Computing Label-Constraint Reachability in Graph Databases

Ruoming Jin, Hui Hong
Kent State University, USA
jin@cs.kent.edu
hhong@cs.kent.edu

Haixun Wang
Microsoft Research Asia
Beijing, China
haixunw@microsoft.com

Ning Ruan, Yang Xiang
Kent State University, USA
nruan@cs.kent.edu
yxiang@cs.kent.edu

ABSTRACT

Our world today is generating huge amounts of graph data such as social networks, biological networks, and the semantic web. Many of these real-world graphs are edge-labeled graphs, i.e., each edge has a label that denotes the relationship between the two vertices connected by the edge. A fundamental research problem on these labeled graphs is how to handle the label-constraint reachability query: Can vertex \( u \) reach vertex \( v \) through a path whose edge labels are constrained by a set of labels? In this work, we introduce a novel tree-based index framework which utilizes the directed maximal weighted spanning tree algorithm and sampling techniques to maximally compress the generalized transitive closure for the labeled graphs. An extensive experimental evaluation on both real and synthetic datasets demonstrates the efficiency of our approach in answering label-constraint reachability queries.

Categories and Subject Descriptors

H.2.8 [Database management]: Database Applications—graph indexing and querying

General Terms

Performance

Keywords

Label-Constraint Reachability, Generalized Transitive Closure, Maximal Directed Spanning Tree, Hoeffding and Bernstein Bounds

1. INTRODUCTION

Our world today is generating huge amounts of graph data such as social networks, biological networks, and the semantic web. How to manage these large graphs in a database system has become an important research issue in the database research community. One of the most fundamental research problems is the reachability query, which asks if one vertex can reach another or not. This is a seemingly simple but very difficult problem due to the sheer size of these large graphs. In recent years, a number of algorithms have been proposed to handle graph reachability queries [2, 15, 25, 16].

However, many real-world graphs are edge-labeled graphs, i.e., edges are associated with labels to denote different types of relationships between vertices. The reachability query for labeled graphs often involves constraints on the path connecting two vertices. Here, we list several such applications:

Social Networks: In a social network, each person is represented as a vertex and two persons are linked by an edge if they are related. The relationships between two persons are represented through different types of labels. For instance, such relationships may include parent-of, brother-of, sister-of, friend-of, employee-of, etc. Many queries in social networks seek to discover how one person \( A \) relates to another person \( B \). These queries in general can be written as if there is a path from \( A \) to \( B \) where the labels of all edges in the path are either of a specific type or belong to a specified set of labels. For instance, if we want to know whether \( A \) is a remote relative of \( B \), then we ask if there is a path from \( A \) to \( B \) where each edge label along the path is one of parent-of, brother-of, sister-of.

Bioinformatics: Understanding how metabolic chain reactions take place in cellular systems is one of the most fundamental questions in system biology. To answer these questions, biologists utilize so-called metabolic networks, where each vertex represents a compound, and a directed edge between two compounds indicates that one compound can be transformed into another one through a certain chemical reaction. The edge label records the enzymes which control the reaction. One of the basic questions is whether there is a certain pathway between two compounds which can be active or not under certain conditions. The condition can be described as the availability of a set of enzymes. Here, again, our problem can be described as a reachability query with certain constraints on the labels of the edges along the path.

To summarize, these queries ask the following question: Can vertex \( u \) reach vertex \( v \) through a path whose edge labels must satisfy certain constraints? Typically, the constraint is membership: the path’s edge labels must be in the set of constraint labels. Alternatively, we can ask for a path which avoids any of these labels. These two forms are equivalent. We note that this type of query can also find applications in recommendation search in viral marketing [19] and reachability computation in RDF graphs, such as Wikipedia and YAGO [24].

The constraint reachability problem is much more complicated than the traditional reachability query which does not consider any constraints. Existing work on graph reachability typically constructs a compact index of the graph’s transitive closure matrix. The transitive closure can be used to answer the Yes/No question of reachability, but it cannot tell how the connection is made between any two vertices. Since the index does not include the labeling information, it cannot be expanded easily to answer our aforementioned label-constraint reachability query.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGMOD ’10, June 6–11, 2010, Indianapolis, Indiana, USA.
Copyright 2010 ACM 978-1-4503-0032-2/10/06 ...$10.00.
The constraint reachability problem is closely related to the simple path query and the regular expression path query for XML documents and graphs. Typically, these queries describe the desired path as a regular expression and then search the graph to see if such a path exists or not. An XPath query is a simple iteration of alternating axes (/ and //) and tags (or labels), and it can be generalized by using regular expressions to describe the paths between two or a sequence of vertices [1]. We can look at our constraint reachability query as a special case of the regular simple path query. However, the general problem of finding regular simple paths has proven to be NP-complete [20]. The existing methods to handle such queries are based on equivalence classes and refinement to build compact indices and then match the path expression over such indices [21, 10]. On the other hand, a linear algorithm exists for the constraint reachability problem. Thus, the existing work on querying XML and graphs cannot handle our constraint reachability query efficiently.

1.1 Our Contributions

In this work, we provide a detailed study of the constraint reachability problem and offer an efficient solution. We begin by investigating two simple solutions, representing two extremes in the spectrum of solutions: an online search (DFS/BFS) approach uses the least amount of memory but has high computational cost for query answering, while the precomputation of all pair-wise path information answers the query efficiently, but uses a large amount of memory. The major research problem here is how to find a (good) compromise between these two approaches. In this work, we propose a novel tree-based index framework which can not only significantly reduce the memory cost of the precomputation approach, but also is able to answer the reachability query efficiently.

Specifically, our main contributions are as follows:

1. We introduce the label-constraint reachability (LCR) problem, provide two algorithms to categorize its computational complexity and generalize the transitive closure (Section 2).
2. We introduce a new index framework utilizing a directed spanning tree to compress the generalized transitive closure (Section 3).
3. We study the problem of constructing the index with minimal memory cost and propose methods which utilize the directed maximal weighted spanning tree algorithm and sampling techniques to maximally compress the generalized transitive closure (Section 4).
4. We present a fast query processing approach for LCR queries. Our approach is built upon an interesting link between the reachability search problem and geometric search problem, permitting us to utilize geometric search structures, such as multidimensional kd-trees or range search tree for LCR queries (Section 5).
5. We conducted detailed experiments on both real and synthetic datasets. We found our tree-based index can significantly reduce the memory cost of the generalized transitive closure while still being able to answer queries very efficiently (Section 6).

2. PROBLEM STATEMENT

A database graph (db-graph) is a labeled directed graph \( G = (V, E, \Sigma, \lambda) \), where \( V \) is the set of vertices, \( E \) is the set of edges, \( \Sigma \) is the set of edge labels, and \( \lambda \) is the function that assigns each edge \( e \in E \) a label \( \lambda(e) \in \Sigma \). A path \( p \) from vertex \( u \) to \( v \) in a db-graph \( G \) can be described as a vertex-edge alternating sequence, i.e., \( p = (u, e_1, v_1, \ldots, v_{i-1}, e_i, v_i, \ldots, e_n, v) \). When no confusion will arise, we use only the vertex sequence to describe the path \( p \) for simplicity, i.e., \( p = (v_0, v_1, \ldots, v_n) \). We use path-label \( L(p) \) to denote the set of all edge labels in the path \( p \), i.e., \( L(p) = \{ \lambda(e_1) \} \cup \{ \lambda(e_2) \} \cup \cdots \cup \{ \lambda(e_n) \} \).

Definition 1. (Label-Constraint Reachability) Given two vertices, \( u \) and \( v \) in db-graph \( G \), and a label set \( A \), where \( u, v \in V \) and \( A \subseteq \Sigma \), if there is a path \( p \) from vertex \( u \) to \( v \) whose path-label \( L(p) \) is a subset of \( A \), i.e., \( L(p) \subseteq A \), then we say \( u \) can reach \( v \) with label-constraint \( A \), denoted as \( (u, \lambda \rightarrow v) \), or simply \( v \) is \( A \)-reachable from \( u \). We also refer to path \( p \) as an \( A \)-path from \( u \) to \( v \). Given two vertices \( u \) and \( v \), and a label set \( A \), the label-constraint reachability (LCR) query asks if \( v \) is \( A \)-reachable from \( u \).

Throughout this paper, we will use Figure 1 as a running example, using integers to represent vertices and letters to represent edge labels. In this example, we can see that vertex 0 can reach vertex 9 with the label-constraint \( \{a, b, c\} \). In other words, vertex 9 is \( \{a, b, c\} \)-reachable from vertex 0. Furthermore, the path \((0, 3, 6, 9)\) is a \( \{a, b, c\} \)-path from vertex 0 to 9.

