Computing Label-Constraint Reachability in Graph Databases

Ruoming Jin[1] Hui Hong[1] Haixun Wang[2] Ning Ruan[1] Yang Xiang[1]
[1]Kent State University, Kent, OH 44242
{jin, hhong, nruan, yxiang}@cs.kent.edu
[2]Microsoft Research Asia
haixunw@microsoft.com

ABSTRACT

A huge amount of graph data (biological networks, semantic web, ontologies, social networks) is being generated. Many of these real-world graphs are edge-labeled graphs, i.e., each edge is associated with a label that denotes the relationship between the two vertices connected by the edge. Given this, a fundamental research problem on the 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 formally introduce the labeled reachability query problem, and provide two initial algorithms that categorize its computational complexity. We then generalize the transitive closure for the labeled graph, and introduce a new index framework which utilizes a directed spanning tree to compress the generalized transitive closure. We optimize the index for minimal memory cost and we propose methods which utilize the directed maximal weighted spanning tree algorithm and sampling techniques to maximally compress the generalized transitive closure. After establishing an interesting link between the reachability problem and the geometric search problem, we utilize the geometric search structures, such as multidimensional kd-trees or the range-search trees, to enable fast query processing. In our extensive experimental evaluation on both real and synthetic datasets, we find our tree-based index can significantly reduce the memory cost of the generalized transitive closure while still being able to answer the query very efficiently.

Categories and Subject Descriptors

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

General Terms

Performance

Keywords

Graph indexing, Reachability queries, Transitive closure, Maximal directed spanning tree, Sampling

1. INTRODUCTION

A huge amount of graph data (biological networks, semantic webs, social networks) is being generated. 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 [16, 2, 27, 7, 26, 17, 8].

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*, *student-of*, *brother-of*, *sister-of*, *friend-of*, *employee-of*, *consultant-of*, *follower-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 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*, *child-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 [20] and reachability computation in RDF graphs, such as Wikipedia and YAGO [25].

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.

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 [21]. 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 [22, 12]. On the other hand, a linear algorithm exists for the constraint reachability problem. Thus, the existing work on quering XML and graphs cannot provide an efficient solution to our constraint reachability query.

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. First, we present an online search (DFS/BFS) algorithm. This method has very low cost for index construction and minimal index size. However, online search can be expensive for very large graphs. Second, we consider precomputing the necessary path information (of labels) between any two vertices. This is similar to building the transitive closure for answering the traditional reachability query. This approach can answer the constraint reachability query much faster than the first method since we can use the precomputed table to answer queries without traversing the graph. However, the size of the precomputed path-label sets is much larger than that of the traditional transitive closure for reachability query. In summary, the first approach uses the least amount of memory but has high computational cost for query answering, while the second approach 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. Specifically, we would like to significantly reduce the memory cost of the fully precomputed table and still be able to answer the reachability query efficiently. In this work, we propose a novel *tree-based index framework* for this purpose. Conceptually, we decompose any path into three fragments: beginning, end, and middle fragments. Its beginning and end parts always come from a spanning tree structure, and its middle fragment will be precomputed by a partial transitive closure. We will not fully materialize the transitive closure, but only record a small portion of it, which we refer to as the *partial transitive closure*. Interestingly, we find that the problem of minimizing the size of the partial transitive closure can be effectively transformed into a *maximal directed spanning tree problem*. We also devise a query processing scheme which will effectively utilize *multi-dimensional geometric search structures*, such as kd-tree or range search tree, to enable fast query processing.

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 which utilizes 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 a detailed experimental evaluation 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 the query 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, Σ is the set of edge labels, and λ 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., $p = (u, e_1, v_1, \cdots, v_{i-1}, e_i, v_i, \cdots, e_n, v)$. When no confusion will arise, we use only the vertex sequence to describe the path p for simplicity, i.e., $p = (v_0, v_1, \cdots, v_n)$. 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 $(u \xrightarrow{A} v)$, 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.*

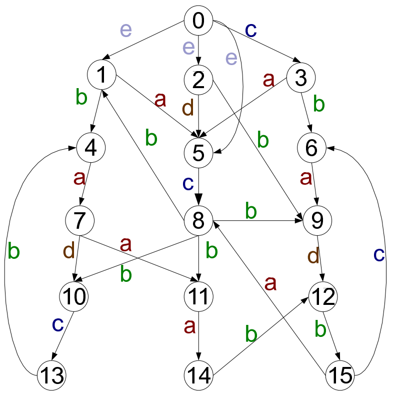


Figure 1: Running Example

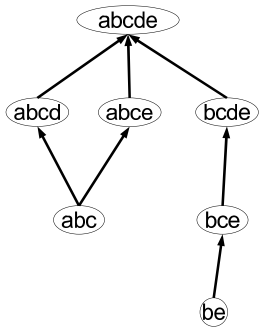


Figure 2: $(0, 9)$ **Path-Label**

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.

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 straight-forward 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 [16, 2, 27, 7, 26, 17, 8], 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 $2^{|\Sigma|}$. Thus, the total storage complexity is $O(|V|^2 2^{|\Sigma|})$, 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 if and only if 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 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.,*

$$S_{min} = \{L(p)|L(p) \in S_0 \wedge \nexists L(p') \in S_0,\ such\ that,\ L(p') \subset L(p)\}$$

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 order set (poset) to describe the minimal sufficient path-label set. Let the subset ($\subseteq$) relationship be the binary relationship "$\preceq$" over S_0, i.e., $L_1 \preceq L_2$ 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 $\binom{|\Sigma|}{\lfloor|\Sigma|/2\rfloor}$. This is also equivalent to the maximal number of non-comparable elements in the power set of Σ. 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 [10]).

