Answering Label-Constraint Reachability in Large Graphs *

Kun Xu
Peking University
Beijing, China
xukun@icst.pku.edu.cn

Lei Zou †
Peking University
Beijing, China
zoulei@cse.ust.hk

Jeffrey Xu Yu
The Chinese Univ. of Hong Kong
Hong Kong, China
yu@se.cuhk.edu.hk

Lei Chen
Hong Kong Univ. of Sci. & Tech.
Hong Kong, China
leichen@cse.ust.hk

Yanghua Xiao
Fudan University
Shanghai, China
shawyh@fudan.edu.cn

Dongyan Zhao
Peking University
Beijing, China
zdy@icst.pku.edu.cn

ABSTRACT

In this paper, we study a variant of reachability queries, called *l*abel-*c*onstraint *r*eachability (LCR) queries, specifically, given a label set S and two vertices u_1 and u_2 in a large directed graph G, we verify whether there exists a path from u_1 to u_2 under label constraint S. Like traditional reachability queries, LCR queries are very useful, such as pathway finding in biological networks, inferring over RDF (*r*esource *d*escription *f*ramework) graphs, relationship finding in social networks. However, LCR queries are much more complicated than their traditional counterpart. Several techniques are proposed in this paper to minimize the search space in computing path-label transitive closure. Furthermore, we demonstrate the superiority of our method by extensive experiments.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; H.2.8 [**Database Management**]: Database Applications—*Graph Database*

General Terms

Algorithm

†corresponding author: Lei Zou, zoulei@icst.pku.edu.cn

*Lei Zou and Dongyan Zhao were supported by NSFC under Grant No.61003009 and RFDP under Grant No. 20100001120029. Jeffrey Xu Yu was supported by RGC of the Hong Kong SAR under Grant No. 419008 and 419109. Lei Chen's research is partially supported by HKUST SSRI11EG01 and NSFC No.60003074. Yanghua Xiao was supported by the NSFC under Grant No.61003001 and RFDP under Grant No. 20100071120032.

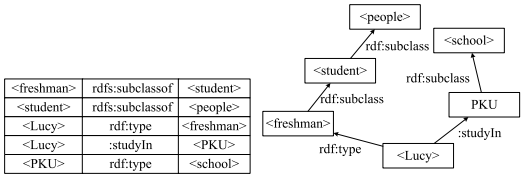
CIKM'11, October 24–28, 2011, Glasgow, Scotland, UK.

1. INTRODUCTION

The growing popularity of graph databases has generated many interesting data management problems. One important type of queries over graphs are *reachability queries* [1, 2, 3, 4, 5, 6]. Specifically, given two vertices u_1 and u_2 in a directed graph G, we want to verify whether there exists a directed path from u_1 to u_2. There are many applications of reachability queries, such as, pathway finding in biological networks [7], inferring over RDF (*r*esource *d*escription *f*ramework) graphs [8], relationship finding in social networks [9]. There are two extreme solutions to answer reachability queries. One approach is to materialize the full transitive closures of G, enabling one to answer reachability queries efficiently. In the other extreme, one can perform DFS (*d*epth-*f*irst *s*earch) or BFS (*b*reath-*f*irst *s*earch) over graph G until reaching the target vertex, or the search process cannot be continued. Obviously, these two methods cannot work in a large graph G, since the former needs $O(V^2)$ space to store the transitive closure (large index space cost), and the latter needs $O(V)$ time in answering reachability queries (slow query response time). The key issue in reachability queries is how to find a good trade-off between the two basic solutions. Therefore, many algorithms have been proposed, such as 2-hop [1, 10, 11], GRIPP [2], path-cover [5], tree-cover [2, 6], pathtree [3] and 3-hop [4], to address this issue.

In many real applications, edge labels are utilized to denote different relationships between two vertices. For example, edge labels in RDF graphs denote different properties. We can also use edge labels to define different relationships in social networks. In this paper, we study a variant of reachability queries, called *l*abel-*c*onstraint *r*eachability (LCR) queries, which are originally proposed in [12]. Conceptually, LCR query verifies some specified type of relationships between two vertices. Here, we give a motivation example to demonstrate the usefulness of LCR queries.

Let us consider an inference example in RDF Schema (RDFS for short) dataset in Figure 1. Assume that we want to verify whether $\langle freshman \rangle$ is a subclass of $\langle people \rangle$. The traditional database system can simply answer *no*, since there exists no triple ($\langle freshman \rangle$, *rdfs:subclass*, $\langle people \rangle$). However, due to RDFS semantics, "rdfs:subclass" is a tran-



(a) RDF Dataset (b) RDF Graph

Figure 1: Inference VS. LCR Query

sitive property. Therefore, given two triples ($\langle freshman\rangle$, *rdfs:subclass*, $\langle student\rangle$) and ($\langle student\rangle$, *rdfs:subclass*, $\langle people\rangle$), we can infer that ($\langle freshman\rangle$, *rdfs: subclass*, $\langle$ *people*$\rangle$). Therefore, the correct answer should be *yes*. This kind of queries are called *inference queries*. Obviously, it is prohibitive to materialize all inferred facts, according to RDFS reasoning rules, in a very large RDF dataset. Actually, some inference queries can be reduced into LCR queries over RDF graphs. For example, the above example can be transformed into the following LCR query: verifying whether there exists a directed path from entry $\langle freshman\rangle$ to $\langle people\rangle$ in the RDF graph, where all edge labels along the path are "rdfs:subclass".

