otherwise, $s[i] = 0$. Let s, s' be the bit string for $R[1]$ and ($R[2]$), respectively. We call $S = [s \mid s']$ the signature of R, respectively referred to as $R.S[1] = s$ and $R.S[2] = s'$.

Example 2 Continued with Example 1. In Fig. 3(a), we show all the lists each associated with a label in G^{δ}. In Fig. 3(b), we redraw the relations shown in Fig. 2(b) with each vertex replaced with its order number in the corresponding list. Their signatures are shown in Fig. 3(c).

A	
1	u_6
2	u_7
3	u_8

B	
1	u_3
2	u_5
3	u_9

C	
1	u_2
2	u_4
3	u_{10}

D	
1	u_1

(a)

Relation signatures:

R_1: [111 | 111]
R_2: [010 | 1]
R_3: [110 | 110]
R_4: [100 | 011]
R_5: [111 | 111]
R_6: [011 | 111]
R_7: [110 | 100]
R_8: [111 | 111]
R_9: [111 | 111]
R_{10}: [1 | 010]

(c)

R_1:

A	B	d
1	1	5
3	3	2
3	2	5
2	3	3

R_2:

A	D	d
2	1	3

R_3:

A	C	d
1	1	1
2	2	4

R_4:

B	A	d
1	2	4
1	3	3

R_5:

B	C	d
1	1	5
2	2	1
3	3	4

R_6:

C	A	d
3	1	5
3	3	4
2	2	5

R_7:

C	B	d
1	1	4
2	1	3

R_8:

A	A	d
1	2	4
2	3	2
3	1	1

R_9:

B	B	d
1	3	5
2	1	4
3	2	3

R_{10}:

D	C	d
1	2	1

(b)

Fig. 3: Illustration for relation signatures

b) *Counters*: By using relation signatures, we are able to indicate whether a vertex appears in a column of a certain relation. However, the information on how many times it appears in that column is missing. So, for each vertex u in a $\boldsymbol{D}[l]$, we will also associate it with a set of counters each for a column in a different relation, in which it occurs. Assume that u is a vertex appearing in the first column of some R_i. Then, it will have a counter, denoted as $u.C_i[1]$, to record how many times it appears in that column of R_i. In general, if it appears in k columns (respectively from different relations), it will be associated with k counters. For example, u_6 (represented by (A, 1)) appears in 6 columns: $R_1[1]$, $R_3[1]$, $R_4[2]$, $R_6[2]$, $R_8[1]$ and $R_8[2]$ (see Fig. 3(b)). Thus, u_6 will be associated with 6 counters: $u_6.C_1[1] = 1$, $u_6.C_3[1] = 1$, $u_6.C_4[2] = 1$, $u_6.C_6[2] = 1$, $u_6.C_8[1] = 1$, $u_6.C_8[2] = 1$. But u_8 (represented by (A, 3)) appears only in 5 columns. Thus, it

has 5 counters: $u_8.C_1[1] = 2$, $u_8.C_4[2] = 1$, $u_8.C_6[2] = 1$, $u_8.C_8[1] = 1$, $u_8.C_8[2] = 1$.

Obviously, both relation signatures and counters for vertices can be established off-line as part of indexes.

IV. RELATION FILTERING

In this section, we present our algorithm for relation filtering. First, we give the algorithm in Subsection *A*. Then, we prove the correctness and analyze the computational complexity in Subsection *B*.

A. Algorithm Description

When a query Q with parameter δ' arrives, a simple way to evaluate it can be described as follows. First, we locate all the relevant relations (in G^δ). Then, for each edge $(v_i, v_j) \in Q$, remove all those tuples $<u, u'>$ with $dist(u, u') > \delta'$ from the corresponding relation $R_{ij} = R(label(v_i), label(v_j))$. Next, we join such R_{ij}'s to form the final result.

However, we can do better by filtering all those tuples which cannot contribute to the final result before the joins are carried out. To this end, we need a new concept of *triangle consistency*.