![Figure 1: Running Example](image1)

![Figure 2: (0, 9) Path-Label](image2)

In the following, we will discuss two basic approaches for answering the label-constraint reachability (LCR) queries.

2.1 Online DFS/BFS Search

The most straightforward method for answering an LCR query is online search. We can apply either DFS (depth-first search) or BFS (breadth-first search), together with the label-constraint, to reduce the search space. Let us consider the DFS for the \( A \)-reachable query from vertex \( u \) to \( v \). For simplicity, we say that a vertex \( x \) is \( A \)-adjacent to vertex \( u \), if there is an edge \( e \) linking \( u \) to \( x \), i.e., \( e = (u, x) \) and the edge label of \( e \) is contained in \( A \), \( \lambda(e) \in A \). Starting from vertex \( u \), we recursively visit all \( A \)-adjacent vertices until we either reach vertex \( v \) or have searched all the reachable vertices from \( u \) without reaching \( v \). Clearly, in \( O(|V| + |E|) \), we can conclude \( v \) is \( A \)-reachable from \( u \) or not. Thus, answering the label-constraint reachability query can be done in polynomial time. However, since the size of the db-graph is very large (it can easily contain millions of vertices), such simple online search is not acceptable for fast query answering in a graph database.

We may speedup such online search by a more “focused” search procedure. The speedup is based on the observation that in the above procedure, we may visit a lot of vertices from vertex \( u \) which cannot reach vertex \( v \) no matter what path we take. To avoid this, we can utilize the existing work on the reachability query [2, 15, 25, 16], which tells whether one vertex can reach another vertex very quickly. Thus, when we try to visit a new vertex from the current vertex, denoted as \( u' \), we require this vertex is not only \( A \)-adjacent to \( u' \), but also can reach the destination vertex \( v \) (utilizing the traditional reachability index). Note that the BFS procedure can be extended in a similar manner.

2.2 Generalized Transitive Closure

An alternative approach is to precompute the path-label set, i.e., all path-labels between any two vertices. Note that this corresponds to the transitive closure for the reachability query. The major difficulty for the path-label sets is the space cost. The upper bound for the number of all path-labels between any two vertices is \( 2^{2|\Sigma|} \).
Thus, the total storage complexity is \(O(|V|^22^{3|\Sigma|})\), which is too expensive.

However, to answer the label-constraint reachability (LCR) query, we typically only need to record a small set of path-labels. The intuition is that if vertex \(u\) can reach vertex \(v\) with label constraint \(A\), then \(u\) can reach \(v\) with any label constraint \(A' \supseteq A\). In other words, we can always drop the path-labels from \(u\) to \(v\) which are supersets of another path-label from \(u\) to \(v\) without affecting the correctness of the LCR query result. For instance, in our running example from vertex 0 to 9, we have one path \((0, 2, 9)\), which has the path-label \([b, d]\) and another path \((0, 2, 5, 8, 9)\), which has the path-label \([b, d, e, a]\). Then, for any label-constraint \(A\), we will not need to check the second path to answer the query.

To formally introduce such a reduction, we need to consider what set of path-labels are sufficient for answering LCR query.

**Definition 2. (Sufficient Path-Label Set)** Let \(S\) be a set of path-labels from vertex \(u\) to \(v\). Then, we say \(S\) is a sufficient path-label set if for any label-constraint \(A\), \(u \xrightarrow{A} v\), the LCR query returns true iff there is a path-label \(s \in S\), such that \(s \subseteq A\).

Clearly the set of all path-labels from vertex \(u\) to \(v\), denoted as \(S_0\), is sufficient. Our problem is how to find the minimal sufficient path-label set, which contains the smallest number of path-labels. This set can be precisely described in Theorem 1.

**Theorem 1.** Let \(S_0\) be the set of all path-labels from vertex \(u\) to \(v\). The minimal sufficient path-label set from vertex \(u\) to \(v\), referred to as \(S_{min}\) is unique and includes only those path-labels which do not have any (strict) subsets in \(S_0\), i.e.,

\[
S_{min} = \{ (L(p)|L(p) \in S_0 \land \exists L'(p') \in S_0, \text{such that, } L(p') \subseteq L(p) \}.
\]

In other words, for any two path-labels, \(s_1\) and \(s_2\) in \(S_{min}\), we have \(s_1 \not\subset s_2\) and \(s_2 \not\subset s_1\).

This theorem clearly suggests that we can remove all path-label sets in \(S\) which contain another path-label set in \(S\) without affecting the correctness of LCR queries. Its proof is omitted for simplicity. In addition, we can also utilize the partially ordered set (poset) to describe the minimal sufficient path-label set. Let the subset \((\subseteq)\) relationship be the binary relationship “\(\subseteq\)” over \(S_0\), i.e., \(L_1 \subseteq L_2\) iff \(L_1 \subseteq L_2\) (\(L_1, L_2 \in S_0\)). Then, the minimal sufficient path-label set is the lower bound of \(S_0\) and consists all and only minimal elements [3]. Figure 2 shows the set of all path-labels from vertex 0 to 9 using a Hasse Diagram. We see that its minimal sufficient path-label set contains only two elements: \([b, e]\) and \([a, b, c]\). The upper bound for the cardinality of any minimal sufficient path-label set is \(\binom{2}{|\Sigma|/2}\). This is also equivalent to the maximal number of non-comparable elements in the power set of \(\Sigma\). The bound can be easily observed and in combinatorics is often referred to as Sperner’s Theorem [3].

**Algorithm Description:** We present an efficient algorithm to construct the minimal sufficient path-label set for all pairs of vertices in a given graph using a dynamic programming approach corresponding to a generalization of Floyd-Warshall algorithm for shortest path and transitive closure [8].

Let \(M_k(u, v)\) denote the minimal sufficient path-label set of those paths from \(u\) to \(v\) whose intermediate vertices are in \(\{v_1, \ldots, v_k\}\). Now we consider how to compute the minimal sufficient path-label sets of the paths from each vertex \(u\) to \(v\) with intermediate vertices up to \(v_{k+1}\), i.e., \(M_{k+1}(u, v)\). Note that \(M_{k+1}(u, v)\) describe two types of paths, the first type is those paths using only intermediate vertices \(\{v_2, \ldots, v_k\}\) and the second type is those paths going through the intermediate vertices up to \(v_{k+1}\). In other words, the second type of paths can be described as a path composed of two fragments, first from \(u\) to \(k + 1\) and then from \(k + 1\) to \(j\). Given this, we can compute the minimal sufficient path-label sets using these two types of paths recursively by the following formula:

\[
M_{k+1}(u, v) = Prune(M_k(u, v) \cup \{M_k(u, k) \otimes M_k(k, v)\})
\]

\[
M_0(u, v) = \begin{cases} 
\lambda((u, v)) & \text{if } (u, v) \in E \\
\emptyset & \text{if } (u, v) \notin E 
\end{cases}
\]

Here, \(Prune\) is the function which simply drops all the path-labels which are the supersets of other path-labels in the input set. In other words, \(Prune\) will produce the lower bound of the input path-label set. The \(\otimes\) operator joins two sets of sets, such as \(\{s_1, s_2\} \otimes \{s_1', s_2', s_3'\}\) = \(s_1 \cup s_1', s_1 \cup s_2', \ldots, s_2 \cup s_3'\), where \(s_1\) and \(s_2\) are sets of labels. Besides, we can easily observe that

\[
Prune(S_1 \cup S_2) = Prune(Prune(S_1) \cup Prune(S_2))
\]

\[
Prune(S_1 \otimes S_2) = Prune(Prune(S_1) \otimes Prune(S_2))
\]

where \(S_1\) and \(S_2\) are two path-label sets. The correctness of the recursive formula for the \(M_{k+1}(u, v)\) naturally follows these two equations.

**Algorithm 1** LabeledTransitiveClosure(G(V, E, \(\Sigma\), \(\lambda\)))

**Parameter:** \(G(V, E, \Sigma, \lambda)\) is a Database Graph

1: for each \((u, v)\) in \(V \times V\) do
2: if \((u, v) \in E\) then
3: \(M[u, v] \leftarrow \{\lambda((u, v))\}\)
4: else
5: \(M[u, v] \leftarrow \emptyset\)
6: end if
7: end for
8: for \(k = 1\) to \(|V|\) do
9: for each \((u, v)\) in \(V \times V\) do
10: \(M[u, v] \leftarrow Prune(M[u, v] \cup (M[u, k] \otimes M[k, v]))\)
11: end for
12: end for

3. A TREE-BASED INDEX FRAMEWORK

The two methods in Section 1, online search and generalized transitive closure computation, represent two extremes for label-constraint reachability (LCR) query processing: the first method has the least memory cost for indexing but very high computational cost for answering the query, while the second method has very high memory cost for indexing but has low computational cost to answer the query. Thus, the major research question we address in this work is how to devise an indexing mechanism which has a low memory cost but still can process LCR queries efficiently.