Although LCR queries are quite useful, it is non-trivial to answer LCR queries over a large directed graph. Traditional indexing methods can only verify reachability without considering how the connection is made between two vertices [12]. Furthermore, the traditional transitive closure does not contain path labels.

In order to address LCR queries efficiently, we make the following contributions in this work:

1) For LCR queries, we propose a method to transform an edge-labeled directed graph into an *augmented DAG* by representing the maximal strongly connected components as *bipartite graphs*. Then, based on the augmented DAG, we propose a method to compute transitive closure efficiently.

2) We re-define the "distance" of a path and propose a Dijkstra-like algorithm to compute single-source path-label transitive closure. We prove the optimality of our algorithm.

3) Extensive experiments confirm that our method is faster than the existing ones by orders of magnitude. For example, given a random network satisfying ER model with 100K vertices and 150K edges, the method in [12] consumes 277 hours for index building. However, given the same graph, our method only needs 0.5 hour for indexing building. Furthermore, our method can work well in a very large RDF graph (Yago dataset) having more than 2 million vertices and 6 million edges and 97 edge labels.

2. BACKGROUND

2.1 Problem Definition

DEFINITION 2.1. *A* directed edge-labeled graph *G is denoted as $G = \{V, E, \sum, \lambda\}$, where (1) V is a set of vertices, and (2) $E \subseteq V \times V$ is a set of directed edges, and (3) $\sum$ is a set of edge labels, and (4) the labeling function λ defines the mapping $E \rightarrow \sum$.*

Given a path p from u_1 to u_2 in graph G, the path-label *of p is denoted as $L(p) = \bigcup_{e \in p} \lambda(e)$, where $\lambda(e)$ denotes e's edge label.*

Given a graph G in Figure 2, the numbers inside vertices are *vertex IDs* that we introduce to simplify description of the graph; and the letters beside edges are *edge labels*. Considering path $p_1 = (1, 2, 5)$, the path-label of p_1 is $L(p_1) = \{ac\}$.

DEFINITION 2.2. *Given two vertices u_1 and u_2 in graph G and a label constraint (set) $S = \{l_1, ..., l_n\}$, we say that u_1 can reach u_2 under label constraint S (denoted as $u_1 \xrightarrow{S} u_2$) if and only if there exists a path p from u_1 to u_2 and $L(p) \subseteq S$.*

DEFINITION 2.3. (Problem Definition) *Given two vertices u_1 and u_2 in graph G and a label set $S = \{l_1, ..., l_n\}$, a* label-constraint reachability *(LCR) query verifies whether u_1 can reach u_2 under the label constraint S, denoted as $LCR(u_1, u_2, S, G)$.*

For example, given two vertices 1 and 6 in graph G in Figure 2 and label constraint $S = \{ac\}$, it is easy to know that 1 can reach 6 under label constraint S, i.e., $LCR(1, 6, S, G) =$ true, since there exists path $p_1 = \{1, 2, 5, 6\}$, where $L(p_1) \subseteq S$. If $S = \{bc\}$, query $LCR(1, 6, S, G)$ returns false.

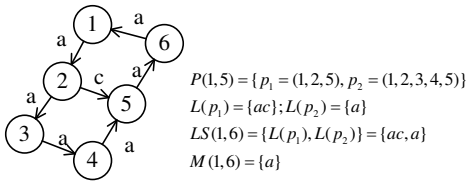


Figure 2: A Running Example

DEFINITION 2.4. *Given two vertices u_1 and u_2 in graph G, $P(u_1, u_2)$ denotes all paths from u_1 to u_2. The* path-label set *from u_1 to u_2 is defined as $LS(u_1, u_2) = \{L(p) | p \in P(u_1, u_2)\}$.*

Consider two paths $(1, 2, 3, 4, 5, 6)$ and $(1, 2, 5, 6)$ in $P(1, 6)$, where $L(1, 2, 3, 4, 5, 6) \subseteq L(1, 2, 5, 6)$. Obviously, if path $(1, 2, 5, 6)$ can satisfy some label constraint S, path $(1, 2, 3, 4, 5, 6)$ can also satisfy S. Therefore, path $(1, 2, 5, 6)$ is *redundant* (Definition 2.5) for any LCR query.

