Reachability Labeling for Distributed Graphs

Junhua Zhang[♮], Wentao Li[♮], Lu Qin[♮], Ying Zhang[♮], Dong Wen[§], Lizhen Cui[‡], and Xuemin Lin[§]
[♮]*AAII, FEIT, University of Technology Sydney;* [§]*The University of New South Wales;* [‡]*Shandong University*
junhua.zhang@student.uts.edu.au; {wentao.li, lu.qin, ying.zhang}@uts.edu.au
dong.wen@unsw.edu.au; lxue@cse.unsw.edu.au; clz@sdu.edu.cn

***Abstract*—Real-world graphs are typically distributed across multiple data centers. When performing reachability queries on these distributed graphs, reachability labeling methods ensure fast query processing by using lightweight indexes. One of the best-known labeling methods is TOL; however, TOL is a *serial* algorithm and cannot handle distributed graphs. The main goal of this paper is to design new labeling methods that can work in parallel while producing the *same* index as TOL. To this end, we investigate the limitation of TOL and thus propose a filtering-and-refinement framework for index creation. This framework first obtains a super-set of each vertex's label sets and then eliminates the invalid elements. Based on this framework, we design distributed labeling algorithms and then use batch processing to improve efficiency. Experimental results on real-world graphs show that the proposed algorithms can index distributed graphs efficiently.**

I. Introduction

Graphs are used widely to represent real-world entities and their relationships [1]. As a common graph operation, reachability query $q(s,t)$ asks whether there is a path from vertex s to vertex t. Reachability query plays an important role in applications such as geographic navigation, Internet routing, and ontology reasoning [2]. Moreover, it is a building block for fields such as social sciences, computational biology, and software engineering [3].

Due to the importance of reachability query processing, many approaches have been proposed. These approaches are usually classified into three categories [2]: 1) index-free approaches that use the online search [4] (e.g., breadth-first search or depth-first search) on graphs to determine whether one vertex can reach another [5]; 2) index-assisted approaches that accelerate the online search using auxiliary information [6]–[9]; 3) index-only approaches that *avoid* the online search using an offline index [10]–[14].

Most current approaches are centralized: they assume that graphs reside in the main memory [15]. However, as the graph size increases, real-world graphs are typically distributed in multiple data centers [16]. When performing reachability queries on distributed graphs, the pioneer work [15], [16] implements the online search in a distributed manner for query processing. However, the query latency can be high due to the need to access distributed graphs during the query. This makes methods that require the online search (i.e., index-free or index-assisted approaches) undesirable, especially when a large number of queries need to be processed.

To speed up query processing on distributed graphs, an alternative idea is to use index-only approaches — the index created offline eliminates the dependence on the original graph during querying [14]. Specifically, consider a graph $G(V,E)$ with vertex set V and edge set E, index-only approaches create an index that assigns an out-label set $\mathsf{L}_{\mathsf{out}}(v)$ and an in-label set $\mathsf{L}_{\mathsf{in}}(v)$ for each vertex $v \in V$. The out-label set $\mathsf{L}_{\mathsf{out}}(v)$ of v contains vertices that v can reach, while the in-label set $\mathsf{L}_{\mathsf{in}}(v)$ contains vertices that can reach v. To answer the query $q(s,t)$ between vertices s and t, we *only* need to check whether there exists a common vertex w between $\mathsf{L}_{\mathsf{out}}(s)$ and $\mathsf{L}_{\mathsf{in}}(t)$: the existence of w means that s can reach t via w.

The state-of-the-art index-only methods is Total Order Labeling (TOL) [14]. TOL assigns an order value to each vertex in a graph, and then selects vertices for labeling in a decreasing sequence of order values. When labeling a vertex v, TOL tries to add v to the in-label/out-label sets of other vertices, using a **pruning operation**. When all vertices have finished labeling, the in-label/out-label sets generated by TOL for each vertex are used as an index.

The pruning operation of TOL is a double-edged sword: TOL reduces the index size by eliminating redundancy through the pruning operation; however, each vertex v needs to wait for vertices whose order values are higher than v to finish labeling, thus making the pruning operation of v feasible. This means that the execution of TOL is inherently serial. In other words, TOL *does not work properly on distributed graphs*.

On the other hand, for a distributed graph, the index created by TOL can efficiently support queries. This is because the index of TOL is small enough that we can put it on a single machine to achieve fast in-memory queries. For example, the index size for graph SK (see Table V for graph's details) with billions of edges is bounded by 1 GB. The main purpose of our paper is to design new labeling methods to handle a distributed graph while obtaining the same index as TOL.

For this purpose, we delve into the labeling process of TOL. We find that when labeling a vertex v, v joins the label sets of some specific vertices. These vertices are defined as the **backward label set** of v. The working process of TOL can be equated to finding the backward label set for each vertex v. To find v's backward label set, we use a filtering-and-refinement framework, thereby avoiding the pruning operation used by TOL. Specifically, we first generate a super-set of the backward label set of v as candidates, and then remove invalid elements from candidates to obtain the actual backward label set. This novel framework gets the same index as TOL while letting all vertices run in parallel. Based on this framework, we design efficient labeling algorithms and provide distributed implementations. In addition, we split vertices into batches for labeling to further improve efficiency.