The basic idea of our approach is to utilize a spanning tree to compress the generalized transitive closure which records the minimal sufficient path-label sets between any pair of vertices in the db-graph. Though spanning trees have been applied to directed acyclic graphs (DAG) for compressing the traditional transitive closure [2], our problem is very different and much more challenging. First, both db-graphs and LCR queries include edge labels which cannot be handled by traditional reachability index. Second, the db-graph in our problem is a directed graph, not a DAG. We cannot transform a directed graph into a DAG by coalescing the strongly connected components into a single vertex since much more path-label information is expressed in these components. Finally, the complexity
arising from the label combination, i.e., the path-label sets, is orders of magnitude higher than the basic reachability between any two vertices. Coping with such complexity is very challenging.

To deal with these issues, our index framework includes two important parts: a spanning tree (or forest) of the db-graph, denoted as $T$, and a partial transitive closure, denoted as $NT$, for answering the LCR query. At the high level, the combination of these two parts contains enough information to recover the complete generalized transitive closure. However, the total index size required by these two parts is much smaller than the complete generalized transitive closure. Furthermore, LCR query processing, which involves a traversal of the spanning tree structure and searching over the partial transitive closure, can be carried out very efficiently.

Let $G(V, E, \Sigma, \lambda)$ be a db-graph. Let $T(V, ET, \Sigma, \lambda)$ be a directed spanning tree (forest) of the db-graph, where $ET \subseteq E$. Note that we may not be able to find a spanning tree, instead, we may find a spanning forest. In this case, we can always construct a spanning tree by creating a virtual root to connect the roots of the trees in the spanning forest. Therefore, we will not distinguish spanning tree and spanning forests. For simplicity, we always assume there is a virtual root. For convenience, we say an edge $e \in ET$ is a tree edge and an edge in $E$ but not in $ET$ is a non-tree edge. There are many possible spanning trees for $G$. Figure 3 shows one spanning tree for our running example graph (Figure 1), where the bold lines highlight the edges in the spanning tree, and the dotted lines are the non-tree edges in the graph.

![Figure 3: Spanning Tree and Non-Tree Edges](image)

![Figure 4: Partial Transitive Closure (NT) (S. for Source and T. for Target)](image)

To utilize the spanning tree structure $T$ of the db-graph $G$ for compressing the generalized transitive closure, we formally introduce a classification of paths between any two nodes.

**Definition 3. (Path Classification)** Consider a db-graph $G$ and its spanning tree $T$. For a path $p = (v_0, e_1, v_1, \ldots, e_n, v_n)$ in $G$, we classify it into three types based on its starting edge ($e_1$) and ending edge ($e_n$):

1. $(P_s)$ contains all the paths whose starting edge is a tree-edge, i.e., $e_1 \in ET$;
2. $(P_n)$ contains all the paths whose last edge is a tree-edge, i.e., $e_n \in ET$;
3. $(P_{in})$ contains all the paths whose starting edge and end edge are both non-tree edges, i.e., $e_1, e_n \in E \setminus ET$.

We also refer the third type path $P_{in}$ as non-tree path. In addition, if all the edges in path $p$ are tree-edges, i.e., $e_i \in ET$ for $1 \leq i \leq n$, then, we refer to it as an in-tree path.

Note that a path (if it starts with a tree-edge and ends with a tree-edge) can belong to both the first and the second types of paths. Indeed, the in-tree path is such an example, and it is a special case of both $P_s$ and $P_n$ paths. In our running example (Figure 3), path $(0, 2, 9, 12)$ is an in-tree path from vertex 0 to 12, path $(0, 2, 5, 8, 11, 14, 12)$ is an example of $P_s$, path $(0, 5, 8, 11, 14, 12)$ is a non-tree path $P_n$, and path $(0, 5, 8, 9, 12)$ is an example of the second path type $P_n$.

Now, we further introduce the classification of path-label sets and especially the partial transitive closure, which carries the essential non-tree path labels.

**Definition 4. (Partial Transitive Closure)** Let $M(u, v)$ be the minimal sufficient path-label set from vertex $u$ to $v$ in $G$. Let $p$ be a path from vertex $u$ to $v$. Then we define the three subsets of $M(u, v)$ based on the path types:

1. $M_s(u, v) = \{(L(p)|p \in P) \cap M(u, v)\};$
2. $M_n(u, v) = \{(L(p)|p \in P) \cap M(u, v)\};$
3. $NT(u, v) = \{(L(p)|p \in P) \cap M(u, v) − M_s(u, v) − M_n(u, v)\};$

The partial transitive closure $NT$ records all the $NT(u, v)$, $(u, v) \in V \times V$, where $NT(u, v) \neq \emptyset$.

Clearly, the union of these three subsets is the minimal sufficient path-label set from $u$ to $v$, i.e., $M(u, v) = M_s(u, v) \cup M_n(u, v) \cup NT(u, v)$. Further, $M_s(u, v) \cap M_n(u, v)$ may not be empty, but $(M_s(u, v) \cup M_n(u, v)) \cap NT(u, v) = \emptyset$. Theorem 2 below states that we only need to record the partial transitive closure $NT$ in combination with the spanning tree $T$ in order to recover the complete transitive closure $M$. Thus, it lays the theoretical foundation of our indexing framework: $T$ and $NT$ together record sufficient information for LCR query processing. Figure 4 shows the $NT$ for the spanning tree (Figure 3) in our running example. Here, the $NT$ has only a total of 26 entries, i.e., the number of $(u, v)$ which is not empty, $NT(u, v) \neq \emptyset$, and the cardinality of each entry (the number of path-labels) is one. Among these non-empty entries, 12 of them are simply the edges in the original graph. Thus, only extra 14 entries are needed to record in $NT$ to recover the full transitive closure $M$. Note that our running example graph has a total of 16 vertices and 29 edges, and the size of $M$ is 210, $\sum_{(u,v) \in V \times V} |M(u,v)| = 210$.

**Theorem 2. (Reconstruction Theorem: $T + NT \rightarrow M(u,v)$)**

Given a db-graph $G$ and a spanning tree $T$ of $G$, let $NT$ be the partial transitive closure defined in Definition 4. Let $\text{Succ}(u)$ be all the successors of $u$ in the tree $T$. Let $\text{Pred}(v)$ be all the predecessors of $v$ in tree $T$. In addition, $u' \in \text{Succ}(u)$ and $v' \in \text{Pred}(v)$.

Then, we can construct $M(u, v)$ of path-label sets from $u$ to $v$ using $T$ and $NT$ as follows:

$$M(u, v) = \{(L(P_T(u', u'')) \odot NT(u', v') \odot L(P_T(v', v)))| u' \in \text{Succ}(u) \text{ and } v' \in \text{Pred}(v)\}$$

where, for any vertex $x$, $L(P_T(x, x)) = L(P_N(x, x)) = \emptyset$ and $NT(x, x) = \emptyset$. Then we have, for any vertices $u$ and $v$, $M(u, v) \subseteq M^*(u, v)$, and $M(u, v) = \text{Prune}(M^*(u, v))$.

Due to the space limitation, we omit the proof here. Interested readers can refer to our technical report [14] for details.

To apply this theorem for index construction and LCR query processing, we need to address the following two key research questions: 1) Different spanning trees $T$ can have very differently sized partially transitive closures $NT$. How can we find an optimal spanning tree $T$ to minimize the total index size, and specifically, the cost of partial transitive closure $NT$? 2) How can we utilize the spanning tree $T$ and partial transitive closure $NT$ to efficiently answer the LCR query? This is not a trivial question and a good query processing scheme can be as fast as we directly query the complete generalized transitive closure but with a much smaller memory cost. We will study the first question (optimal index construction) in Section 4 and the second question (efficient LCR query processing) in Section 5.
4. OPTIMAL INDEX CONSTRUCTION

In this section, we study how to construct an optimal spanning tree $T$ to minimize the cost of partial transitive closure $NT$, which is the dominant cost of our entire index framework. In Subsection 4.1, we will introduce an interesting approach based on the maximally directed spanning tree algorithm [7, 9] to tackle this problem. However, this solution relies on the fully materialization of generalized transitive closure $M$. Though this can be done using the LabeledTransitiveClosure algorithm (Algorithm 1 in Subsection 2.2), it becomes very expensive (for both computation and storage) when the size of the graph becomes large. In Subsection 4.2, we present a novel approximate algorithm which can find with high probability a spanning tree with bounded cost difference compared with the exact maximal spanning tree.