DEFINITION 2.5. *Considering two paths p and p' from vertex u_1 to u_2, respectively, if $L(p) \subseteq L(p')$, we say $L(p)$* covers *$L(p')$. In this case, p' is a* redundant *path, and $L(p')$ is also* redundant *in the path-label set $LS(u_1, u_2)$.*

DEFINITION 2.6. *The* minimal path-label set *from u_1 to u_2 in graph G is defined as $M_G(u_1, u_2)$, where 1)$M_G(u_1, u_2) \subseteq LS(u_1, u_2)$; and 2) there exists no redundant path-label in $M_G(u_1, u_2)$; and 3) path-labels in $M_G(u_1, u_2)$ cover all path labels in $LS(u_1, u_2)$.*

DEFINITION 2.7. *Given two vertices u_1 and u_2 and a label constraint (set) S, we say $M_G(u_1, u_2)$* covers *S if and only if there exists a path p from u_1 to u_2, where $S \supseteq L(p)$ and $L(p) \in M_G(u_1, u_2)$.*

DEFINITION 2.8. *Given a graph G,* path-label transitive closure *is a matrix $M_G = [M_G(u_1, u_2)]_{|V(G)| \times |V(G)|}$, where $u_1, u_2 \in V(G)$, and a* single-source path-label transitive closure *is a vector $M_G(u, -) = [M_G(u, u_i)]_{1 \times |V(G)|}$, where $u_i \in V(G)$.*

When the context is clear, we also say *transitive closure* instead of *path-label transitive closure* for short. Furthermore, for ease of presentation, we borrow two operator definitions (*Prune* and $\odot$) from reference [12].

Prune$(\cdot)$ is defined as $Prune(LS(u_1, u_2)) = M_G(u_1, u_2)$, which means removing all redundant paths. In Figure 2, $Prune(LS(1,6) = \{a, ac\}) = M_G(1, 6) = \{a\}$, since $a \subseteq ab$.

$(\cdot) \odot (\cdot)$ is defined as follows: $\lambda(\overrightarrow{u_1u_2}) \odot M_G(u_2, u_3) = \{\lambda(\overrightarrow{u_1u_2}) \bigcup L(p_1), \lambda(\overrightarrow{u_1u_2}) \bigcup L(p_2), ...\}$, where $L(p_i) \in M_G(u_2, u_3)$ and $\lambda(\overrightarrow{u_1u_2})$ denotes the label of edge $\overrightarrow{u_1u_2}$.

It is easy to prove that $M_G(u_1, u_3) = Prune(\bigcup_{u' \in V(G)} \{\lambda(\overrightarrow{u_1u'}) \odot M_G(u', u_3)\})$. Analogously, we also define $\lambda(\overrightarrow{u_1u_2}) \odot M_G(u_2, -)$.

An extreme approach to answering LCR queries is to materialize transitive closure M_G. At run time, given a query $LCR(u_1, u_2, S, G)$, LCR queries can be answered by simply checking $M_G(u_1, u_2)$. However, computing M_G is much more complicated than traditional transitive closure. Before the formal discussion, we introduce the following theorem, which forms the basis of our algorithm and performance analysis.

THEOREM 2.1. *(Apriori Property) Given one path p, if one of its subpaths is redundant, p must be redundant.*

2.2 Existing Approaches

LCR queries are proposed in [12]. Generally speaking, the method in [12] employs a spanning tree T and a partial transitive closure NT to compress the full transitive closure. Specifically, a spanning tree T is found in the graph G. Based on T, we partition all pairwise paths into three categories P_n and P_s and P_e. All paths in P_n contains all pairwise paths whose starting edges and end edges are both non-tree edges. All paths in P_s (and P_e) contains all pairwise paths whose starting (and ending) edges are tree-edges. In Figure 3, (4,5,6,1) is a path in P_n, since $\overrightarrow{4,5}$ and $\overrightarrow{6,1}$ are non-tree edges. $NT(u, v)$ contains all path labels between u and v in P_n. We can re-construct $M_G(u, v)$ by Equation 1. Therefore, we can re-construct the full transitive closure by the spanning tree T and partial transitive closure $NT = \{NT(u, v), u, v \in V(G)\}$.

$$M_G(u,v) = \{\{L(P_T(u,u'))\} \odot NT(u',v') \odot \{L(P_T(v',v))\} | u' \in Succ(u) \text{and} v' \in \Pr ed(v)\} \quad (1)$$

where u' is reachable from u in the spanning tree T and $L(P_T(u, u'))$ denotes the corresponding path label in T; and v' can reach v in the spanning tree T and $L(P_T(v', v))$ denotes the corresponding path label in T.

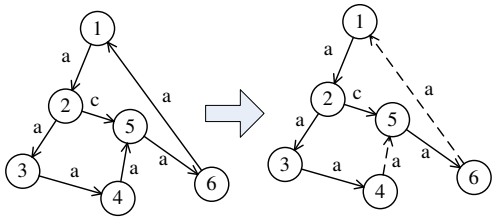


Figure 3: An Example of Tree-Cover

Obviously, different spanning trees will lead to different NT. In order to minimize the size of NT, authors introduce "weight" $w(e)$ for each edge e, where $w(e)$ reflects that if e is included in a spanning tree, the number of path-labels that can be removed from NT. Therefore, they propose to use the maximal spanning tree in G. However, it is quite expensive to assign exact edge weights $w(e)$. Thus, they propose a sampling method. For each sampling seed (vertex), they compute single-source transitive closure, based on which, they propose some heuristic methods to define edge weights.