Definition 4.1 (*triangle consistency*) Let Q be a query with parameter δ'. Let (v_i, v_j) be an edge in Q. A tuple $t = <u, u'> \in R_{ij}$ in G^δ with $\delta \geq \delta'$ is said to be *triangle consistent* with respect to a vertex v_k ($k \neq i, j$) in Q if for R_{jk}, or R_{kj}, and R_{ik}, or R_{ki} in G^δ, one of the following conditions is satisfied:

1) if v_k is incident to v_j but not to v_i, then there exists u'' such that $<u', u''> \in R_{jk}$ if $(v_j, v_k) \in Q$, or $<u'', u'> \in R_{kj}$ if $(v_k, v_j) \in Q$;
2) if v_k is incident to v_i but not to v_j, then there exists u'' such that $<u', u''> \in R_{ik}$ if $(v_i, v_k) \in Q$, or $<u'', u'> \in R_{ki}$ if $(v_k, v_i) \in Q$;
3) if v_k is incident to both v_i and v_j, then there exists u'' such that
- $<u, u''> \in R_{ik}$ and $<u'', u'> \in R_{kj}$ if $(v_i, v_k), (v_k, v_j) \in Q$; or
- $<u, u''> \in R_{ik}$ and $<u', u''> \in R_{jk}$ if $(v_i, v_k), (v_j, v_k) \in Q$; or
- $<u'', u> \in R_{ki}$ and $<u'', u'> \in R_{jk}$ if $(v_k, v_i), (v_k, v_j) \in Q$; or
- $<u'', u> \in R_{ki}$ and $<u', u''> \in R_{kj}$ if $(v_k, v_i), (v_j, v_k) \in Q$.

In each of the above three cases, u'' is said to be consistent with t. The motivation of this concept is that if $<u, u'>$ in R_{ij} is not triangle consistent with some v_k ($k \neq i, j$) in Q incident to v_i or v_j, it cannot be part of any answer to Q. Thus, only if $<u, u'>$ in R_{ij} is triangle consistent with any vertex in Q incident to v_i, v_j, or both, $<u, u'>$ can be possibly part of an answer. In this case, we say, $<u, u'>$ is triangle consistent. Notice that if neither v_i nor v_j is incident to any vertex in Q, all the tuples in R_{ij} are trivially considered to be triangle consistent.

The following example helps for illustration.

Example 3 Consider the query with parameter $\delta' = 5$ shown in Fig. 1(b). To evaluate this query against the graph shown in Fig. 1(a), we will first load three relations into main memory: R_1, R_5, and R_6 (shown in Fig. 3(b)) from G^δ. For illustration, we show the data in the three relations as a graph in Fig. 4(a).

In this graph, the tuple represented by edge $(u_8, u_9) \in R_{12}$ (= R_1 shown in Fig. 3(b)) is triangle consistent with respect to v_3 in Q. But edge $(u_3, u_2) \in R_{23}$ (= R_5 shown in Fig. 3(b)) is not triangle consistent with v_1 since we have an edge (v_3, v_1) labeled with <C, A> in Q, but we do not have an edge going from u_2 to a vertex in $\boldsymbol{D}(label(v_1)) = \boldsymbol{D}(\mathrm{C})$. (Note that edge (u_6, u_2) in Fig. 4(a) is just in the reverse direction of edge (v_3, v_1) in Q.)

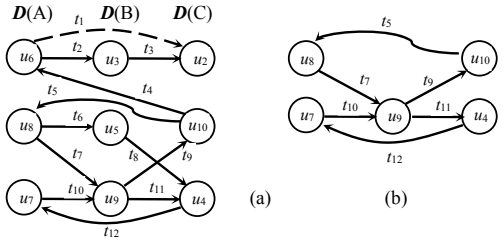


Fig. 4: Data structures for vertices

In fact, only the tuples represented by the edges in the subgraph shown in Fig. 4(b) are triangle consistent while all the other edges not in this subgraph are not. From this example, we can see that by a relation filtering the sizes of relations can be dramatically decreased. Our purpose is to find an efficient way to remove all the triangle inconsistent data from the relevant relations before the joins over them are actually performed.