Let $M_k(u, v)$ denote the minimal sufficient path-label set of those paths from u to v whose intermediate vertices are in $\{v_1, \cdots, v_k\}$. 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 $(v_1, \cdots, v_k)$ 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:

$$\begin{aligned} M_{k+1}(u, v) &= Prune(M_k(u, v) \cup (M_k(u, k) \odot M_k(k, v))); \\ M_0(u, v) &= \begin{cases} \lambda((u, v)) & \text{if} \quad (u, v) \in E \\ \emptyset & \text{if} \quad (u, v) \notin E \end{cases} \end{aligned}$$

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 $\odot$ operator *joins* two sets of sets, such as $\{s_1, s_2\} \odot \{s'_1, s'_2, s'_3\} = \{s_1 \cup s'_1, s_1 \cup s'_2, \cdots, s_2 \cup s'_3\}$, where s_i and s'_j are sets of labels. Besides, we can easily observe that

$$Prune(S_1 \cup S_2) = Prune(Prune(S_1) \cup Prune(S_2))$$
$$Prune(S_1 \odot S_2) = Prune(Prune(S_1) \odot Prune(S_2))$$

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 sketches the dynamic programming procedure which constructs the minimal sufficient path-label sets for all pairs of vertices in a graph G. The worst case computational complexity is $O(|V|^3 2^{|\Sigma|})$ for this dynamic procedure and its memory complexity is $O(|V|^2 \binom{|\Sigma|}{\lfloor|\Sigma|/2\rfloor})$.

3. A TREE-BASED INDEX FRAMEWORK

The two methods we described 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 the present work is how to devise an indexing mechanism which has a low memory cost but still can process LCR queries efficiently.