However, there are two limitations of their method in [12]. First, similar with the counterpart methods in traditional reachability queries [6, 3], a single spanning tree cannot compress the transitive closure greatly, especially in dense graphs. Consequently, NT may be very large. Second, in order to find the optimal spanning tree T, authors propose an algorithm to compute single-source transitive closure for each sampling seed (vertex). However, the search space in their algorithm is not minimal, i.e., containing a large number of redundant paths, which affect the performance greatly. The above two problems (large index size and expensive index building process) affect the scalability of the method in [12].

Since computing single-source transitive closure is also a building block in our method, we argue that our method is optimal in terms of search space. In order to understand the superiority of our method, in the full version of this paper [13], we analyze the algorithm in [12]. Due to space limit, the details are omitted in this paper.

In [14], Fan et al. add regular expressions to graph reachability queries. Specifically, given two vertices u_1 and u_2, the method in [14] verifies whether there exists a directed path P where all edge labels along the path satisfy the specified regular expression. Obviously, LCR query is a special case of the problem in [14]. Fan et al. propose a bi-directional BFS algorithm at runtime. We can utilize the method in [14] for LCR queries. Given two vertices u_1 and u_2 over a large graph G and a label set S, two sets are maintained for u_1 and u_2, respectively. Each set records the vertices that are reachable from (resp. to) u_1 (resp. u_2) only via edges of labels in S. We expand the smaller set at a time until either the two sets intersect, or they cannot be further expanded (i.e., unreachable). The problem of this method lies in its large search space. The search strategy in [14] is different from traditional BFS algorithm, since one vertex may be visited multiple times in bi-directional BFS algorithm [14].

3. COMPUTING TRANSITIVE CLOSURE

As mentioned earlier, compared with traditional transitive closure, it is much more challenging to compute *path-label transitive closure* (Definition 2.8). This section focuses computing path-label transitive closure efficiently. We first propose a Dijkstra-like algorithm to compute single-source transitive closure efficiently (Section 3.1). Obviously, it is very expensive to iterate single-source transitive closure computation from each vertex in G to compute M_G. In order to address this issue, we propose *augmented DAG* (aDAG for short) by representing all strongly connected components as *bipartite graphs*. The aDAG-based solution is discussed in Section 3.2.

3.1 Single-Source Transitive Closure

This subsection discusses how to compute single-source transitive closure efficiently, since it is a building block in our aDAG-based solution. As discussed in Section 2.2, the method in [12] is not optimal. The key problem is that some redundant paths are visited before their corresponding non-redundant paths. In order to address this issue, we propose a Dijkstra-like algorithm. As we know, in each step of Dijkstra's algorithm, we always access one un-visited vertex that has the minimal distance from the origin vertex. In our algorithm, we redefine "distance". A *distance* of a path p is defined as *the number of distinct edge labels* in p instead of

the sum of edge weights. Then, according to the distance definition, we adopt the Dijkstra-like algorithm to compute single-source transitive closure. This algorithm can guarantee that all redundant paths must be visited after their corresponding non-redundant paths. Therefore, all redundant paths can be pruned from the search space (Theorem 3.1).

Heap H	Path Set RS
Step 1. [{a}, (1,2) , 2];	[{a}, (1,2), 2];
Step 2. [{a}, (1,2, 3), 3]; [{ac}, (1,2,5), 5];	[{a}, (1,2,3), 3];
Step 3. [{a}, (1,2, 3, 4), 4]; [{ac}, (1,2,5), 5];	[{a}, (1, 2,3,4), 4];
Step 4. [{a}, (1,2, 3, 4,5), 5]; ~~[{ac}, (1,2,5), 5];~~	[{a}, (1, 2,3,4,5), 5];
Step 5. [{a}, (1,2,3,4,5,6), 6]	[{a}, (1,2,3,4,5,6), 6]

Figure 4: Algorithm Process

Given a graph G in Figure 2, Figure 4 demonstrates how to compute $M_G(1,-)$ from vertex 1 in our algorithm (i.e., Algorithm 1). Initially, we set vertex 1 as the source. All vertex 1's neighbors are put into the heap H. Each neighbor is denoted as a *neighbor triple* $[L(p), p, d]$, where d denotes the neighbor's ID, p specifies one path from source s to d, and $L(p)$ is the path-label set of p. All neighbor triples are ranked according to the *total order* defined in Definition 3.1. Since $[\{a\},(1,2),2]$ is the heap head (see Figure 4), it is moved to path set RS. When we move the heap head T_1 into path set RS, we check whether T_1 is *covered* (Definition 3.2) by some neighbor triple T_2 in RS (Line 5 in Algorithm 1). If so, we ignore T_1 (Lines 5-6); otherwise, we insert T_1 into RS (Lines 7-8).