In the following, we will first devise a procedure to check the triangle consistency of each tuple $t = <u, u'>$ in every R_{ij} with respect to a single vertex v_k ($k \neq i, j$) in Q. Then, a general algorithm for removing all the useless data will be presented.

Below is the formal description of the algorithms. Besides the data structures described in Section III, each t in every relation R_{ij} (corresponding to $(v_i, v_j) \in Q$) is additionally associated with two kinds of extra data structures for efficient computation:

- $\alpha[t, k]$ - the number of vertices u in $\boldsymbol{D}[k]$ (for each $k \neq i, j$), each of which is triangle consistent with t. Initially, each $\alpha[t, k] = 0$.
- $\beta[t]$ - a set containing all those vertices u in $\boldsymbol{D}[k]$ (for each $k \neq i, j$) such that each of them is triangle consistent with t. In $\beta[t]$, each element is referred to as (k, u), indicating that it is a vertex in $\boldsymbol{D}[k]$. Initially, each $\beta[t] = \phi$ (empty).

In addition, a global variable L is used to store all the tuples which are found not triangle consistent with some v in Q, to facilitate the propagation of inconsistency.

Algorithm 1: *tcControl*(R_{ij}, v_k)

Input: R_{ij}, v_k.
Output:
1. **if** v_k is incident to v_j or v_j but not to both **then**
2. **if** $(v_j, v_k) \in Q$ **then** $R := R_{ij}$; $R' := R_{jk}$; $a := 2$; $b := 1$;
3. **if** $(v_k, v_j) \in Q$ **then** $R := R_{ij}$; $R' := R_{kj}$; $a := 2$; $b := 2$;
4. **if** $(v_i, v_k) \in Q$ **then** $R := R_{ij}$; $R' := R_{ik}$; $a := 1$; $b := 1$;
5. **if** $(v_k, v_i) \in Q$ **then** $R := R_{ij}$; $R' := R_{ki}$; $a := 1$; $b := 2$;
6. call *check*-1(R, R', a, b);
7. **if** v_k is incident to both v_i and v_j **then**
8. **if** $(v_i, v_k), (v_k, v_j) \in Q$ **then** *check*-2(i, j; i, k; k, j);
9. **if** $(v_i, v_k), (v_j, v_k) \in Q$ **then** *check*-2(i, j; i, k; j, k);
10. **if** $(v_k, v_i), (v_k, v_j) \in Q$ **then** *check*-2(i, j; k, i; k, j);
11. **if** $(v_k, v_i), (v_j, v_k) \in Q$ **then** *check*-2(i, j; k, i; j, k);
12. call Δ-*consistency*(i, j, k);

The above algorithm is a general control to check the triangle consistency for all the tuples in R_{ij} (relation for edge $(v_i, v_j) \in Q$) with respect to a certain vertex $v_k \in Q$. In general, we will distinguish between two cases: (1) v_k is incident only to one of the two vertices: v_i or v_j (see line 1), and (2) v_k is incident to both v_i and v_j (see line 7). For case 1, we further have four sub-cases (see lines 2, 3, 4, 5, respectively). For each of them, a subprocedure *check*-1( ) (see Algorithm 2 below) is invoked (see line 6), in which three tasks will be performed:

- make a bit-wise *and* operation over the corresponding relation signatures,
- change the counters of the corresponding vertices according to the result of the bit-wise *and* operation, and
- change the relevant relation signatures themselves.

For case 2, we also distinguish among four sub-cases (see lines 8, 9, 10, 11, respectively). But for each of them, we will call a different subprocedure *check*-2( ) (see Algorithm 4 below) to do the same tasks as *check*-1( ), but in different ways. Finally, in line 12, Δ-*consistency* will be invoked to check the triangle consistency involving v_i, v_j, and v_k.