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($M[u, v] \cup (M[u, k] \odot M[k, v])$)
11:     end for
12: end for
```

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 $T(V, E_T, \Sigma, \lambda)$ be a directed spanning tree (forest) of the db-graph, where $E_T \subseteq E$. 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 $e \in E_T$ is a *tree edge* and an edge in E but not in E_T is a *non-tree edge*. There are many possible spanning trees for G. Figure 3 shows one spanning tree for our running example graph

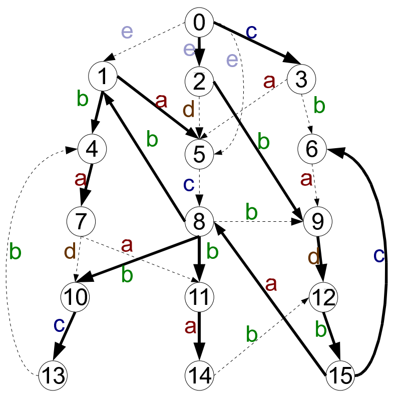


Figure 3: Spanning Tree and Non-Tree Edges

S.	T.	NT	S.	T.	NT
0	1	{e}	5	12	{c, b, a}
0	5	{e}	6	9	{a}
0	8	{e, c}	7	4	{c, b, d}
0	12	{e, b, a}	7	9	{b, a}
2	5	{d}	7	10	{d}
2	8	{c, d}	7	11	{a}
3	5	{a}	7	12	{b, a}
3	6	{b}	8	9	{b}
3	8	{c, a}	13	4	{b}
3	9	{b, a}	13	9	{b, a}
3	12	{c, b, a}	13	12	{b, a}
5	8	{c}	14	9	{b, a}
5	9	{c, b}	14	12	{b}

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

(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.

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 $p = (v_0, e_1, v_1, \cdots, e_n, v_n)$ 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., $e_1 \in E_T$;*
2. **(P_e)** *contains all the paths whose last edge is a tree-edge, i.e., $e_n \in E_T$;*
3. **(P_n)** *contains all the paths whose starting edge and end edge are both non-tree edges, i.e., $e_1, e_n \in E \backslash E_T$.*
We also refer the third type path P_n as **non-tree path**. *In addition, if all the edges in path p are tree-edges, i.e., $e_i \in E_T$ 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_e 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_e.

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. *$M_s(u, v) = \{L(p) | p \in P_s\} \cap M(u, v)$;*
2. *$M_e(u, v) = \{L(p) | p \in P_e\} \cap M(u, v)$;*
3. *$NT(u, v) = \{L(p) | p \in P_n\} \cap M(u, v) - M_s(u, v) - M_e(u, v)$;*
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., $M(u, v) = M_s(u, v) \cup M_e(u, v) \cup NT(u, v)$. Further, $M_s(u, v) \cap M_e(u, v)$ may not be empty, but $(M_s(u, v) \cup M_e(u, v)) \cap NT(u, v) = \emptyset$. 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 $Succ(u)$ be all the successors of u in the tree T. Let $Pred(v)$ be all the predecessors of v in tree T. In addition, $u' \in Succ(u)$ and $v' \in Pred(v)$. Then, we can construct $M'(u, v)$ of path-label sets from u to v using T and NT as follows:*

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

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 $M(u, v) = Prune(M'(u, v))$.

Proof Sketch:

To prove this theorem, we will establish the following three-segment path decomposition scheme for any path p from a vertex u to another vertex v, i.e., $p = (v_0, e_1, v_1, \cdots, e_n, v_n)$, where $u = v_0$ and $v = v_n$. Let e_i be the first non-tree edge and e_j be the last non-tree edge in path p ($i \leq j$). Let u' be the beginning vertex of the first non-tree edge e_i, $u' = v_{i-1}$ and v' be the end vertex of the last non-tree edge e_j, $v' = v_j$. Now, we can decompose path p into three segments: *1) the starting in-tree path from u to u', denoted as $P_T(u, u')$; 2) the intermediate non-tree path from u' to v', denoted as $P_N(u', v')$; 3) the ending in-tree path from v' to v denoted as $P_T(v', v)$*. Note that certain segments can be empty and we define *the empty segment* as $P_N(x, x) = P_T(x, x)$ for any vertex x.

Given this, let us consider such decomposition for each type of path from u to v.

Case 1: If there is an *in-tree path* p from u to v, then we can directly represent it as

$$L(p) = L(P_T(u, v)) = L(P_T(u, u')) \cup \emptyset \cup L(P_T(u', v))$$

where $u' \in \{v_1, \cdots, v_n\}$ and $u' \in Succ(u)$ and $u' \in Pred(v)$ since p is an in-tree path.

Case 2: If $p \in (P_s \cup P_e) \backslash P_T(u, v)$ (path p is a P_s or P_e path but not an in-tree path), then, there is at least one *non-tree edge* in p. Thus, we can find u' and v' in path p, such that $u' \in Succ(u)$, $v' \in Pred(v)$ and the *subpath* of p from u' to v' is a *non-tree path*. Then, if $L(p) \in M(u, v)$, we have

$$L(p) \in \{L(P_T(u, u'))\} \odot NT(u', v') \odot \{L(P_T(v', u))\}$$

Case 3: If $p \in P_n$ (a non-tree path), then, similar analysis as *Case* 2 holds.

Putting these together, we have

$$\begin{aligned} M(u, v) \quad = \quad & Prune(\{\{L(P_T(u, u'))\} \odot NT(u', v') \odot \{L(P_T(v', v))\}| \\ & u' \in Succ(u) \text{ and } v' \in Pred(v)\}) \end{aligned}$$

□

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 [9, 11] 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, $NT(u, v) = M(u, v) - M_s(u, v) - M_e(u, v)$ records the "essential" path-labels of non-tree paths which cannot be replaced by either an type P_s 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 M^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 $M^T(u, v)$ to be a subset of $M(u, v)$ which includes those path-labels of the paths from u to v which ends with a non-tree edge. Formally,*

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

Now we introduce several fundamental equations between M^T, NT, and other subsets (M_s and M_e) 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:*

$$\begin{aligned} M_s(u, v) &= \Big(\bigcup_{(u,u') \in E_T} \{\lambda(u, u')\} \odot M(u', v)\Big) \cap M(u, v), && (1) \\ & \quad \textit{where } u' \textit{ is a child of } u; \\ M_e(u, v) &= (M(u, v') \odot \{\lambda(v, v')\}) \cap M(u, v), && (2) \\ & \quad \textit{where } (v, v') \in E_T(\ v' \textit{ is the parent of } v); \\ NT(u, v) &= M^T(u, v) - M_s(u, v) \\ &= M(u, v) - M_e(u, v) - M_s(u, v) && (3) \end{aligned}$$

Proof Sketch: Simply note that 1) $\bigcup_{(u,u') \in E_T} \{\lambda(u, u')\} \odot M(u', v)$ includes all sufficient path-labels for the paths from u to v starting with a tree-edge (u, u'); and 2) $M(u, v') \odot \{\lambda(v', v)\})$ include the sufficient path-labels for the paths from u to v ends with the tree-edge (v', v). □

Clearly, the size of this new partial transitive closure M^T is no less than the size of our targeted partial transitive closure NT ($M^T(u, v) \supseteq NT(u, v)$). 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 M^T, i.e.,*

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

Given the db-graph G, the optimization problem is how to find an optimal spanning tree T_o of G, such that the $f(T) = cost(M^T)$ 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 $w(v', v)$:*

$$w(v', v) \quad = \quad \sum_{u \in V} |(M(u, v') \odot \{\lambda(v', v)\}) \cap M(u, v)| \qquad (4)$$

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., $\sum_{u \in V} |M(u, 5) - M^T(u, 5)| = 13$.

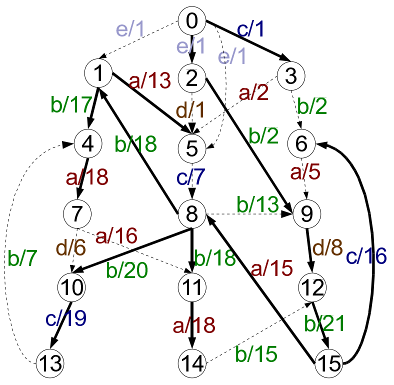


Figure 5: Weighted Graph

Step 2 (Discovering Maximal Directed Spanning Tree): Given this, the maximal directed spanning tree of G is defined as a rooted directed spanning tree $T = (V, E_T)$ [9, 11], 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 [9, 11] to find the maximal directed spanning tree of G.

Theorem 3. *The maximal directed spanning tree of G, T', would minimize the new partial transitive closure M^T, $f(T)$, 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

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

Then, we have for any spanning tree T of G, the following holds:

$$f(T) = cost(M) - W(T)$$

Since $cost(M)$ 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: $f(T) =$

$$\begin{aligned}
& \sum_{(u,v) \in V \times V} |M^T(u, v)| = \sum_{(u,v) \in V \times V} |M(u, v) - M_e(u, v)| \\