4.1 Directed Maximal Spanning Tree for Generalized Transitive Closure Compression

Let $M$ be the generalized transitive closure which contains the minimal sufficient path-label set between any two vertices $u$ and $v$. Recall that for a spanning tree $T$, $NT(u, v) = M(u, v) - M_e(u, v)$ records the "essential" path-labels of non-tree paths which cannot be replaced by either an $P_e$ path (beginning with a tree-edge) or $P_e$ path (ending with a tree-edge). Given this, the size of the partial transitive closure $NT$ can be formally defined as

$$cost(NT) = \sum_{(u, v) \in V \times V} |NT(u, v)|.$$

However, the direct minimization of $cost(NT)$ is hard and the complexity of this optimization problem remains open.

To address this problem, we consider a related partial transitive closure $MT^T$ whose cost $f(T)$ serves as the upper-bound of $cost(NT)$:

**Definition 5.** Given db-graph $G$ and its spanning tree $T$, for any vertices $u$ and $v$ in $G$, we define $MT^T(u, v)$ to be a subset of $M(u, v)$ which include those path-labels of the paths from $u$ to $v$ which ends with a non-tree edge. Formally,

$$MT^T(u, v) = M(u, v) - M_e(u, v).$$

Now we introduce several fundamental equations between $MT$, $NT$, and other subsets ($M_e$ and $M_a$) of the generalized transitive closure $M$. These expressions not only help us compute them effectively, but also help us discover the optimal spanning tree.

**Lemma 1.** Given db-graph $G$ and its spanning tree $T$, for any two vertices $u$ and $v$ in $G$, we have the following equivalence:

$$M_e(u, v) = \bigcup_{(u, u') \in E_T} \{\lambda(u, u')\} \cap M(u', v) \cap M(u, v),$$

where $u'$ is a child of $u$;

$$M_e(u, v) = M(u, v') \cap M(u, v),$$

where $(v, v') \in E_T$, $v'$ is the parent of $v$;

$$NT(u, v) = MT^T(u, v) - M_a(u, v) = M(u, v) - M_e(u, v) - M_a(u, v).$$

Due to the space limitation, we omit the proof here. Interested readers can refer to our technical report [14] for details. Clearly, the size of this new partial transitive closure $MT^T$ is no less than the size of our targeted partial transitive closure $NT (MT^T(u, v) \geq NT(u, v))$. Now we focus on the following optimization problem.

**Definition 6. (Upper Bound of NT Size and its Optimization Problem)** Given db-graph $G$ and its spanning tree $T$, let the new objective function $f(T)$ be the cost of $MT^T$, i.e.,

$$f(T) = cost(MT^T) = \sum_{(u, v) \in V \times V} |MT^T(u, v)|$$

Given the db-graph $G$, the optimization problem is how to find an optimal spanning tree $T_o$ of $G$, such that the $f(T) = cost(MT^T)$ is minimized:

$$T_o = \arg\min_T f(T)$$

Since, $cost(NT) \leq f(T)$, we also refer to $f(T)$ as the upper bound of $cost(NT)$. We will solve this problem by transforming it to the maximal directed spanning tree problem in three steps:

**Step 1 (Weight Assignment):** For each edge $(v', v) \in E(G)$ in db-graph $G$, we will associate it with a weight $w(v', v)$:

$$w(v', v) = \sum_{u \in V} |(M(u, v') \cap \{\lambda(v', v)\}) \cap M(u, v)|$$

(4)

Note that this weight directly corresponds to the number of path-labels in $M(u, v)$ which can reach $v$ via edge $(v', v)$. Especially, if we choose this edge in the spanning tree, then, we have

$$w(v', v) = \sum_{u \in V} |M_e(u, v)|.$$

Thus, this weight $(v', v)$ also reflects that if it is in the tree, the number of path-labels we can remove from all the path-label sets ending with vertex $v$ to generate the new partial transitive closure

$$w(v', v) = \sum_{u \in V} |M(u, v) - M^T(u, v)|.$$

Figure 5 shows each edge in our running example associating with the weight defined in Formula 4. For instance, edge $(1, 5)$ has a weight 13, which suggests that if we include this edge in the spanning tree, then, we can save 13 path-labels from those path-label sets reaching $5$, i.e., $\sum_{u \in V} |M(u, 5) - M^T(u, 5)| = 13$.

**Figure 5: Weighted Graph**

**Step 2 (Discovering Maximal Directed Spanning Tree):** Given this, the maximal directed spanning tree of $G$ is defined as a rooted directed spanning tree $T = (V, E_T)$ [7, 9], where $E_T$ is a subset of $E$ such that the sum of $w(u, v)$ for all $(u, v) \in E_T$ is maximized:

$$T^* = \arg\max_T W(T) = \arg\max_T \sum_{(u, v) \in E_T} w(u, v),$$

where $W(T)$ is the total weight of the spanning tree. We can invoke the Chu-Liu/Edmonds algorithm [7, 9] to find the maximal directed spanning tree of $G$.

**Theorem 3.** The maximal directed spanning tree of $G$, $T^*$, would minimize the partial transitive closure cost $f(T^*)$, which is also the upper bound of $NT$ size:

$$\min_T f(T) = f(T^*)$$

**Proof Sketch:** To prove this, we will show an important equivalent relationship between the new partial transitive closure size $M^T$, $f(T)$, and the overall weight of the directed spanning tree

$$W(T) = \sum_{(v', v) \in E_T} w(v', v).$$
Let the size of complete generalized transitive closure $M$ be
\[
\text{cost}(M) = \sum_{(u,v) \in V \times V} |M(u,v)|
\]

Then, we have for any spanning tree $T$ of $G$, the following holds:
\[
f(T) = \text{cost}(M) - W(T)
\]

Since $\text{cost}(M)$ is a constant for db-graph $G$, this equation suggests the minimization of $f(T)$ is equivalent to the maximization of $W(T)$.

We prove this equation as follows: $f(T) = \sum_{(u,v) \in V \times V} |M(u,v)| - \sum_{(u,v) \in V \times V} |M(u,v) - M_e(u,v)| = \text{cost}(M) - \sum_{(u,v) \in V \times V} |\{M(u,v') \cap \{\lambda(v',v)\} \cap M(u,v)| = \text{cost}(M) - \sum_{(u,v) \in V \times V} \sum_{(u,v') \in E} w(u,v') = \text{cost}(M) - W(T) \qquad \Box$

Finally, once the optimal spanning tree $T$ is identified, we can compute the partial transitive closure $NT$ as follows.

**Step 3 (Partial Transitive Closure NT)**: Calculating $NT$ from $M$ and $T$ according to Lemma 1.

The total computational complexity of our index construction procedure (Step 1-3) is $O(|V|^22^{2|E|})$ since the first steps take $O(|V|^22^{2|E|})$, the second step takes $O(|E| + |V|\log|E|)$, and the last step takes, $O(|V|^2(2^{|E|}))$ time all in worst case.

### 4.2 Scalable Index Construction

The aforementioned index construction algorithm relies on the pre-computation of the generalized transitive closure $M$, which becomes too expensive for the large graphs. Especially, the storage cost of $M$ can easily exceed the size of the main memory and result in memory thrashing. To make our index construction scalable to large graphs, we must avoid the full materialization of $M$. Note that the goal of the maximal spanning tree algorithm is to optimize the total weight of the tree $\sum_{(u,v) \in E_{\text{opt}}} w(u,v)$, which corresponds to the lower bound of the total saving for $NT$. The research question here is how can we discover a spanning tree whose total weight is close to the total weight of the maximal spanning tree with guaranteed probabilistic bound. Specifically, we formulate the approximate maximal spanning tree problem as follows:

**Definition 7. (Approximate Maximal Spanning Tree Problem)** Given a db-graph $G$, let $T_\delta$ be the optimal maximal spanning tree of $G$ which has the maximal total weight $W(T_\delta) = \sum_{(u,v) \in E_{\text{opt}}} w(u,v)$. The approximate maximal spanning tree problem tries to find another spanning tree $T$ of $G$, such that with probability of at least $1 - \delta$, the relative difference between its total weight $W(T)$ and the maximal tree weight $W(T_\delta)$ is no higher than $\theta$:

\[
Pr(\frac{W(T_\delta) - W(T)}{W(T_\delta)} \leq \theta) \geq 1 - \delta.
\]

In this problem, both $\epsilon$ and $\delta$ are user-defined parameters to specify the goodness of the approximate maximal spanning tree. As an example, if $\epsilon = 1\%$ and $\delta = 0.1\%$, then with probability 99.9\%, the desired tree $T$ should have a total weight no less than 99\% of the total weight of the exact maximal spanning tree $T_\delta$, i.e., $W(T) \geq 99\%(W(T_\delta))$. Note that the total weight of the approximate spanning tree can not exceed $W(T_\delta)$.