Algorithm 2: *check*-1(R_1, R_2, a, b)
Input: R_1, R_2, a, b.
Output:
1. $s := R_1.S[a] \wedge R_2.S[b]$;
2. $c := (a \bmod 2) + 1$; $d := (b \bmod 2) + 1$;
3. call *tuple-removing*(s, R_1, R_2, a, c, d);
4. call *tuple-removing*(s, R_2, R_1, b, d, c);

As mentioned above, in *check*-1( ), we will first calculate $s = R_1.S[b] \wedge R_2.S[a]$ (see line 1), where R_1 and R_2 represents two relations corresponding two incident edges in Q. Then, in terms of s, we will make changes respectively with respect to R_1 and R_2 as described above by calling *tuple-removing*( ) given below.

Special attention should also be paid to the modulo operations given in line 2: $y = (x \bmod 2) + 1$, by which $y = 1$ if $x = 2$ and $y = 2$ if $x = 1$.

In the following algorithm, we use $R(l, a)$ to represent the value appearing in the l-th tuple, a-th column of relation R. For a tuple t, $t(i)$ ($i = 1, 2$) stands for its i-th value.

Algorithm 3: *tuple-removing*(s, R, R', a, b, c)
Input: s, R, R', a, b, c.
Output:
1. assume that $R'[c]$ corresponds to v_k in Q;
2. **for** $l = 1$ to $|R|$ **do** {$t :=$ l-th tuple of R;
3. **if** $s[R(l, a)] = 1$ **then** {$\alpha[t, k] :=$ number of tuples t' in R' with $t'(b) = R(l, a)$; append all such t's to $\beta[t]$;}
4. **else** {
5. $L := L \cup \{t\}$; let $t = \langle x, y \rangle$;
6. **if** $y.C_R[b] > 0$ **then** {$y.C_R[b]$ --; if $y.C_R[b] = 0$ then
7. $R.S[b][y] := 0$;}
8. **if** $x.C_R[a] > 0$ **then** {$x.C_R[a]$ --; if $x.C_R[a] = 0$ then
9. $R.S[a][x] := 0$;}}

In *tuple-removing*(s, R, R', a, b, c), we will first eliminate inconsistent tuples from R according to s (lines 2 – 5). Then, for each removed $\langle x, y \rangle$, we will change the counters associated with x, y, respectively (see lines 6 and 8). In particular, if the counter of a vertex becomes 0, the bit for that vertex in the corresponding relation signature will also be modified to 0 (see lines 7 and 9).

Algorithm 4: *check*-2(i, j; r, l; p, q)
Input: i, j, k, l, p, q.
Output:
1. $R := R_{ij}$; $R' := R_{rl}$;
2. **if** $r = i$ **then** {$a := 1$; $b := 1$;} **else** {$a := 2$; $b := 1$;}
3. call *check-1*(R, R', a, b);
4. $R := R_{ij}$; $R' := R_{pq}$;
5. **if** $p = j$ **then** {$a := 1$; $b := 2$;} **else** { $a := 1$; $b := 2$;}
6. call *check-1*(R, R', a, b);
7. $R := R_{rl}$; $R' := R_{pq}$;
8. **if** $r = i \wedge p = j$ **then** {$a :=2$; $b := 2$;}
9. **if** $r = i \wedge p \neq j$ **then** {$a :=1$; $b := 2$;}
10. **if** $r \neq i \wedge p = j$ **then** {$a :=1$; $b := 1$;}
11. **if** $r \neq i \wedge p \neq j$ **then** {$a := 1$; $b := 1$;}
12. call *check-1*(R, R', a, b);

In *check*-2( ), we need to do three bit-wise *and* operations, with each corresponding to a pair of joint edges within a triangle (see lines 1, 4, and 7). Each of them can be simply done by calling *check*-1( ) (see lines 3, 6, and 12). But we should notice the difference between the third case (line 7) and the first two cases (line 1 and line 4). For the third case, we need to distinguish among 4 sub-cases (line lines 8 – 11) while for each of the first two cases only two sub-cases (see lines 2 and 5) need to be handled.