= & \sum_{(u,v) \in V \times V} (|M(u, v)| - |(M(u, v') \odot \{\lambda(v', v)\}) \cap M(u, v)|) \\
= & \ cost(M) - \sum_{(u,v) \in V \times V} |(M(u, v') \odot \{\lambda(v', v)\}) \cap M(u, v)| \\
= & \ cost(M) - \sum_{(v',v) \in E} \sum_{u \in V} |(M(u, v') \odot \{\lambda(v', v)\}) \cap M(u, v)| \\
= & \ cost(M) - \sum_{(v',v) \in E} w(v', v) = cost(M) - W(T) \qquad \square
\end{aligned}$$

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 $O(|V|^2 2^{2^{|\Sigma|}})$ since the first steps takes $O(|V|(2^{|\Sigma|})^2)$, the second step takes $O(|E|+|V|log|V|)$, and the last step takes, $O(|V|^2(2^{|\Sigma|})^2)$ 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 $\sum_{(u,v) \in T} w(u, v)$, 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_o be the optimal spanning tree of G which has the maximal total weight $W(T_o) = \sum_{(v',v) \in E_{T_o}} w(u, v)$. 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 $W(T_o)$ is no higher than θ:*

$$Pr(\frac{W(T_o)-W(T)}{W(T_o)} \leq \theta) \geq 1-\delta. \qquad (5)$$

In this problem, both ϵ and δ 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_o, i.e., $W(T) \geq 99\% W(T_o)$. Note that the total weight of the approximate spanning tree can not exceed $W(T_o)$.

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 ($w(v',v)$, 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 [13, 23] 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

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

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 $w(v',v)$ is that $w(v',v)$ is the sum of $|V|$ individual quantities, $|(M(u,v') \odot \{\lambda(v',v)\}) \cap M(u,v)|$. This gives rise to this question: *assuming we have n samples $u_1, u_2, \cdots, u_n$, and for each sample vertex u_i, can we compute efficiently each individual term*

$$X_{e,i} = |(M(u_i,v') \odot \{\lambda(v',v)\}) \cap M(u_i,v)| \qquad (6)$$

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

$$X_e = w(v',v)/|V|, \text{where } e=(v',v) \in E(G). \qquad (7)$$

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 $w(v',v)$ as the population sum and the new quantity X_e is the population mean, where the weight of a quantity u in the population is $|(M(u,v') \odot \{\lambda(v',v)\}) \cap M(u,v)|$.

Given this, we define the sampling estimator of X_e as

$$\hat{X}_{e,n} = \frac{1}{n}\sum_{i=1}^{n} X_{e,i} \qquad (8)$$

Note that for each sample X_i, $E(X_{e,i}) = X_e$, and X_i is a bounded random variable, i.e., $X_i \in [0,R]$ where $R = \binom{|\Sigma|}{\lfloor |\Sigma|/2 \rfloor}$.

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 [14], the deviation (error bound) of the empirical mean from the true expectation (X_e) is: with probability of at least $1-\delta'$,

$$|\hat{X}_{e,n} - E(\hat{X}_{e,n})| = |\hat{X}_{e,n} - E(X_{e,i})| = |\hat{X}_{e,n} - X_e| \leq R\sqrt{\frac{\ln \frac{2}{\delta'}}{2n}}$$

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* [13, 23], which depends on only the empirical standard deviation:

$$\hat{\sigma}^2_{e,n} = \frac{1}{n}\sum_{i=1}^{n}(X_{e,i} - \hat{X}_{e,n})^2 = \frac{1}{n}(\sum_{i=1}^{n} X^2_{e,i} - \hat{X}^2_{e,n}) \qquad (9)$$

The combined Hoeffding and Bernstein bound is expressed as:

$$Pr(|\hat{X}_{e,n} - X_e| \leq \epsilon_{e,n}) \geq 1-\delta', \ \epsilon_{e,n} = \hat{\sigma}^2_{e,n}\sqrt{\frac{2\ln\frac{3}{\delta'}}{n}} + \frac{3R\ln\frac{3}{\delta'}}{n} \qquad (10)$$

Here, the dependency of the range R in the error term $\epsilon_{e,n}$ decreases linearly with respect to the sample size n. The error term also depends on $\frac{\hat{\sigma}^2_{e,n}}{\sqrt{n}}$. However, since the empirical standard deviation $\hat{\sigma}^2_{e,n}$ is typically much smaller than the range R, this bound is tighter than the Hoeffding bound. In our algorithm, we use the Hoeffding and Bernstein bound (10) as our error bound. Note that both the estimated edge weight $\hat{X}_{e,n}$ and the error bound $\epsilon_{e,n}$ can be easily incrementally maintained (without recording $X_{e,i}$ for each sample u_i and edge e). We simply record the following two sample statistics, $\sum_{i=1}^{n} X_{e,i}$ and $\sum_{i=1}^{n} X^2_{e,i}$. For the error bound determination (10), we need choose δ', which in our algorithm is specified as $\delta' = \frac{\delta}{|E|}$. By setting up this confidence level, we can easily derive the following observation:

Lemma 2. *Given n sample vertices and $\delta' = \frac{\delta}{|E|}$, with probability more than $1-\delta$, the deviation of each estimated edge weight $\hat{X}_{e,n}$ in the graph from their exact edge weight X_e is no greater than the error bound defined in (10):*

$$Pr(\bigwedge_{e \in E(G)} |\hat{X}_{e,n} - X_e| \leq \epsilon_{e,n}) \geq 1-\delta. \qquad (11)$$

This can be easily proven by the Bonferroni inequality. Lemma 2 also suggests a computation procedure we can use to estimate each edge weight. Basically, assuming we have n samples, $u_1, u_2, \cdots, u_n$, for each sample vertex u_i, we will compute the *single-source generalized transitive closure*, $M(u_i,v)$, for any vertex $v \in V$. From there, we can calculate $X_{e,i}$ for each edge e in the graph.

Approximate Maximal Spanning Tree Construction. Assuming we have n sample vertices, the above discussion describes that each edge in the graph is associated with two values, the estimated weight $\hat{X}_{e,n}$ and its error bound $\epsilon_{e,n}$. Given this, our algorithm then will discover two maximal spanning trees in the db-graph G, tree T for the estimated edge weight $\hat{X}_{e,n}$ assignment and tree T' for the error bound $\epsilon_{e,n}$ assignment. *Under what condition, can we tell that tree T would be the desired tree which meet the criterion (5)?* Theorem 4 provides a positive answer to this question.

THEOREM 4. *Given n samples, let T be the maximal spanning tree of G where each edge e has weight $\hat{X}_{e,n}$, and let T' be the maximal spanning tree of G where each edge has weight $\epsilon_{e,n}$. We denote $W_n(T) = \sum_{e\in T}\hat{X}_{e,n}$ and $\Delta_n(T') = \sum_{e'\in T'}\epsilon_{e',n}$. Then, if T satisfies (12), then T is our approximate tree:*

$$\frac{2\Delta_n(T')}{W_n(T)-\Delta_n(T')} \le \theta, \text{ where } W_n(T) \ge \Delta_n(T'), \tag{12}$$

$$\Longrightarrow Pr(\frac{W(T_o)-W(T)}{W(T_o)} \le \theta) \ge 1-\delta.$$

Proof Sketch: First, we have

$$Pr(|W_n(T)-W(T)| \le \Delta_n(T')) \ge 1-\delta$$

This is because with probability of at least $1-\delta$,

$$|W_n(T)-E(W_n(T))| = |\sum_{e\in T}\hat{X}_{e,n} - \sum_{e\in T}E(\hat{X}_{e,n})| =$$

$$|\sum_{e\in T}(\hat{X}_{e,n}-X_e)| = |W_n(T)-W(T)| \le \sum_{e\in T}\epsilon_{e,n}(Lemma\ 2)$$

$$\le \Delta_n(T')(\Delta_n(T') \text{ is the maximal total error of any tree in } G)$$

Similarly, from Lemma 2, this also holds:

$$Pr(|W_n(T_o)-W(T_o)| \le \Delta_n(T')) \ge 1-\delta.$$

In addition, by definition, we have

$$W_n(T_o) \le W_n(T) \text{ and } W(T_o) \ge W(T).$$

Putting them together, we have

$$Pr(W_n(T)-\Delta_n(T') \le W(T) \le W(T_o) \le W_n(T)+\Delta_n(T')) \ge 1-\delta.$$

Thus, the following hold:

$$Pr(W(T_o)-W(T) \le 2\Delta_n(T')) \ge 1-\delta \tag{13}$$
$$Pr(W_n(T)-\Delta_n(T') \le W(T_o)) \ge 1-\delta \tag{14}$$

Finally, if $\frac{2\Delta_n(T')}{W_n(T)-\Delta_n(T')} \le \theta(12)$, with (13) and (14) $\Longrightarrow$

with probability of at least $1-\delta$,

$$\frac{W(T_o)-W(T)}{W(T_o)} \le \frac{2\Delta_n(T')}{W_n(T)-\Delta_n(T')} \le \theta \quad \square$$

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_0 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.

Algorithm 2 Hoeffding-Bernstein-Tree($G(V,E,\Sigma,\lambda)$)