In the following, we present a novel algorithm which solves this problem through sampling, thus avoiding the full materialization of the generalized transitive closure $M$ for large graphs. In a nutshell, our algorithm works as follows:

**Step 1**: We first sample a list of vertices in the db-graph $G$.

**Step 2**: We then compute each of their generalized transitive closure in $G$, i.e., for a sample vertex $u$, we compute $M(u,v)$ for each vertex $v$ in the graph. The latter is referred to as single-source transitive closure problem and can be solved efficiently.

**Step 3**: We use the single-source $M(u,v)$ from those sample vertices to estimate the exact edge weight ($w(u',v)$, Formula (4)) and the error bound (confidence interval) for such estimation. Specifically, we leverage the combined Hoeffding and Bernstein bounds for the error bound estimation. Given this, each edge is associated with two values, one for the estimated weight and another for the error bound.

**Step 4**: We discover two maximal spanning trees in the db-graph $G$ based on each of these two values as edge weight assignment.

**Step 5**: We introduce a simple test condition using the total weights of these two trees to determine if the criterion (5) is satisfied. If the answer is no, we will repeat the above steps until the test condition holds. We refer to this algorithm as Hoeffding-Bernstein-Tree, as it utilizes Hoeffding and Bernstein Bounds [11, 22] to determine the stop condition of a sampling process. In the reminder of this subsection, we will detail the key steps in this algorithm.

**Sampling Estimator with Hoeffding and Bernstein Bounds.** Recall the edge weight
\[
w(u',v) = \sum_{u \in V} |\{M(u,v') \cap \{\lambda(u,v')\} \cap M(u,v)|
\]

In our algorithm, an important ingredient is to utilize sampling and statistical bounds (Hoeffding and Bernstein Bounds) to provide an accurate estimation of each edge weight in the graph. The key observation which enables a sampling based estimator for $w(u',v)$ is that $w(u',v)$ is the sum of $|V|$ individual quantities, $|\{M(u,v') \cap \{\lambda(u,v')\} \cap M(u,v)|$. This gives rise to this question: assuming we have $n$ samples $u_1, u_2, \ldots, u_n$, for each sample vertex $u_i$, can we compute efficiently each individual term
\[
X_{e,i} = |\{M(u_i,v') \cap \{\lambda(u,v')\} \cap M(u_i,v)|
\]

Here, we utilize sampling with replacement to simplify the statistical inference. We also focus on approximating this quantity for convenience:

\[
X_e = w(u',v)/|V|, \text{ where } e = (u',v) \in E(G).
\]

Note that since each edge weight (and error) is divided by the same constant $|V|$, the bounds for relative difference on the (total) edge weight remains the same. Indeed, we may think of the original $w(u',v)$ as the population sum and the new quantity $X_e$ is the population mean, where the weight of a quantity $v$ in the population is $|\{M(u,v') \cap \{\lambda(u,v')\} \cap M(u,v)|$. Given this, we define the sampling estimator of $X_e$ as

\[
\bar{X}_{e,n} = \frac{1}{n} \sum_{i=1}^{n} X_{e,i}
\]

Note that for each sample $X_e, E(X_{e,n}) = X_e$ and $X_e$ is a bounded random variable, i.e., $X_e \in [0, R]$ where $R = \left(\frac{|E|}{|V|^2}\right)$. As we mentioned earlier, during the sampling process, each edge $e$ is maintained with two values, one is the estimated edge weight $\hat{X}_{e,n}$ and the other is the error bound $\epsilon_{e,n}$. Using the classical Hoeffding inequality [12], the deviation (error bound) of the empirical
mean from the true expectation \( (X_{e_i}) \) is: with probability of at least \( 1 - \delta' \),
\[
|\hat{X}_{e_i} - E(\hat{X}_{e_i})| = |\hat{X}_{e_i} - E(X_{e_i})| = |\hat{X}_{e_i} - X_{e_i}| \leq R\sqrt{\frac{\ln \frac{3}{\delta'}}{2n}}
\]

However, this bound is rather loose as it decreases according to \( \sqrt{n} \). Recently, the Hoeffding bound has been improved by incorporating the empirical Bernstein bound [11, 22], which depends on only the empirical standard deviation:
\[
\sigma^2_{e_i,n} = \frac{1}{n} \sum_{i=1}^{n} (X_{e_i} - X_{e_i})^2 = \frac{1}{n} \left[ \sum_{i=1}^{n} X_{e_i}^2 - \hat{X}_{e_i}^2 \right]
\]

The combined Hoeffding and Bernstein bound is expressed as:
\[
Pr(\hat{X}_{e_i} - X_{e_i} \leq \epsilon_{e_i,n}) \geq 1 - \delta', \quad \epsilon_{e_i,n} = \sigma^2_{e_i,n} \sqrt{\frac{2\ln \frac{3}{\delta'}}{n}} + 3R\ln \frac{3}{\delta'} \quad (10)
\]

Proof Sketch: First, we have
\[
Pr(|W_n(T) - W(T)| \leq \Delta_n(T')) \geq 1 - \delta
\]

This is because with probability of at least \( 1 - \delta \),
\[
|W_n(T) - E(W_n(T))| = \left| \sum_{e \in T} \hat{X}_{e,n} - \sum_{e \in T} E(\hat{X}_{e,n}) \right| = |\sum_{e \in T} (X_{e,n} - X_{e})| = |W_n(T) - W(T)| \leq \sum_{e \in T} \epsilon_{e,n} \quad (Lemma 2)
\]

\[
\leq \Delta_n(T')(\Delta_n(T')) = \text{the maximal total error of any tree in } G
\]

Similarly, from Lemma 2, this also holds:
\[
Pr(|W_n(T_0) - W(T_0)| \leq \Delta_n(T')) \geq 1 - \delta.
\]

In addition, by definition, we have
\[
W_n(T_0) \leq W_n(T) \text{ and } W(T_0) \geq W(T).
\]

Putting them together, we have
\[
Pr(W_n(T) - \Delta_n(T') \leq W(T) \leq W_n(T) + \Delta_n(T')) \geq 1 - \delta.
\]

Thus, the following hold:
\[
Pr(W_n(T) - W(T) \leq 2\Delta_n(T')) \geq 1 - \delta \quad (13)
\]

Finally, if
\[
\frac{2\Delta_n(T')}{{W_n(T)} - \Delta_n(T')} \leq \theta(12), \text{ with } (13) \text{ and } (14) \implies \text{with probability of at least } 1 - \delta,
\]

\[
\frac{W(T)}{W_n(T)} \leq \frac{2\Delta_n(T')}{W_n(T) - \Delta_n(T')} \leq \theta \quad \Box
\]

Overall Algorithm. The sketch of the Hoeffding-Bernstein-Tree algorithm is illustrated in Algorithm 2. This algorithm performs in a batch fashion. In each batch, we sample \( n \) vertices. Lines 3–4 correspond to the sampling step (Step 1); Lines 5–11 describe the single-source transitive closure computation for each sampled vertex (Step 2); Lines 12–15 compute the two values for each edge, the estimated edge weight and its error bound (Step 3); Lines 16 and 17 find the maximal spanning trees for edge value (step 4); and finally, Line 18 tests if these two trees satisfy the condition described in Formula (12). The batch sampling reduces the invocation of MST (directed maximal spanning tree algorithm). The batch sample size \( n_0 \) is adjustable and does not affect the correctness of this algorithm. For very large graph G, we typically set \( n_0 = 100 \). As later we will show in the experimental results, the choice of \( n_0 \) does not significantly affect the running time due to the relatively cheap computational cost of the maximal spanning tree algorithm.

The SingleSourcePathLabel procedure is for computing the generalize transitive closure from a single vertex \( u \). This procedure is essentially a generalization of the well-known Bellman-Ford algorithm for the shortest path computation [8]. The procedure works in an iterative fashion. In set \( V_1 \), we record all the vertices whose path-label sets have been changed (Lines 25–26, and 30) in the current iteration. Then in the next iteration, we visit all the direct neighbors of vertices in \( V_1 \) and modify their path-label sets accordingly (Lines 22–29). For the single source computation, \( V_1 \) is initialized to contain only the single vertex \( u \). The procedure will continue until no change occurs for any path-label sets (\( V_1 = \emptyset \)). It is easy to see that the maximal number of iteration is \( |V| \) by considering the longest path vertex \( u \) can reach is \( |V| \) steps. Thus, the worst case computational complexity of SingleSourcePathLabel is \( O(|V||E|/(\epsilon^2/2 \delta^2)) \).