Now, we give the formal description of Δ-*consistency*(i, j, k), which is invoked in line 12 in *tcControl*( ). For simplicity, however, we only show the algorithm for the case (v_j, v_k), $(v_k, v_i) \in Q$. For this, we will check, for each tuple $\langle u', u'' \rangle \in R_{jk}$, whether there exists l with $s[l] = 1$ such that $\langle l, u' \rangle \in R_{ij}$, $(u'', l) \in R_{ki}$, where $s = R_{ij}.S[2] \wedge R_{jk}.S[1]$. For the other three cases (i.e., (v_i, v_k), $(v_j, v_k) \in Q$, (v_i, v_k), $(v_j, v_k) \in Q$, (v_k, v_i), $(v_k, v_j) \in Q$), a similar process can be established for each of them.

Algorithm 5: Δ-*consistency*(i, j, k)
Input: i, j, k.
Output:
1. $s := R_{ij}.S[2] \wedge R_{jk}.S[1]$.
2. **for** $l = 1$ to $|s|$ **do** {
3. **if** $s[l] = 1$ **then**
4. **for** each pair of tuples: $\langle u', l \rangle \in R_{ij}$, $\langle l, u'' \rangle \in R_{jk}$ **do**
5. **if** $\langle u'', u' \rangle \in R_{ki}$ **then** {$t_1 := \langle u', l \rangle$; $t_2 := \langle l, u'' \rangle$; $t_3 := \langle u'', u' \rangle$;
6. add (k, u'') to $\beta[t_1]$; (i, u') to $\beta[t_2]$; (j, l) to $\beta[t_3]$;
7. $\alpha[t_1, k]$ ++; $\alpha[t_2, i]$ ++;$\alpha[t_3, j]$ ++; }}
8. **for** each $t \in R_{ij} \cup R_{ki} \cup R_{ki}$ with $\alpha[t, x] = 0$ **do**
9. {let $t = \langle u, u' \rangle$; $L := L \cup \{t\}$;
10. **if** $t \in R_{ij}$ **then** {remove t from R_{ij}; $R := R_{ij}$;}}
11. **if** $t \in R_{jk}$ **then** remove t from R_{jk}; $R := R_{jk}$;}}
12. **if** $t \in R_{ki}$ **then** {remove t from R_{ki}; $R := R_{ki}$;}
13. $u.C_R[1]$ --; if $u.C_R[1] = 0$, change $R.S[1][u]$ to 0;
14. $u'.C_R[2]$ --; if $u'.C_R[2] = 0$, change $R_R.S[2][u']$ to 0;
15. }

The above algorithm mainly comprises two steps. In the first step (lines 1 – 7), we create $s := R_{ij}.S[2] \wedge R_{jk}.S[1]$, and then scan s bit by bit. For each $s[l] = 1$, we will check, for each pair of tuples:$\langle u', l \rangle \in R_{ij}$ and $\langle l, u'' \rangle \in R_{ij}$, whether $\langle u'', u' \rangle$ appears in R_{ij}. If such a tuple exist, $\alpha[\ ]$ and $\beta[\ ]$ will be accordingly changed (lines 6 – 7). In the second step (lines 8 – 15), for each $\alpha[t, x] = 0$ (for some x) we remove t from the corresponding relation (see lines 10 – 12), and

accordingly change relevant counters and relation signatures (see lines 13 – 14).

Example 4 Applying Δ-*consistency*(2, 3, 1) to the three edges of Q shown in Fig. 1(b) against the δ-transitive closure shown in Fig. 3(b), three relations: R_5, R_6, and R_1 will be loaded into main memory. The following computation will be conducted. (Since R_5, R_6 and R_1 correspond to the three edges in Q, they are also referred to as R_{23}, R_{31} and R_{12}, respectively.)

i) $s := R_{23}.S[2] \wedge R_{31}.S[1] = R_5.S[2] \wedge R_6.S[1] = 111 \wedge 011 = 011$.

ii) $s[2] = 1$. For $t_8 = <2, 2>$ ($<u_5, u_4>$) $\in R_{23} = R_5$ and $t_{12} = <2, 2>$ ($<u_4, u_7>$) $\in R_{31} = R_6$, we will check whether $<2, 2>$ ($<u_7, u_5>$) is in $R_{12} = R_1$. Since $<2, 2>$ ($<u_7, u_5>$) $\notin R_{12}$, lines 6 and 7 will not be executed and therefore $\beta[t_8] = \beta[t_{12}] = \phi$ and $\alpha[t_8, 1] = \alpha[t_{12}, 2] = 0$.