```
1: $n \leftarrow 0$ {$n_0$ is the initial sample size}
2: repeat
3:   $S \leftarrow SampleVertices(V, n_0)$ {sample $n_0$ vertices in $V$}
4:   $n \leftarrow n + n_0$ {$n$ is the total number of samples}
5:   for each $u_i \in S$ do
6:     $M_i \leftarrow SingleSourcePathLabel(G, u)$
7:     for each $e \in E$ {$e = (v', v)$} do
8:       $X_{e,i} \leftarrow |(M_i[v'] \odot \{\lambda(v', v)\}) \cap M_i[v]|$ {Formula (6)}
9:       Update $\sum_{i=1}^n X_{e,i}$ and $\sum_{i=1}^n X_{e,i}^2$
10:    end for
11:  end for
12:  for each $e \in E$ {$e = (v', v)$} do
13:    $\hat{X}_{e,n} \leftarrow \frac{1}{n}\sum_{i=1}^n X_{e,i}$ {edge weight estimator (8)}
14:    $\epsilon_{e,n} \leftarrow \hat{\sigma}^2_{e,n}\sqrt{\frac{2\ln\frac{3}{\delta'}}{n}} + \frac{3R\ln\frac{3}{\delta'}}{n}$ {error bound (10)}
15:  end for
16:  $T \leftarrow MST(G, [\hat{X}_{e,n}])$ {Maximal Spanning Tree with $\hat{X}_{e,n}$ as edge weight}
17:  $T' \leftarrow MST(G, [\epsilon_{e,n}])$ {$\epsilon_{e,n}$ as edge weight}
18: until Condition (12)=true
Procedure SingleSourcePathLabel($G(V,E,\Sigma,\lambda)$, $u$)
19: $M \leftarrow \varnothing$; $V_1 \leftarrow \{u\}$;
20: while $V_1 \neq \emptyset$ do
21:   $V_2 \leftarrow \varnothing$;
22:   for each $v \in V_1$ do
23:     for each $v' \in N(v)\{v' \in N(v) : (v, v') \in E\}$ do
24:       $New \leftarrow$ Prune($M[v'] \bigcup M[v] \odot \{\lambda(v, v')\}$ );
25:       if $New \neq M[v']$ then
26:         $M[v'] \leftarrow New$; $V_2 \leftarrow V_2 \cup \{v'\}$;
27:       end if
28:     end for
29:   end for
30:   $V_1 \leftarrow V_2$;
31: end while
```

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 [10]. 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 $O(|V||E|\binom{|\Sigma|}{|\Sigma|/2})$.

Given this, we can see that the total computational complexity for *Hoeffding-Bernstein-Tree* is $O(n|V||E|\binom{|\Sigma|}{|\Sigma|/2} + n/n_0(|E| + |V|\log|V|))$. 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 *SingleSourcePathLabel* for only a single vertex. We note that this algorithm would work well when the total label size $|\Sigma|$ is small. Since the sampling error bound is in the linear order of $\binom{|\Sigma|}{2}$, 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 $\sum_{i=1}^{n} X_{e,i}$. 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 *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 *SingleSourcePathLabel*, we will maintain three path-label sets based on Definition 4: 1) $M_s[v]$ corresponds to $M_s(u,v)$ which records all the path-labels from u to v with first edge being a tree-edge; 2) $M_e[v]$, corresponds to $M_e(u,v) - M_s(u,v)$, 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 *SingleSourcePathLabel* algorithm. Basically, we can compute the NT for each vertex in turn, and it has the same computational complexity as the *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 $u' \in Succ(u)$, 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 $v' \in Pred(v)$, 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 $u' \in Succ(u)$ and $v' \in Pred(v)$, 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'), $u' \in Succ(u)$ and $v' \in Pred(v)$, such that $NT(u',v') \neq \emptyset$.*

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

Our new approach utilizes the interval-labeling scheme [12] 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: $[pre(u), index(u)]$, where $pre(u)$ is u's *preorder* number and $index(u)$ 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 $[pre(v), index(v)] \subseteq [pre(u), index(u)]$* [12].

The key idea of our new approach is to utilize the spanning tree to transform this search problem into a geometric *range-search* problem. 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 ($pre(u')$, $index(u')$, $pre(v')$, $index(u')$). Then, for any two query vertices u and v, the vertex pair (u',v'), $u' \in Succ(u)$ and $v' \in Pred(v)$, such that $NT(u',v') \neq \emptyset$, corresponds to a four-dimensional point ($pre(u')$, $index(u')$, $pre(v')$, $index(v')$) in the range: $[pre(u), index(u)]$, $[pre(u), index(u)]$, $[1, pre(v)]$, $[index(v), |V|]$ of a four-dimensional space.*

Proof Sketch: We will show that for any $NT(u',v') \neq \emptyset$, $(pre(u'), index(u'), pre(v'), index(v'))$ is in the range, $pre(u') \in [pre(u),$
$index(u)]$, $index(u') \in [pre(u), index(u)]$, $pre(v') \in [1, pre(v)]$, $index(v') \in [index(v), |V|]$ iff $u' \in Succ(u)$ and $v' \in Pred(v)$.

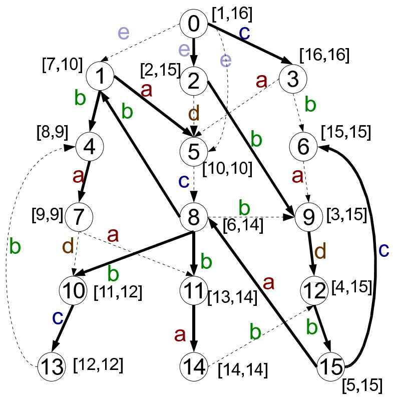


Figure 6: Interval Labeling for Spanning Tree

S.	T.	NT	Coordinate	S.	T.	NT	Coordinate
0	1	{e}	(1,16,7,10)	5	12	{c,b,a}	(10,10,4,15)
0	5	{e}	(1,16,10,10)	6	9	{a}	(15,15,3,15)
0	8	{e,c}	(1,16,6,14)	7	4	{c,b,d}	(9,9,8,9)
0	12	{e,b,a}	(1,16,4,15)	7	9	{b,a}	(9,9,3,15)
2	5	{d}	(2,15,10,10)	7	10	{d}	(9,9,11,12)
2	8	{c,d}	(2,15,6,14)	7	11	{a}	(9,9,13,14)
3	5	{a}	(16,16,10,10)	7	12	{b,a}	(9,9,4,15)
3	6	{b}	(16,16,15,15)	8	9	{b}	(6,14,3,15)
3	8	{c,a}	(16,16,6,14)	13	4	{b}	(12,12,8,9)
3	9	{b,a}	(16,16,3,15)	13	9	{b,a}	(12,12,3,15)
3	12	{c,b,a}	(16,16,4,15)	13	12	{b,a}	(12,12,4,15)
5	8	{c}	(10,10,6,14)	14	9	{b,a}	(14,14,3,15)
5	9	{c,b}	(10,10,3,15)	14	12	{b}	(14,14,4,15)

Figure 7: Coordinates in 4-dimensions

$$pre(u) \le pre(u') \le index(u) \text{ and}$$
$$pre(u) \le index(u') \le index(u) \iff$$
$$[pre(u'), index(u')] \subseteq [pre(u), index(u)] \iff u' \in Succ(u)$$
$$1 \le pre(v') \le pre(v) \text{ and}$$
$$index(v) \le index(v') \le |V| \iff$$
$$[pre(v'), index(v')] \supseteq [pre(v), index(v)] \iff v' \in Pred(v)$$

$\square$

Figure 7 give the 4-dimension coordinates $(pre(u'), index(u'), pre(v'), index(v'))$ 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 $u' \in Succ(u)$ and $v' \in Succ(v)$. It is easy to verify $(pre(u'), index(u'), pre(v'), index(v'))$, which is $(14, 14, 4, 15)$ within the range $[pre(u), index(u)]$, $[pre(u), index(u)]$, $[1, pre(v)]$, $[index(v), |V|]$.

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 $O(n \log^3 d)$ time and answers the query with $O(\log^4 n + k)$ 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, $x \in Pred(y)$. Our goal is to compute the path-label set of the in-tree path from x to y, $L(P_T(x, y))$, very fast. We build a histogram for each vertex in tree T. Let r 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., $L(P_T(r, u))$, 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 $(u \xrightarrow{A} v)$, in the first step (Line 1), we search for all the vertex pairs (u', v'), where $u' \in Succ(u)$ and $v' \in Pred(v)$, 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 $Succ(u')$ (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 $Pred(v')$ (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: $u \xrightarrow{A} v$?
 1: Finding all $(u', v')$, $u' \in Succ(u) \wedge v' \in Pred(v)$, such that
    $NT(u', v') \neq \emptyset$, and store them in a list $L$
 2: while $L \neq \emptyset$ do
 3:   $(u', v') \leftarrow L.top()$
 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 $v'' \in Pred(v')$
11:     end if
12:   else
13:     delete in $L$ those $(u'', v'')$ with $u'' \in Succ(u')$
14:   end if
15: end while
16: return false
```