DEFINITION 3.1. *Given two neighbor triples $T_1 = [L(p_1), p_1, d_1]$ and $T_2 = [L(p_2), p_2, d_2]$ in the heap H, $T_1 \leq T_2$ if and only if 1) $|L(p_1)| < |L(p_2)|$; or 2) $|L(p_1)| = |L(p_2)|$, the orders of T_1 and T_2 are arbitrarily defined.*

DEFINITION 3.2. *Given one neighbor triple $T_1 = [L(p_1), p_1, d_1]$, T_1 is* redundant *if and only if there exists another neighbor triple $T_2 = [L(p_2), p_2, d_2]$, where $L(p_1) \supseteq L(p_2)$ and $d_1 = d_2$. In this case, we say that T_1 is* covered *by T_2.*

DEFINITION 3.3. *Given two neighbor triple $T_1 = [L(p_1), p_1, d_1]$ and $T_2 = [L(p_2), p_2, d_2]$, if p_1 is a parent path (or a child path) of p_2, we say that T_1 (or T_2) is a* parent neighbor triple *(or a* child neighbor triple*) of T_2 (or T_1).*

Then, we put all child neighbor triples (Definition 3.3) of $[\{a\},(1,2),2]$ into heap H. Considering one neighbor of vertex 2, such as vertex 3, we put neighbor triple $[\{a\} \cup L(\overrightarrow{2,3}) = \{a\},(1,2,3),3]$ into H, where $(1,2,3)$ is $(1,2)$'s child path. Analogously, we put $[\{a\} \cup L(\overrightarrow{2,5}) = \{ac\},(1,2,5),5]$ into H. When we insert some neighbor triple $T'[L(p'),p',d']$ into H, we first check whether p' is a non-simple path. If so, we ignore T' (Lines 10-11). Furthermore, we also check whether there exists another triple T'' that has existed in H and T' is *covered* by T'', or T' covers T'' (Lines 12-15). If T' is *covered* by T'', we ignore T'; otherwise, T' is inserted into H. If T' covers some triple T'' in H, T'' is deleted from H. At Step 2, the heap head is $[\{a\},(1,3),3]$, which is moved to path set RS.

Iteratively, we put all child neighbor triples of $[\{a\},(1,3),3]$ into heap H. At Step 4, we find that $[\{ac\},(1,2,5),5]$ is covered by $[\{a\},(1,2,3,4,5),5]$. Therefore, we remove $[\{ac\},(1,2,5),5]$ from H. Figure 4 illustrates the whole process. All paths and path-labels in RS are non-redundant. According to RS, it is straightforward to obtain $M_G(1,-)$. Note that, our algorithm stops the infection from the redundant path to its child paths (Theorem 3.1). For example, path $(1,2,5,6)$ is pruned from search space in our algorithm.

Algorithm 1 Single-Source Transitive Closure Computation

Require: Input: A graph G and a vertex u in G;
Output: A compressed path tree C-$PT(u)$.
1: Set u as the source. Set answer set $RS=\phi$ and heap $H=\phi$.
2: Put all neighbor triples of u into H.
3: **while** $H \neq \phi$ **do**
4: Let $T_1 = [L(p_1), p_1, d]$ to denote the head in H.
5: **if** T_1 is covered by some neighbor triple T_2 in RS **then**
6: Delete T_1 from H
7: **else**
8: Move T_1 into RS
9: **for** each child neighbor triple $T'[L(p'),p',d']$ of T_1 **do**
10: **if** p' is a non-simple path **then**
11: continue
12: **if** T' is *not* covered by some neighbor triple T'' in H **then**
13: Insert T' into H
14: **if** T' covers some neighbor triple T'' in H **then**
15: Delete T'' from H
16: According to paths in RS to build C-$PT(u)$.

THEOREM 3.1. *Given a vertex u in graph G, the following statements about Algorithm 1 hold:*

1. *(correctness) Any non-redundant path beginning from vertex u can be found in path set RS in Algorithm 1.*
2. *(optimality) Given any redundant path p', if one of p''s parent paths is also redundant, p' is pruned from the search space in Algorithm 1.*

3.2 Computing Transitive Closures

Given a graph G, a straightforward method to compute transitive closure M_G is to repeat Algorithm 1 from each vertex in G. Obviously, it is inefficient to do that. Intuitively, given two adjacent vertices u_1 and u_2, most computations in Algorithm 1 for u_1 and u_2 are the same to each other. Therefore, an efficient algorithm should employ the property. Usually, a directed graph G is transformed into a DAG by coalescing each strongly connected component into a single vertex to compute transitive closure efficiently. However, this method cannot be used for LCR queries, since it may miss some edge labels. Instead, we propose an *augmented DAG D* by representing all strongly connected components as bipartite graphs. Then, we can compute single-source transitive closure $M(u,-)$ according to the reverse order of D. During the computation, $M(u,-)$ is always transmitted to its parent vertices in D. In this way, redundant computation can be avoided.