Given this, we can see that the total computational complexity for Hoeffding-Bernstein-Tree is \( O(n|V||E|/(\epsilon^2/2 \delta^2) + n/n_0||E| + \ldots) \).
Algorithm 2 Hoeffding-Bernstein-Tree\((G, E, \Sigma, \lambda)\)
1: \(n \leftarrow 0\) \{\(n_0\) is the initial sample size\}
2: \(\text{repeat}\)
3: \(S \leftarrow \text{SampleVertices}(V, n_0)\) \{sample \(n_0\) vertices in \(V\)\}
4: \(n \leftarrow n + n_0\) \{\(n\) is the total number of samples\}
5: \(\text{for each } n \in S \text{ do}\)
6: \(M_n \leftarrow \text{SingleSourcePathLabel}(G, u)\)
7: \(\text{for each } e \in E \{e = (v', v)\} \text{ do}\)
8: \(X_{e,n} \leftarrow \{(M_n[v'] ∩ \{\lambda(v', v)\}) ∩ M_n[v]\} \{\text{Formula (6)}\}\)
9: \(\text{Update } \sum_{e,i}^n X_{e,i} \text{ and } \sum_{e}^n X_{e,i}^\prime\)
10: \(\text{end for}\)
11: \(\text{end for}\)
12: \(\text{for each } e \in E \{e = (v', v)\} \text{ do}\)
13: \(\hat{X}_{e,n} \leftarrow \frac{1}{n} \sum_{e,i}^n X_{e,i}\) \{edge weight estimator (8)\}
14: \(e_{e,n} \leftarrow \hat{X}_{e,n} \pm \frac{2\ln \frac{2}{n}}{\sqrt{n}} + \frac{3R\ln \frac{2}{n}}{n}\) \{error bound (10)\}
15: \(\text{end for}\)
16: \(T' \leftarrow \text{MST}(G, [\hat{X}_{e,n}])\) \{Maximal Spanning Tree with \(\hat{X}_{e,n}\) as edge weight\}
17: \(T'' \leftarrow \text{MST}(G, [e_{e,n}])\) \{\(e_{e,n}\) as edge weight\}
18: \(\text{until Condition (12)} = \text{true}\)
Procedure \(\text{SingleSourcePathLabel}(G, V, E, \Sigma, \lambda, u)\)
19: \(M \leftarrow \emptyset; V_1 \leftarrow \{u\};\)
20: \(\text{while } V_1 \neq \emptyset \text{ do}\)
21: \(V_2 \leftarrow \emptyset;\)
22: \(\text{for each } v \in V_1 \text{ do}\)
23: \(\text{for each } v' \in N(v)\{v' \in N(v) : (v, v') \in E\} \text{ do}\)
24: \(\text{New} \leftarrow \text{Prune}(M[v'], M[v] ∩ \{\lambda(v, v')\});\)
25: \(\text{if } \text{New} \neq M[v'] \text{ then}\)
26: \(M[v'] \leftarrow \text{New}; V_2 \leftarrow V_2 ∪ \{v';\}\)
27: \(\text{end if}\)
28: \(\text{end for}\)
29: \(\text{end for}\)
30: \(V_1 \leftarrow V_2;\)
31: \(\text{end while}\)

\(|V| \log |V|\). The first term comes from the single source generalized transitive closure computation and the second term comes from the maximal spanning tree algorithms. Clearly, the first term is the dominant part. However, since the sample size is typically quite small (as we will show in the experimental evaluation), the Hoeffding-Bernstein-Tree algorithm is very fast. In addition, this algorithm has very low memory requirement as we need temporarily store the intermediate results of the \(\text{SingleSourcePathLabel}\) for only a single vertex. We note that this algorithm would work well when the total label size \(|E|\) is small. Since the sampling error bound is in the linear order of \(\left(\frac{\ln 2}{n}\right)^2\), when the number of distinct labels is large, the sampling size can easily exceed the total number of vertices in the graph. In this case, instead of sampling, we can simply compute the single source path label set for each vertex in the graph, and then maintain \(\sum_{e,i}^n X_{e,i}\). Therefore, in both cases, we can completely eliminate the memory bottleneck due to the fully materialization of the generalized transitive closure \(M\) for generating the spanning tree.

Finally, we note that after the spanning tree \(T\) is derived, we need to compute the partial transitive closure \(NT\). In Step 3 of the original index construction (Subsection 4.1), we rely on Lemma 1 and still need the generalized transitive closure \(M\). In the following, we describe a simple procedure which can compute the single source partial transitive closure, i.e., \(NT(u, v)\) of vertex \(u\) for each \(v \in V\). The idea is similar to the aforementioned \(\text{SingleSourcePathLabel}\) algorithm. However, instead of maintaining only one path-label set for each vertex \(v\) from \(u\), denoted as \(M[v]\) (corresponding \(M(u, v)\)) in \(\text{SingleSourcePathLabel}\), we will maintain three path-label sets based on Definition 4: 1) \(M_{u}[v]\) corresponds to \(M_{u}(u, v)\) which records all the path-labels from \(u\) to \(v\) with first edge being a tree-edge; 2) \(M_{v}[v]\), corresponds to \(M_{v}(u, v) - M_{u}(u, v)\), which records all the path-labels from \(u\) to \(v\) with last edge being a tree-edge and first edge being a non-tree edge; and 3) \(NT[v]\), corresponds to \(NT(u, v)\), which records the path-labels from \(u\) to \(v\) with both first edge and last edge being non-tree edges. Note that since each set is uniquely defined by either the first and/or last edge type in a path, we can easily maintain them during the computation of \(\text{SingleSourcePathLabel}\) algorithm.

Basically, we can compute the \(NT\) for each vertex in turn, and it has the same computational complexity as the \(\text{SingleSourcePathLabel}\) algorithm. Due to the space limitation, we omit the details here.

5. FAST QUERY PROCESSING

In this section, we study how to use the spanning tree \(T\) and partial transitive closure \(NT\) to efficiently answer the LCR queries. Using Theorem 2, we may easily derive the following procedure to answer a LCR query \((u \xrightarrow{A} v)\): 1) we perform a DFS traversal of the sub-tree rooted at \(u\) to find \(u' \in \text{Suc}(u)\), where \(L(P_{T}(u, u')) \subseteq A\); 2) for a given \(u'\), we check each of its neighbors \(v'\) in the partial transitive closure, i.e., \(NT(u', v') \neq \emptyset\) and \(v' \in \text{Pred}(v)\), to see if \(NT(u', v')\) contains a path-label which is a subset of \(A\); and 3) for those \(v'\), we check if \(L(P_{T}(v', v)) \subseteq A\). In other words, we try to identify a path-label which is a subset of \(A\) and is represented by three parts, the beginning in-tree path-label \(L(P_{T}(u, u'))\), the non-tree part \(NT(u', v')\), and the ending in-tree part \(L(P_{T}(v', v))\).

This procedure, however, is inefficient since the partial transitive closure is very sparse, and many successors of \(u\) do not link to any of \(v'\)’s predecessor through \(NT\). Moreover, the size of \(u'\)’s successor set can be very large. This leads to the following question: can we quickly identify all the \((u', v')\), where \(u' \in \text{Suc}(u)\) and \(v' \in \text{Pred}(v)\), such that \(NT(u', v') \neq \emptyset\)? The second important question we need to address is how to efficiently compute the in-tree path-labels, \(L(P_{T}(u, u'))\) and \(L(P_{T}(v', v))\), this is a key operation for the query processing for a LCR query, and we apparently do not want to traverse the in-tree path from the starting vertex, such as \(u\), to the end vertex, such as \(u'\), to construct the path-label.

If we can answer these two questions positively, how can we utilize them to devise an efficient query answering procedure for LCR queries? We investigate these questions in the following subsections.

5.1 Searching Non-Empty Entry of \(NT\)

In this subsection, we will derive an efficient algorithm by utilizing a multi-dimensional geometric search structure to quickly identify non-empty entries of the partial transitive closure: i.e., given two vertices \(u\) and \(v\), quickly identify all the vertex pairs \((u', v')\), \(u' \in \text{Suc}(u)\) and \(v' \in \text{Pred}(v)\), such that \(NT(u', v') \neq \emptyset\).

The straightforward way to solve this problem is to first find all \(u'\)’s successors (\(\text{Suc}(u)\)) and \(v'\)’s predecessors (\(\text{Pred}(v)\)), and then test each pair of the Cartesian product \(\text{Suc}(u) \times \text{Pred}(v)\), to see if the corresponding entry in \(NT\) is not empty. This method has \(O(|\text{Suc}(u)| \times |\text{Pred}(v)|)\) complexity, which is clearly too high.