For $t_{11} = <3, 2>$ ($<u_9, u_4>$) $\in R_{23} = R_5$ and $t_{12} = <2, 2>$ ($<u_4, u_7>$) $\in R_{31} = R_6$, we will check whether $<2, 3>$ ($<u_7, u_9>$) is in $R_{12} = R_1$. Since $t_{10} = <2, 3>$ ($<u_7, u_9>$) $\in R_{12}$, lines 6 and 7 will be executed and therefore $\beta[t_{11}] = \{(1, u_7)\}$, $\beta[t_{12}] = \{(2, u_9)\}$, $\beta[t_{10}] = \{(3, u_4)\}$; and $\alpha[t_{11}, 1] = \alpha[t_{12}, 2] = \alpha[t_{10}, 3] = 1$.

iii) $s[3] = 1$. For $t_9 = <3, 3>$ ($<u_9, u_{10}>$) $\in R_{23} = R_5$ and $t_5 = <3, 3>$ ($<u_{10}, u_8>$) $\in R_{31} = R_6$, we will check whether $<3, 3>$ ($<u_8, u_9>$) is in $R_{12} = R_1$. Since $t_7 = <3, 3>$ ($<u_8, u_9>$) $\in R_{12}$, lines 6 and 7 will be executed and therefore $\beta[t_9] = \{(1, u_8)\}$, $\beta[t_5] = \{(2, u_9)\}$, $\beta[t_7] = \{(3, u_{10})\}$; and $\alpha[t_9, 1] = \alpha[t_5, 2] = \alpha[t_7, 3] = 1$.

After the above steps, we must have:

$\alpha[t_5, 2] = 1$	$\beta[t_5] = \{(2, u_9)\}$	$\alpha[t_2, 3] = 0$	$\beta[t_2] = \phi$
$\alpha[t_7, 3] = 1$	$\beta[t_7] = \{(3, u_{10})\}$	$\alpha[t_3, 1] = 0$	$\beta[t_3] = \phi$
$\alpha[t_9, 1] = 1$	$\beta[t_9] = \{(1, u_8)\}$	$\alpha[t_4, 2] = 0$	$\beta[t_4] = \phi$
$\alpha[t_{10}, 3] = 1$	$\beta[t_{10}] = \{(3, u_4)\}$	$\alpha[t_6, 3] = 0$	$\beta[t_6] = \phi$
$\alpha[t_{11}, 1] = 1$	$\beta[t_{11}] = \{(1, u_7)\}$	$\alpha[t_8, 1] = 0$	$\beta[t_8] = \phi$
$\alpha[t_{12}, 2] = 1$	$\beta[t_{12}] = \{(2, u_9)\}$		

Then, all those tuples, whose α-values equal 0, will be first stored in L (see line 9), and then checked to propagate inconsistency before they are finally removed (see lines 10 – 13). Accordingly, the counters of the corresponding vertices and also possibly relation signatures will be changed (see lines 13 – 14).

In this way, the graph shown in Fig. 4(a) will be reduced to the graph shown in Fig. 4(b).

Based on *tcControl*( ), a general algorithm for the relation filtering can be easily designed. It works in two phases.