Specifically, we first identify all maximal connected components in graph G, denoted as $C_1,...,C_m$. For each C_i $(i=1,...,m)$, we compute local transitive closure in C_i (denoted as M_{C_i}) by iterating Algorithm 1 for each vertex in C_i. For example, given a graph G in Figure 5(a), one maximal connected component C_1 is identified in G. We compute M_{C_1} for C_1. Then, we represent a maximal connected component C_i as a bipartite graph $B_i = (V_{i_1}, V_{i_2})$, where

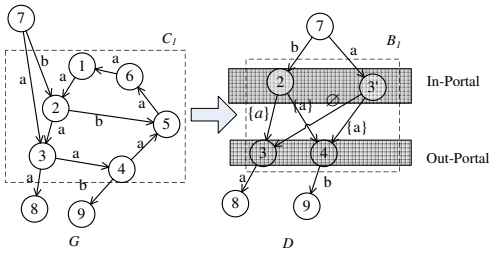


Figure 5: Augmented DAG

V_{i_1} contains all *in-portal* vertices in C_i and V_{i_2} contains all *out-portal* vertices in C_i. A vertex u in C_i is called as an in-portal if and only if it has at least one incoming edge from vertices out of C_i. A vertex u in C_i is called as an out-portal if and only if it has at least one outgoing edge to vertices out of C_i. If one vertex u is both an in-portal and an out-portal, it has two instances u and u' that occur in V_{i_1} and V_{i_2}, respectively. For any two vertices $u_1 \in V_{i_1}$ and $u_2 \in V_{i_2}$, we introduce a directed edge u_1 to u_2, whose edge label is $M_{C_i}(u_1, u_2)$. For example, a bipartite graph B_1 corresponding to C_1 is given in Figure 5(b). Finally, we replace all maximal connected components C_i by the corresponding bipartite graph B_i. In this way, we can get a graph D, as shown in Figure 5(b). We can prove that D must be a DAG. In this paper, we call it *augmented DAG* (aDAG for short). Note that, given a vertex u in C_i, if $u \notin D$, u is called an *intra vertex*, such as vertices 1,5 and 6 in Figure 5(a).

Algorithm 2 aDAG-Based Compute Local Transitive Closure

Require: Input: A graph G and its corresponding aDAG D;
Output: M_G.
1: Identify all maximal connected component C_i.
2: Employ Algorithm 1 to compute local transitive closure M_{C_i} for each C_i.
3: Build aDAG D by replacing C_i by the bipartite graphs B_i.
4: Perform Topological Sorting Over D. Set $M_G(u,-) = \{\phi, ..., \phi\}$ for each vertex u.
5: **for** each vertex u according to the reverse topological order **do**
6: **for** each child c_i of u **do**
7: $M_G(u,-) = Prune(M_G(u,-) \bigcup Prune(\lambda(\overrightarrow{uc_i}) \odot M_G(c_i,-)))$
8: **for** each maximal connect component C_i **do**
9: **for** each intra-vertex $u \in C_i$ **do**
10: $M_G(u,-) = M_{C_i}(u,-)$
11: **for** each out-portal u_i in V_{2_i} **do**
12: $M_G(u,-) = Prune(M_G(u,-) \bigcup Prune(M_{C_i}(u,u_i) \odot M_G(u_i,-)))$
13: **for** each vertex $u' \notin C_i$ **do**
14: **for** each intra-vertex $u \in C_i$ **do**
15: **for** each in-portal u_i in V_{1_i} **do**
16: $M_G(u',u) = Prune(M_G(u',u) \bigcup Prune(M_G(u',u_i) \odot M_{C_i}(u_i,u)))$

Algorithm 2 lists the pseudo code to compute transitive closure for aDAG D. First, we perform topological sorting over D. Initially, for all vertices u in G, we set $M(u,-) = \{\phi, ..., \phi\}$. Then, we process each vertex according to the reverse topological sort. If a vertex u has n children c_i, for each c_i, we update $M_G(u,-) = Prune(M_G(u,-) \bigcup Prune(\lambda(\overrightarrow{uc_i}) \odot M_G(c_i,-)))$ iteratively. In this way, we can obtain $M_G(u,-)$ for each vertex u in aDAG D.

Now, we need to consider "intra vertices" in each cluster C_i, such as vertices 4 and 7 in Figure 5. Given an intra vertex u in C_i, we initialize $M_G(u,-) = M_{C_i}(u,-)$. Then, for each out-portal u_i in V_{2_i}, we update $M_G(u,-) = Prune(M_G(u,-) \bigcup Prune(M_{C_i}(u,u_i) \odot M_G(u_i,-)))$ iteratively.

Consider any one vertex $u' \notin C_i$. Given an intra vertex u in C_i, we compute $M_G(u',u)$ as follows: Initially, we set $M_G(u',u) = \phi$. For each in-portal u_i in V_{1_i}, we update $M_G(u',u) = Prune(M_G(u',u) \bigcup Prune(M_G(u',u_i) \odot M_{C_i}(u_i,u)))$ iteratively.