Our new approach utilizes the interval-labeling scheme [10] for the spanning tree \(T\). We perform a preorder traversal of the tree to determine a sequence number for each vertex. Each vertex \(u\) in the tree is assigned an interval: \([\text{pre}(u), \text{index}(u)]\), where \(\text{pre}(u)\) is \(u\)’s preorder number and \(\text{index}(u)\) is the highest preorder number of \(u\)’s successors. Figure 6 shows the interval labeling for the spanning tree in Figure 3. It is easy to see that vertex \(u\) is a predecessor of vertex \(v\) iff \([\text{pre}(v), \text{index}(v)] \subseteq [\text{pre}(u), \text{index}(u)]\) [10].

The key idea of our new approach is to utilize the spanning tree to transform this search problem into a geometric range-search prob-
lem. Simply speaking, we map each non-empty entry \((u', v')\) of \(NT\) into a four-dimensional point and convert the query vertices \(u\) and \(v\) into an axis-parallel range. We describe our transformation and its correctness using the following theorem.

**Theorem 5.** Let each pair \((u', v')\), where \(NT(u', v') \neq \emptyset\), be mapped to a four-dimensional point \((\text{pre}(u'), \text{index}(u'), \text{pre}(v'), \text{index}(v'))\). Then, for any two query vertices \(u\) and \(v\), the vertex pair \((u', v')\), \(u' \in \text{Succ}(u)\) and \(v' \in \text{Pred}(v)\), such that \(NT(u', v') \neq \emptyset\), corresponds to a four-dimensional point \((\text{pre}(u'), \text{index}(u'), \text{pre}(v'), \text{index}(v'))\) in the range: \([\text{pre}(u), \text{index}(u)], [\text{pre}(u), \text{index}(u)], [\text{pre}(v), \text{index}(v)], [\text{index}(v), \text{index}(v)]\) of a four-dimensional space.

Due to the space limitation, we omit the proof here. Interested readers can refer to our technical report [14] for details.

Figure 7 gives the 4-D coordinates \((\text{pre}(u'), \text{index}(u'), \text{pre}(v'), \text{index}(v'))\) for each NT \((u', v')\). For example, consider a query \((u, v)\) on graph \(G\) with spanning tree given as in Figure 6, where \(u = 11\) and \(v = 6\). There is a vertex pair \((u', v')\) where \(NT(u', v') = \{c\}\) and \(u' = 14\) and \(v' = 12\) such that \(u' \in \text{Succ}(u)\) and \(v' \in \text{Pred}(v)\). It is easy to verify \((\text{pre}(u'), \text{index}(u'), \text{pre}(v'), \text{index}(v'))\), which is \((14, 14, 4, 15)\) within the range \([\text{pre}(u), \text{index}(u)], [\text{pre}(u), \text{index}(u)], [\text{pre}(v), \text{index}(v)], [\text{index}(v), \text{index}(v)]\).

This transformation allows us to apply the test conditions for both the successors of \(u\) and the predecessors of \(v\) simultaneously, and those test conditions correspond to a multi-dimensional range search. Using a kd-tree or range search tree, we can efficiently index and search all the points in an axis-parallel range [5]. Specifically, the construction of kd-tree takes \(O(n \log^2 n)\), and querying an axis-parallel range in a balanced kd-tree takes \(O(n^{3/4} + k)\) time (in four dimensional space), where \(n\) is the total number of pairs with non-empty entries in \(NT\), and \(k\) is the number of the reported points. The range-search tree provides faster search time. It can be constructed in \(O(n \log^d n)\) time and answers the query with \(O((\log n)^d + k)\) time (for four dimensional space).

### 5.2 Computing In-Tree Path-Labels

In this subsection, we present a histogram-based technique to quickly compute the path-label of an in-tree path. Let \(x\) and \(y\) be two vertices in a tree \(T\) where \(x\) is \(y\)'s ancestor, \(x \in \text{Pred}(y)\). Our goal is to compute the path-label set of the in-tree path from \(x\) to \(y\), \(L_P(x, y)\), very fast. We build a histogram for each vertex in \(T\). Let \(v\) be the root vertex of \(T\). Then, the histogram of vertex \(u\), denoted as \(H(u)\), records not only each unique label in the in-tree path from the root vertex to \(u\), i.e., \(L_P(x, u)\), but also the number of edges in the path containing each label. For instance, the histogram for vertex 14 in Figure 3 is \(H(14) = \{b : 1, d : 2, e : 2\}\) and the histogram for vertex 6 is \(H(6) = \{a : 2, b : 2, c : 1, d : 1, e : 1\}\). Clearly, a simple DFS traversal procedure can construct the histogram for each vertex of \(T\) in \(O(|V||\Sigma|)\).

We can compute the path-label sets from vertex \(u\) to \(v\) by subtracting the corresponding counters in \(H(u)\) from \(H(v)\). Specifically, for each edge label in \(H(v)\), we will subtract its counter in \(H(v)\) by the counter of the same label in \(H(u)\), if it exists. If the counter of the same label does not appear in \(H(u)\), we treat the counter as 0. Given this, the path-label set from \(u\) to \(v\) include all the labels whose resulting counter is more than 0, i.e., we know that there is an edge with this label on the in-tree path from \(u\) to \(v\). Clearly, this approach utilizes \(O(|V||\Sigma|)\) storage and can compute an in-tree path-label in \(O(|\Sigma|)\) time, where \(|\Sigma|\) is the number of distinct labels in tree \(T\). Since the number of possible labels, \(|\Sigma|\), is typically small, and can be treated as a constant, this approach can compute the path-label for any in-tree path very fast.

### 5.3 LCR Query Processing

Here, we present a new algorithm for fast LCR query processing which utilizes the new technique for efficiently searching non-empty entries in \(NT\) developed in last subsection. The sketch of the algorithm is in Algorithm 3. For a LCR query \((u \rightarrow v)\), in the first step (Line 1), we search for all the vertex pairs \((u', v')\), where \(u' \in \text{Succ}(u)\) and \(v' \in \text{Pred}(v)\), such that \(NT(u', v') \neq \emptyset\). We can achieve this very efficiently by transforming this step into a geometric search problem and then utilizing a kd-tree or range-search tree to retrieve all targeted pairs. We put all these pairs in a list \(L\). Then, we test each vertex pair \((u', v')\) in \(L\) to see if we can construct a path-label using these three segments, \(P_T(u, u')\), \(NT(u', v')\), and \(P_T(v, v')\). Here, the in-tree path-label is computed using the histogram-based approach. Thus, we first test if the in-tree path-label sets, \(L(P_T(u, u'))\) and \(L(P_T(v, v'))\), are subsets of \(A\), and then check if \(NT(u', v')\) contains a path-label set which is a subset of \(A\).

We aggressively prune those \((u', v')\) pairs which will not be able to connect \(u\) to \(v\) with label constraint \(A\). This is done in Lines 10 and 13. Note that when \(L(P_T(u, u'))\) is not a subset of \(A\), we know any vertex in \(\text{Succ}(u')\) (i.e. any successor of \(u'\) in tree \(T\)) will not be \(A\)-reachable from \(u\) using the in-tree path. Similarly, when \(L(P_T(v, v'))\) is not a subset of \(A\), we can infer any vertex in \(\text{Pred}(v')\) (i.e. any predecessor of \(v'\) in tree \(T\)) will not reach \(v\) with label constraint \(A\) using the in-tree path. Thus, we can prune those pairs from the list \(L\).

**Algorithm 3 QueryProcessing\((u, A, v)\)**

**Parameter:** Tree \(T\) and Partial Transitive Closure \(NT\)

**Parameter:** LCR query: \(u \rightarrow A \rightarrow v\)

1: Finding all \((u', v')\) such that \(u' \in \text{Succ}(u)\) and \(v' \in \text{Pred}(v)\), and such that \(L(P_T(u, u')) \subseteq A\) and \(L(P_T(v, v')) \subseteq A\) and \(NT(u', v') \neq \emptyset\), and store them in a list \(L\)
2: while \(L \neq \emptyset\) do
3: \((u', v') \leftarrow \text{L.pop()}\)
4: if \(L(P_T(u, u')) \subseteq A\) then
5: if \(L(P_T(v, v')) \subseteq A\) then
6: if \(\exists l \in NT(u', v'), l \subseteq A\) then
7: return true
8: end if
9: else
10: delete in \(L\) those \((u'', v'')\) with \(v'' \in \text{Pred}(v')\)
11: end if
12: else
13: delete in \(L\) those \((u'', v'')\) with \(u'' \in \text{Succ}(u')\)
14: end if
15: end while
16: return false

The computational complexity of this procedure is \(O(n \log n + \sum_{i=1}^{k} |NT(u_i, v_i)|)\), where \(n\) is the total number of non-empty entries in \(NT\), and \(k\) is the resulting non-empty pairs of \(NT\), which
are from the first step, listed as \( (u, v) \). We further assume the in-tree path label-set can be computed in constant time since the number of possible labels is fixed and typically small.