The computational complexity of this procedure is $O(logn + \sum_{i=1}^{k} |NT(u_i, v_i)|)$, 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_i, v_i). 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 entires 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 Experimental 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 [18] 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 possbile 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 $\alpha = 2$ [24]. In addition, for the Sampling-Tree, we set the relative error bound $\theta = 1\%$ and user confidence level parameter $\delta = 1\%$.

6.2 Experimental 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.

(a) Expr1.a: Density $D(|E|/|V|)$ from 1.5 to 5;

	Opt-Tree	Sampling-Tree		Error
D	O-W	S-S	S-W	Error
1.5	5040719	800	5040196	0.01%
2	15904544	600	15904427	0.00%
2.5	18437857	500	18437795	0.00%
3	22010168	500	22010156	0.00%
3.5	22762424	500	22762417	0.00%
4	23936508	500	23931612	0.02%
4.5	24142458	500	24142457	0.00%
5	24690876	500	24690876	0.00%

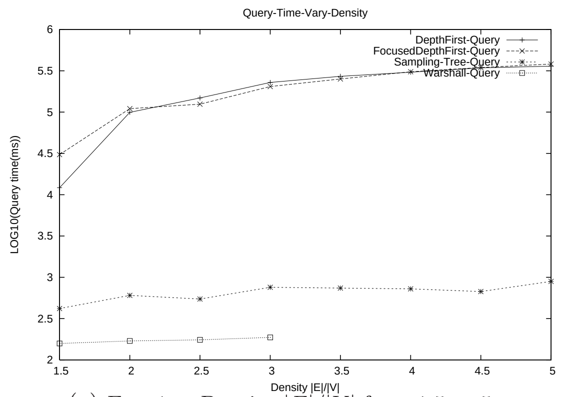
(b) Expr1.a: Density $(|E|/|V|)$ from 1.5 to 5;

	Sampling-Tree		Warshall	
D	I-S	C-T	I-S	C-T
1.5	452	187	197053	2172
2	599	605	621623	62728
2.5	424	773	720412	109864
3	631	1088	860334	226615
3.5	820	1220	-	-
4	480	1463	-	-
4.5	393	1626	-	-
5	834	1881	-	-

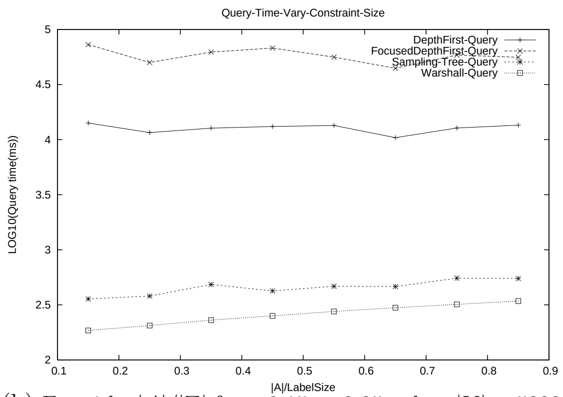
(c) Expr1.b: $|A|/|\Sigma|$ from 0.15 to 0.85