Algorithm 5: *rFiltering*(Q, R_{ij}'s)
Input: Q.
Output:
1. **for** each $(v_i, v_j) \in Q$ **do** {
2. **for** each $k \neq i, j$ **do** {
3. call *tcControl*(R_{ij}, v_k); }}
4. **while** L is not empty **do** {
5. choose t from L and remove t from L;
6. assume that $t = <x, y> \in R_{ij}$;
7. **for** each $(k, z) \in \beta[t]$ **do** {
8. assume that $t_1 = <y, z>$ and $t_2 = <z, x>$;
9. $\alpha[t_1, i]$ --; remove $(i, x) \in \beta[t_1]$;
10. **if** $\alpha[t_1, i] = 0$ **then** $L := L \cup \{t_1\}$;
11. $\alpha[t_2, j]$ --; remove $(j, y) \in \beta[t_2]$;
12. **if** $\alpha[t_2, j] = 0$ **then** $L := L \cup \{t_2\}$;
13. }

In the first phase (lines 1 –3), we check the triangle consistency for each edge in Q. In the second phase (lines 4 – 12), we propagate inconsistency among all those triangles which share one edge. To see this, we consider a larger query shown in Fig. 5(a). Obviously, by checking the consistency with respect to triangle Δ_{143} tuple $t_5 \in R_{31}$ (= R_6 shown in Fig. 3(c)) will be removed. Then, $(2, u_9)$ in $\beta[t_5]$ will be checked (see line 9), leading to the elimination of $t_7 = <u_8, u_9>$ from R_{12} (= R_1) and $t_9 = <u_9, u_{10}>$ from R_{23} (= R_5) see lines 11 – 14). For this query, only five edges $<u_7, u_9>$, $<u_9, u_4>$, $<u_4, u_7>$, $<u_7, u_1>$, and $<u_1, u_4>$ (in Fig. 2(b)) will survive the relation filtering. (See Fig. 5(b) for illustration.

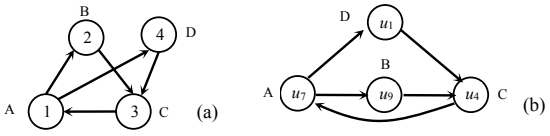


Fig. 5: A larger query and filtering results

V. CONCLUSION

In this paper, we have proposed a novel algorithm for pattern match problem over large data graphs G. The main idea behind it is the concept of δ-transitive closures and a relation filtering method based on the concept of triangle consistency. As part of an index, a δ-transitive closure G^δ of G will be constructed off-line for a certain δ-value. Then, for a query with $\delta' \leq \delta$, the relevant relations in G^δ can be directly loaded into main memory, instead of constructing them on-the-fly from some auxiliary data structures such as 2-hops and LLR embedding vectors. Especially, the useless tuples in the relations will be filtered before they take part in joins, which enables us to achieve high performance. In addition, the bit mapping technique has been integrated into the relation filtering to expedite the working process. Also, extensive experiments have be conducted, which shows that our method is promising.

REFERENCES

[1] H. Jiang, H. Wang, P. S. Yu, and S. Zhou, "Gstring: A novel approach for efficient search in graph databases," *Proc. 23rd Int. Conf. ICDE*, pp. 566–575, IEEE, 2007.

[2] J. Cheng and J. X. Yu, "On-line exact shortest distance query processing," *Proc. 12th Int. Conf. Extending Database Technol. Adv. Database Technol*, *EDBT 09*, pp. 481-492, 2009.

[3] Y. Chen and Y. Chen, "An efficient algorithm for answering graph reachability queries," *Proc. ICDE*, pp. 893–902, 2008.

[4] Y. Chen and Y.B. Chen, Decomposing DAGs into spanning trees: A new way to compress transitive closures, in Proc. 27th Int. Conf. on Data Engineering (ICDE 2011), IEEE, April 2011, pp. 1007-1018.

[5] J. Cheng, J. X. Yu, B. Ding, P. S. Yu, and H. Wang, "Fast graph pattern matching," *Proc. Int. Conf. ICDE.*, pp. 913–922, 2008.

[6] J. E. Hopcroft and J. Wong, "Linear time algorithm for isomorphism of planar graphs," in *Proc. 6th Annual ACM Symp. Theory of Computing*, pp. 172–184, 1974.

[7] E. M. Luks, "Isomorphism of graphs of bounded valence graphs can be tested in polynomial time," *J. Comput. Syst. Sci.* **25** (1982) 42–65.