6. EXPERIMENTS

In this section, we perform a detailed experimental study on both real and synthetic datasets. Specifically, we are interested in understanding the trade-off between index size, query time, and index construction time on the five approaches presented in this work: 1) online depth-first search (DFS in Subsection 2.1); 2) Focused depth-first search (Focused DFS in Subsection 2.1); 3) the complete generalized transitive closure which is constructed using generalized Floyd-Warshall (algorithm 1 in Subsection 2.2); 4) Optimal Spanning Tree (Opt-Tree), constructed after generating the generalized transitive closure by Warshall (Subsection 4.1); and 5) the Hoeffding-Bernstein-Tree (Algorithm 2 in Subsection 4.2), referred to as Sampling-Tree in this section. In our framework, we use the multi-dimensional geometric data structure, kd-tree, for searching the non-empty entries of \( NT \). All these algorithms are implemented using C++, and our experiments were performed on a 2.0GHz Dual Core AMD Opteron(tm) with 6GB of memory.

6.1 Setup on Synthetic Datasets

We use a collection of synthetic datasets, in five groups of experiments, to study several different aspects of the five approaches for answering LCR queries. The synthetic datasets are generated in two steps. First, we generate two types of random directed graphs, one is based on Erdös-Rényi (ER) model [17] and the other is based on the scale-free (SF) model [4]. Second, we assign a random label associated with each edge in these generated random graphs. The distribution of labels is generated according to a power-law distribution to simulate the phenomena that only a few labels appear rather frequently, while the majority of labels appear infrequently.

- Expr1.a (Varying Density on ER Graphs): In the first group of experiments, we study how the index size, query time, and index construction time vary according to the edge density. We fix the number of vertices \( |V| = 5000 \). Then we range density \( \frac{|E|}{|V|} \) from 1.5 to 5 according to ER model.
- Expr1.b (Varying Query Constraint Size on ER Graphs): We fix \( |V| = 5000 \), and density \( \frac{|E|}{|V|} = 1.5 \), then change query label constraints \( |A| \) from 15% to 85% of the total edge labels according to ER model.
- Expr1.c (Scalability on ER Graphs): We perform experiments on graphs with density \( \frac{|E|}{|V|} = 1.5 \), then vary the graph size \( |V| \) from 20,000 to 100,000.
- Expr1.d (Varying Query Constraint Size on SF Graphs): We fix \( |V| = 5000 \), and generate scale-free graphs based on RMAT algorithm [6], then change query label constraints \( |A| \) from 15% to 85% of the total edge labels.
- Expr1.e (Scalability on SF Graphs): We perform scalability experiments on scale-free graphs based on RMAT algorithm, by varying the graph size \( |V| \) from 20,000 to 100,000.

In addition, for each randomly generated graph, we set the total number of possible edge labels \( |\Sigma| \) to be 20. In Expr1.a, and Expr1.c, we fix the query label set \( |A| \) to be 30% of the number of total edge labels \( |\Sigma| \). Also, we set the power-law distribution parameter \( \alpha = 2 \) [23]. In addition, for the Sampling-Tree, we set the relative error bound \( \theta = 1\% \) and user confidence level parameter \( \delta = 1\% \).

6.2 Results on Synthetic Datasets

We report query time, construction time and index size for the experiments on the synthetic datasets. Specifically, the query time reported is the overall running time for each method, DFS, Focused DFS, Warshall, Sampling-Tree and Opt-Tree, to process a total of 10,000 random LCR queries, in milliseconds (ms), while the construction time is measured in seconds (s). The construction time for Sampling-Tree is the time to construct the approximate maximal spanning tree and \( NT \), while the Warshall construction time is the time to get the full transitive closure; Index size is measured in Kilobytes (KB). For Warshall, the index size is the size of the generalized transitive closure; For Sampling-Tree, it is the size of the spanning tree \( T \) plus \( NT \). Further, the sampling size is the number of sampled vertices for Sampling-Tree (Hoeffding-Bernstein-Tree, Algorithm 2) and “.” means the method does not work on the graph (mainly due to the memory cost).

- Expr1.a Varying Density on ER graphs: In this experiment, we fix \( |V| = 5000 \) and range density \( \frac{|E|}{|V|} \) from 1.5 to 5, to study how the index size, query time, and index construction time vary according to the graph density. Table 1(a) shows that Sampling-Tree can be effectively constructed using at most 800 vertices, performing as well as the exact optimal maximal spanning tree, since the weight error is quite small. Table 1(b) shows that in Expr1.a, the index size for Sampling-Tree is not only small but also increases by only 1.5 when the density increases from 1.5 to 3.0. The index size for Warshall is not only 29 to 510 times larger, but also much more sensitive to density. Although the construction time for Sampling-Tree for \( \frac{|E|}{|V|} \) from 3.5 to 5 is about 10 times larger than for \( \frac{|E|}{|V|} = 1.5 \), Warshall completely fails to work under high density. In addition, the query time of Sampling-Tree remains almost unchanged with the increase of \( \frac{|E|}{|V|} \) as shown in Figure 10(a), although the query time of DFS and Focused DFS increases significantly. This shows that Sampling-Tree can handle relatively dense graphs.
- Expr1.b Varying Query Constraint Size on ER Graphs: In this experiment, we fix \( |V| = 5000 \), density \( \frac{|E|}{|V|} = 1.5 \), then change query label constraints \( |A| \) from 15% to 85% of the total edge labels to compare how these approaches differ from each other. Table 1(c) shows that in Expr1.b the Warshall index size is about 400 times that of Sampling-Tree, while the construction time is about 15 times Sampling-Tree’s. In addition, the query time for DFS is 20 to 40 times Sampling-Tree’s when \( |A| \) increases as shown in Figure 10(b). To sum, this experiment shows that Sampling-Tree can perform well when increasing the query constraint size. Finally, we note that the performance of Focused DFS is even worse than the DFS without the reachability indexing. By further analysis, we found that the search space being pruned due to the reachability index is very small as the directed graph is quite well-connected. In this case, querying the traditional reachability index at each vertex during the search process does not pay off and simply becomes the additional cost of the Focused DFS.
- Expr1.c Scalability on ER Graphs: In this experiment, we fix density \( \frac{|E|}{|V|} = 1.5 \), then vary the graph size \( |V| \) from 20,000 to 100,000. Due to the memory cost for the reachability index, Focused DFS cannot handle very large graphs. In addition, Warshall and Opt-Tree also fail due to memory bottleneck when the number of vertices exceeds 20,000 (Table 2). Therefore, we only report experimental results on Sampling-Tree and DFS. Table 2 shows that only a small number of sample vertices (S-S) is needed for constructing the optimal spanning tree. Figure 10(c) shows the query time for DFS is 75 to 400 times that of Sampling-Tree. To sum, the Sampling-Tree scales well when increasing the graph size.
with 66 labels, and has density $|E|/|V| = 5.7$. Since the label size is large, we do not perform sampling in Sampling-Tree. Instead, we simply compute single source path labels. We still refer to it as Sampling-Tree as this process can be viewed as a special case of the sampling algorithm. Here, we vary the label-constraint $|A|$ from 20 to 60. Figure 9 shows that our indexing approach is 5 to 31 times faster on the yago graph. To sum, we see that in real datasets, our approach is effective and efficient for answering label-constraint queries.

7. CONCLUSION

In this paper, we introduced a new reachability problem with label constraint. This problem has a lot of potential applications in real world applications, ranging from social network analysis, viral marketing, to bioinformatics, and RDF graph management. On one side, this problem is more complicated than the traditional reachability query which does not involve any label constraint. On another side, this problem can be looked as a special case of the simple regular expression path query on a labeled graph. However, instead of being NP-hard, our new problem can be solved in polynomial time. In this work, we develop a novel tree-based framework to compress the generalized transitive closure for labeled graph. We introduce a novel approach to optimize the index size by utilizing the directed maximal weighted spanning algorithms. We derive a fast query processing algorithm based on the geometric search data structures. Our experimental evaluation on both real and synthetic datasets showed that our approach can compress the transitive closure on average by more than two orders of magnitude, while it can deliver the query processing being very close to the fully materialized transitive closure.

8. REFERENCES


Figure 10: Experimental Results on Synthetic Datasets

- Expr1.a: Density $|E|/|V|$ from 1.5 to 5;
- Expr1.b: $|A|/|\Sigma|$ from 0.15 to 0.85, when $|V| = 5000$;
- Expr1.c: $|V|$ from 20000 to 100000, when $|V| = 5000$ for SF graphs;
- Expr1.d: $|A|/|\Sigma|$ from 0.15 to 0.85, when $|V| = 5000$ for SF graphs;
- Expr1.e: $|V|$ from 20000 to 100000 for SF graphs;