	Sampling-Tree		Warshall	
C	I-S	C-T	I-S	C-T
0.15	375	221	219987	3930
0.25	394	197	195117	2749
0.35	594	225	207304	3025
0.45	478	214	201410	3500
0.55	412	224	218034	3987
0.65	414	198	176274	1963
0.75	530	194	197036	2693
0.85	471	207	209504	3586

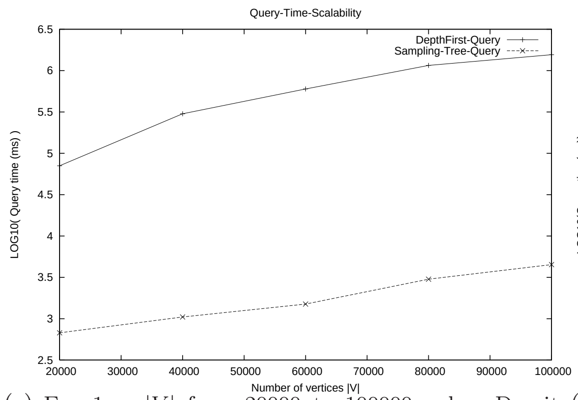
Table 1: I-S(Index Size in KB), C-T(Construction Time in s), O-W(Optimal Weight), S-W(Sampling Weight), S-S(Sampling-Size)



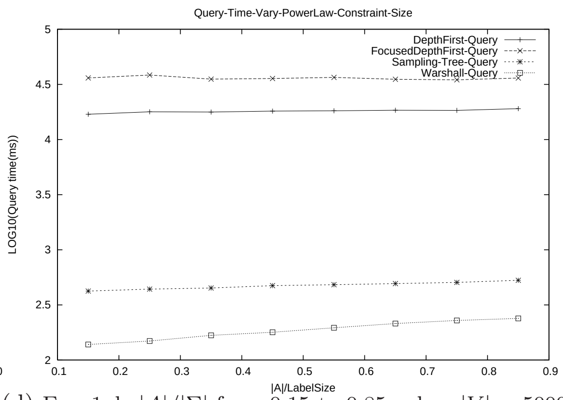
(a) Expr1.a: Density $|E|/|V|$ from 1.5 to 5;



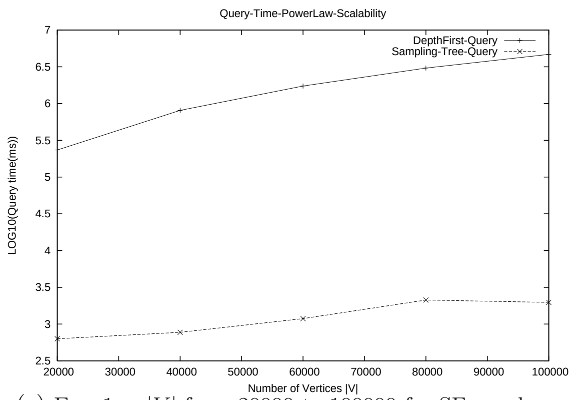
(b) Expr1.b: $|A|/|\Sigma|$ from 0.15 to 0.85, when $|V| = 5000$;



(c) Expr1.c: $|V|$ from 20000 to 100000, when Density $|E|/|V| = 1.5$;



(d) Expr1.d: $|A|/|\Sigma|$ from 0.15 to 0.85, when $|V| = 5000$ for SF graphs;



(e) Expr1.e: $|V|$ from 20000 to 100000 for SF graphs;

Figure 8: Experimental Results on Synthetic Datasets

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 $|E|/|V|$ from 3.5 to 5 is about 10 times larger than for $|E|/|V| = 1.5$, Warshall completely fails to work under high density. In addition, the query time of Sampling Tree remains almost unchanged with the increase of $|E|/|V|$ as shown in Figure 8(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 8(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 payoff 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 8(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.

	Sampling-Tree			Warshall	Opt-Tree
$\|V\|$	I-S	C-T	S-S	C-T	C-T
20000	2990	3606	800	-	-
40000	7766	16754	800	-	-
60000	18590	49756	800	-	-
80000	32806	217473	800	-	-
100000	53451	998711	200	-	-

Table 2: Expr1.c: $|V|$ from 20000 to 100000, when Density $|E|/|V| = 1.5$; I-S(Index Size in KB), C-T(Construction Time in s), S-S(Sampling-Size)

(Expr1.d) Varying Query Constraint Size on SF Graphs: In this experiment, we fix $|V| = 5000$, for SF graphs based on RMAT algorithm, then change query label constraints $|A|$ from 15% to 85% of the total edge labels to compare how these approaches differ from each other. The query time for DFS is 35 to 40 times Sampling-Tree's when increasing $|A|$, and the query time for Focused DFS is even 151 to 261 times Sampling-Tree's, as shown in Figure 8(d). Expr1.d shows that Sampling-Tree can perform well when increasing the query constraint size for SF graphs.

(Expr1.e) Scalability on SF Graphs: In this experiment, we generate SF graphs based on RMAT algorithm and vary the graph size $|V|$ from $20,000$ to $100,000$. Similar to Expr1.c, Focused DFS, Warshall and Opt-Tree fail on our machine when the number of vertices exceeds $20,000$. The query time for DFS is 370 to 2365 times that of Sampling-Tree as shown in Figure 8(e). This shows that Sampling-Tree scales well when increasing the SF graph size.

To summarize, in our experimental results for query time, Sampling-Tree is much faster than DFS, Focused DFS and very close to Warshall, which has the expected fastest query time. On average, the Sampling-Tree is within three times the query time for Warshall and 20 to 400 times faster than DFS, even 2365 times faster than Focused DFS. The index size for Sampling-Tree is much smaller than Warshall's, ranging from 0.1% to 0.3% of the generalized transitive closure. These results show the Sampling-Tree algorithm is quite successful in efficiently answering label constraint reachability queries with very compact index size for both ER graphs and SF graphs.