As we know, 2-hop labeling technique is proposed to compress traditional transitive closure [1, 10]. We also extend the labeling technique to compress the path-label transitive closure.

DEFINITION 3.4. *A 2-label-hop coding over a graph G assigns to each vertex u ($\in V(G)$) a code $C(u) = (C_{in}(u), C_{out}(u))$, where the entries in $C_{in}(u)$ and $C_{out}(u)$ are in form of $\{(w, M_G(w,u)\}$ and $\{(w, M_G(u,w)\}$, respectively.*

DEFINITION 3.5. *A 2-label-hop coding over a graph G is called* complete *if and only if Equation 2 holds.*

$$\begin{array}{l} \forall u_1, u_2 \in V(G), \forall S \in \sum^+ \\ u_1 \xrightarrow{S} u_2 \Leftrightarrow \exists w, w \in C_{out}(u_1) \wedge w \in C_{in}(u_2) \\ \wedge(\exists L(p_1), L(p_1) \in M_G(u_1,w) \wedge S \supseteq L(p_1)) \\ \wedge(\exists L(p_2), L(p_2) \in M_G(w,u_2) \wedge S \supseteq L(p_2)) \end{array} \quad (2)$$

where $L(p_1)$ denotes one path label-set from u_1 to w, and $L(p_2)$ denotes one path label-set from w to u_2.

4. EXPERIMENTS

In this section, we evaluate our methods over both random networks and real datasets, and compare them with the existing solution the sampling-tree method in [12]. Specifically, we experimentally study the performance of three approaches: 1) the *sampling-tree method* proposed in [12]; 2) we compute path-label transitive closure method by Algorithm 2 and compress it by 2-label-hop technique. Then, based on 2-label-hop codes, we can answer LCR queries. This method is called *transitive closure method*; 3) the *bi-directional BFS* proposed in [14]. Our methods are implemented using C++, and our experiments are conducted on a P4 3.0GHz machine with 2G RAM running Ubuntu Linux.

4.1 Datasets

There are two types of synthetic datasets to be used in our experiments: Erdos Renyi Model (ER) and Scale-Free Model (SF). ER is a classical random graph model. It defines a random graph as $|V|$ vertices connected by $|E|$ edges, chosen randomly from the $|V|(|V|-1)$ possible edges. In our experiments, we vary the density $\frac{|E|}{|V|}$ from 1.5 to 5.0, and vary $|V|$ from 1K to 10K. SF defines a random network with $|V|$ vertices satisfying power-law distribution in vertex degrees. In our implementations, we use the graph generator *gengraphwin* (http://fabien.viger.free.fr/liafa/generation/) to generate a large graph G satisfying power-law distribution. Usually, the power-law distribution parameter γ is between 2.0 and 3.0 to simulate real complex networks [15]. Thus, default value of parameter γ is set to 2.5 in this work. In order to study the scalability, we also vary $|V|$ in SF networks from 1K to 10K. The number of edge labels ($|\Sigma|$) is 20. The distribution of labels is generated according to uniform distribution.

Table 1: Performance VS. $|V|$ in ER Graphs

|V|	2-Label-Hop Codes			Sampling-Tree method			Bi-directional BFS (Memory-based)
density = 1.5	IT	IS	QT	IT	IS	QT	QT
	sec.	KB	ms.	sec.	KB	ms.	ms.
1K	15	415	0.01	113	13725	0.05	0.01
2K	23	1396	0.01	493	51020	0.09	0.02
4K	217	4920	0.02	3680	78920	0.10	0.02
6K	623	9000	0.03	5689	92890	0.15	0.03
8K	964	13200	0.03	9290	100890	0.18	0.04
10K	5065	33000	0.04	100560	123450	0.29	0.05

Table 2: Performance VS. Graph Density in ER Graphs

density	2-Label-Hop Codes			Sampling-Tree method			Bi-directional BFS (Memory-
|V|=10K	IT	IS	QT	IT	IS	QT	QT
	sec.	MB	ms.	sec.	MB	ms.	ms.
2	6860	26.3	0.08	123890	95.6	0.31	0.05
3	11112	102.3	0.09	378563	232.7	0.53	0.09
4	25347	160	0.12	F	F	F	0.15
5	33169	186	0.23	F	F	F	0.18

Note: F denotes "fail due to memory crash" or "cannot finish index building in 48 hours"

We also employ two real graph datasets (Yeast, Small-Yago) in our experiments, which are provided by authors in [12]. More details about the two datasets and more experiments are given in the full version of this paper [13].

4.2 Performance of Transitive Closure Method

In this section, we use Algorithm 2 to compute transitive closure for a graph G. Then, we use 2-label-hop coding technique to compress the transitive closure. We report index construction time (IT), index size (IS) and average query response time (QT) for the experiments on the synthetic datasets. Note that, the default query constraint size ($|S|$) is $30\% \times |\sum| = 6$. Furthermore, we also compare our method with the sampling tree method. Note that, in the following experiments, we always randomly generate 10000 queries to evaluate query performance. QT is reported as the average response time for one query. In these experiments, we evaluate the performance with regard to graph size, graph density and label constraint size $|S|$. Furthermore, we also test the performance of bi-directional BFS in Table 1. Since bi-directional BFS does not need offline processing, thus, we only report QT in the following experiments.

Exp1. Varying Graph Size ($|V|$) on ER graphs. In this experiment, we fix the density $\frac{|E|}{|V|}$=1.5 and label constraint size $|S|$=6 and vary $|V|$ from 1,000 to 10,000 to study the performance by varying graph sizes. Table 1 reports the detailed performance, such as, index sizes (IS), index building times (IT) and average query response time (QT). From Table 1, we know that transitive closure method is faster than the sampling-tree method in offline processing by orders of magnitude. For example, when $|V|$ =1K, transitive closure method only spends 15 seconds to build index, but the sampling-tree method needs 113 seconds. The index size of our method is much smaller than that in the sampling-tree method. Furthermore, our query performance are also better than the sampling tree method. From Table 1, we know that memory-based bi-directional BFS is also very fast for LCR queries. However, this method is not scalable with regard to graph size due to its exponential time complexity.

Exp2. Varying Density ($\frac{|E|}{|V|}$) on ER graphs. In this experiment, we fix $|V|$=10,000 and vary the density $\frac{|E|}{|V|}$ from 2 to 5 to study the performance of our method in dense graphs. From Table 2, we know that the index building time and index size increase when varying $\frac{|E|}{|V|}$ from 2 to 5 in both methods. Furthermore, the sampling tree method cannot finish index building in 48 hours when $\frac{|E|}{|V|} \geq 4$. From Table 2, we know that transitive closure method has better scalability with regard to the graph density $\frac{|E|}{|V|}$ than the sampling tree method. Actually, the two methods both need to compute $M_G(u, -)$ (i.e., single-source transitive closure). As proven in Theorem 3.1, our method has the minimal search space, but the search space in the sampling tree method is not minimal. Thus, large search space affects the salability of the sampling tree method.

5. CONCLUSIONS

In this paper, we address label-constraint reachability (LCR) queries over large graphs. Theoretically, we propose several methods to optimize path-label transitive closure computing. We also demonstrate the superiority of our method by extensive experiments.

6. REFERENCES

[1] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick, "Reachability and distance queries via 2-hop labels," *SIAM J. Comput.*, vol. 32, no. 5, 2003.

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

[3] R. Jin, Y. Xiang, N. Ruan, and H. Wang, "Efficiently answering reachability queries on very large directed graphs," in *SIGMOD*, 2008, pp. 595–608.

[4] R. Jin, Y. Xiang, N. Ruan, and D. Fuhry, "3-hop: a high-compression indexing scheme for reachability query," in *SIGMOD*.

[5] H. V. Jagadish, "A compression technique to materialize transitive closure," *ACM Trans. Database Syst.*, vol. 15, no. 4, 1990.

[6] H. Wang, H. He, J. Y. 0001, P. S. Yu, and J. X. Yu, "Dual labeling: Answering graph reachability queries in constant time," in *ICDE*, 2006.

[7] S. Lu, F. Zhang, J. Chen, and S.-H. Sze, "Finding pathway structures in protein interaction networks," *Algorithmica*, vol. 48, no. 4, 2007.

[8] J. P. McGlothlin and L. R. Khan, "Rdfkb: efficient support for rdf inference queries and knowledge management," in *IDEAS*, 2009.

[9] J. Zhu, Z. Nie, X. Liu, B. Zhang, and J.-R. Wen, "Statsnowball: a statistical approach to extracting entity relationships," in *WWW*, 2009.

[10] J. Cheng, J. X. Yu, X. Lin, H. Wang, and P. S. Yu, "Fast computing reachability labelings for large graphs with high compression rate," in *EDBT*, 2008.

[11] R. Bramandia, B. Choi, and W. K. Ng, "Incremental maintenance of 2-hop labeling of large graphs," *IEEE Trans. Knowl. Data Eng.*, vol. 22, no. 5, 2010.

[12] R. Jin, H. Hong, H. Wang, N. Ruan, and Y. Xiang, "Computing label-constraint reachability in graph databases," in *SIGMOD*, 2010.

[13] L. Zou, K. Xu, J. X. Yu, L. Chen, Y. Xiao, and D. Zhao, "Answering label-constraint reachability in large graphs," Peking University, Tech. Rep., 2011. [Online]. Available: http://www.icst.pku.edu.cn/intro/leizou/TR/labelConstraintQuery.pdf

[14] W. Fan, J. Li, S. Ma, N. Tang, and Y. Wu, "Adding regular expressions to graph reachability and pattern queries," in *ICDE*, 2011.

[15] Réka Albert and Albert-László Barabási, "Statistical mechanics of complex networks," *Reviews of Mordern Physics*, vol. 74, pp. 47–97, 2002.