6.3 Experimental Results on Real Datasets

In this experiment, we evaluate different approaches on two real graph datasets: the first one is an integrated biological network for Yeast [19, 15], and the second one comes from the semantic knowledge network, Yago [25]. The Yeast graph contains 3063 vertices (genes) with density 2.4. It has 5 labels, which corresponds to different type of interactions, such as protein-DNA and protein-protein interaction. Since the protein-protein interaction is undirected, we transform it into two directed edges. In the experimental study, we use the Sampling-Tree to build the index and then we vary the size of the label-constraint $|A|$ from 2 to 5 to study the query time. Compared to DFS, Sampling-Tree is on average 2 times faster on the yeast graph. The second Yago graph contains 5000 vertices 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 10 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

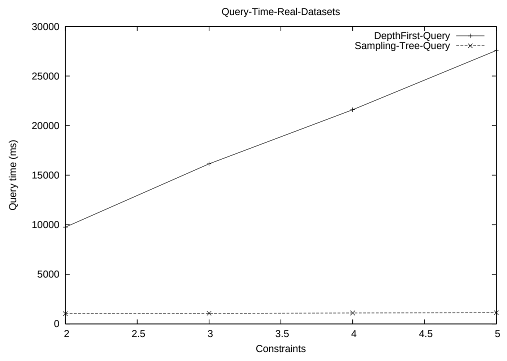


Figure 9: Expr2.b: varying $|A|$ in Yeast

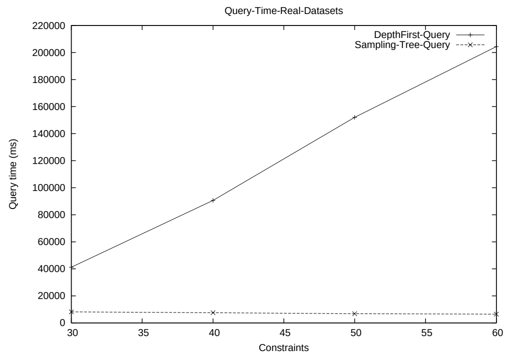


Figure 10: Expr2.a: varying $|A|$ in Yago

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

[1] Serge Abiteboul and Victor Vianu. Regular path queries with constraints. In *PODS*, pages 122–133, 1997.

[2] R. Agrawal, A. Borgida, and H. V. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In *SIGMOD*, pages 253–262, 1989.

[3] Ian Anderson. *Combinatorics of Finite Sets*. Clarendon Press, Oxford, 1987.

[4] A. L. Barabasi and R. Albert. Emergence of scaling in random networks. *Science (New York, N.Y.)*, 286, 1999.

[5] M.de Berg, M.van Kreveld, M.Overmars, and O.Schwarzkopf. *Computational Geometry*. Springer, 2000.

[6] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-mat: A recursive model for graph mining. In *Fourth SIAM International Conference on Data Mining*, 2004.

[7] Li Chen, Amarnath Gupta, and M. Erdem Kurul. Stack-based algorithms for pattern matching on dags. In *VLDB '05*.

[8] Jiefeng Cheng, Jeffrey Xu Yu, Xuemin Lin, Haixun Wang, and Philip S. Yu. Fast computation of reachability labeling for large graphs. In *EDBT*, pages 961–979, 2006.

[9] Y. J. Chu and T. H. Liu. On the shortest arborescence of a directed graph. *Science Sinica*, 14:1396–1400, 1965.

[10] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. McGraw Hill, 1990.

[11] J. Edmonds. Optimum branchings. *J. Research of the National Bureau of Standards*, 71B:233–240, 1967.

[12] Gang Gou and Rada Chirkova. Efficiently querying large xml data repositories: A survey. *IEEE Trans. Knowl. Data Eng.*, 19(10):1381–1403, 2007.

[13] Verena Heidrich-Meisner and Christian Igel. Hoeffding and bernstein races for selecting policies in evolutionary direct policy search. In *ICML '09*.

[14] W. Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30, March 1963.

[15] Thorsson V Ranish JA Christmas R Buhler J Eng JK Bumgarner R Goodlett DR Aebersold R Hood L. Ideker, T. Integrated genomic and proteomic analyses of a systematically perturbed metabolic network. In *Science*, pages 929–934, 2001.

[16] H. V. Jagadish. A compression technique to materialize transitive closure. *ACM Trans. Database*

Syst., 15(4):558–598, 1990.

[17] Ruoming Jin, Yang Xiang, Ning Ruan, and Haixun Wang. Efficiently answering reachability queries on very large directed graphs. In *SIGMOD Conference*, pages 595–608, 2008.

[18] Richard Johnsonbaugh and Martin Kalin. A graph generation software package. In *SIGCSE*, pages 151–154, 1991.

[19] Rinaldi N.J. Robert F. Odom D.T. Bar-Joseph Z. Gerber G.K. Hannett N.M. Harbison C.R. Thompson C.M. Simon I. Zeitlinger J. Jennings E.G. Murray H.L. Gordon D.B. Ren B. Wyrick J.J. Tagne J. Volkert T.L. Fraenkel E. Gifford D.K. Lee, T.I. and R.A. Young. Transcriptional regulatory networks in saccharomyces cerevisiae. In *Science*, pages 799–804, 2002.

[20] Jure Leskovec, Ajit Singh, and Jon M. Kleinberg. Patterns of influence in a recommendation network. In *PAKDD '06*.

[21] Alberto O. Mendelzon and Peter T. Wood. Finding regular simple paths in graph databases. *SIAM J. Comput.*, 24(6), 1995.

[22] Tova Milo and Dan Suciu. Index structures for path expressions. In *ICDT*, pages 277–295, 1999.

[23] Volodymyr Mnih, Csaba Szepesvári, and Jean-Yves Audibert. Empirical bernstein stopping. In *ICML '08*.

[24] M. E. J. Newman. Power laws, pareto distributions and zipf's law. *Contemporary Physics*, 46:323, 2005.

[25] Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. Yago: a core of semantic knowledge. In *WWW*, pages 697–706, 2007.

[26] Silke Trißl and Ulf Leser. Fast and practical indexing and querying of very large graphs. In *SIGMOD '07*.

[27] Haixun Wang, Hao He, Jun Yang, Philip S. Yu, and Jeffrey Xu Yu. Dual labeling: Answering graph reachability queries in constant time. In *ICDE '06*.