The contributions of this paper are summarized as follows.

- *Analysis of* TOL*'s limitation (Section II).* We investigate TOL's limitation, i.e., the pruning operation of TOL makes parallel work challenging. This motivates the design of new labeling methods.
- *Novel labeling algorithms (Section III).* We find that each vertex's labeling process can be replaced by finding the backward label set for that vertex. We use a filtering-and-refinement

Wentao Li is the corresponding author.

framework to find the backward label set of each vertex in parallel. Using this framework, we propose new labeling algorithms and provide implementations in a distributed system.

- *Batch labeling optimization (Section IV).* To further improve the labeling algorithms' efficiency, we split vertices into batches and then construct the index in batches.
- *Extensive empirical studies (Section VI).* We conduct numerous experiments to validate the efficiency of the proposed labeling algorithms. On medium-sized graphs, our algorithms can outperform TOL by nearly an order of magnitude. Furthermore, on billion-sized graphs, we can create indexes in half an hour while TOL cannot.

II. Preliminary

We first introduce some notations in Section II-A, then give the labeling algorithm TOL in Section II-B, followed by the problem statement in Section II-C. For ease of understanding, Table I lists frequently used notations.

A. Notations

Given a directed graph $G = (V, E)$ with $n = |V|$ vertices and $m = |E|$ edges, we define the in-neighbor set $N^G_{in}(v)$ of a vertex $v \in V$ as $N^G_{in}(v) = \{u|(u, v) \in E\}$; the out-neighbor set $N^G_{out}(v)$ as $N^G_{out}(v) = \{u|(v, u) \in E\}$. The **in-degree** $d^G_{in}(v)$ (resp. the **out-degree** $d^G_{out}(v)$) of v is the size of its in-neighbor set (resp. out-neighbor set), i.e., $d^G_{in}(v) = |N^G_{in}(v)|$ (resp. $d^G_{out}(v) = |N^G_{out}(v)|$). The path between a vertex pair $s, t \in V$ is defined as $p^G(s, t) = (v_1 = s, v_2, \cdots, v_l = t)$, where $(v_i, v_{i+1}) \in E$, for $\forall i \in [1, l-1]$. If there exists a path between s and t, then s can reach t (denoted as $s \to t$).

Definition 1. The **ancestors** $\mathsf{ANC}(v)$ (resp. **descendants** $\mathsf{DES}(v)$) of a vertex $v \in V$ contain all the vertices that can reach v (resp. that v can reach). Vertex v is contained both in $\mathsf{ANC}(v)$ and $\mathsf{DES}(v)$.

If the context is obvious, we drop G from notations. The **inverse graph** $\overline{G} = (V, \overline{E})$ of a graph $G(V, E)$ contains the same vertices but with all edges reversed in direction (i.e., $\overline{E} = \{(v, u)|(u, v) \in E(G)\}$).

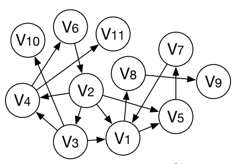


Fig. 1: Graph G

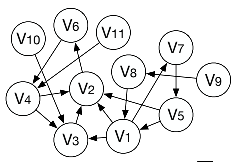


Fig. 2: Inverse Graph $\overline{G}$

Example 1. Consider the graph G in Fig. 1, which has 11 vertices and 15 edges. For vertex v_2, $N_{in}(v_2) = \{v_6\}$ and $d_{in}(v_2) = 1$; $N_{out}(v_2) = \{v_1, v_3, v_4, v_5\}$ and $d_{out}(v_2) = 4$. The vertex v_2 can reach vertex v_7 since there exists a path from v_2 to v_7. For v_2, $\mathsf{ANC}(v_2) = \{v_2, v_3, v_4, v_6\}$ and $\mathsf{DES}(v_2) = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}, v_{11}\}$. The inverse graph $\overline{G}$ of G is shown in Fig. 2.

The in-label/out-label sets over all vertices form index L, which can be used to answer reachability queries in graph G.

Definition 2. The **in-label set** $\mathsf{L_{in}}(v)$ of v contains vertices that can reach v, i.e., $\mathsf{L_{in}}(v) \subseteq \mathsf{ANC}(v)$; the **out-label set** $\mathsf{L_{out}}(v)$ of v contains vertices that v can reach, i.e., $\mathsf{L_{out}}(v) \subseteq \mathsf{DES}(v)$.

TABLE I: Notations

Notation	Meaning
$G(V, E)$	graph
$\overline{G}(V, \overline{E})$	inverse graph
$d_{in}(v), d_{out}(v)$	in-degree, out-degree of v
$\mathsf{ANC}(v), \mathsf{DES}(v)$	ancestors, descendants of v
$\mathsf{L_{in}}(v)$, $\mathsf{L_{out}}(v)$	in-label, out-label sets of v
$\mathsf{L^-_{in}}(v)$, $\mathsf{L^-_{out}}(v)$	backward in-label, out-label sets of v
$\mathsf{DES_{hig}}(v)$	high-order descendants of v
$\mathsf{BFS_{low}}(v), \mathsf{BFS_{hig}}(v)$	low-order, high-order vertices in trimmed BFS
$\mathsf{IBFS_{low}}(v)$	inverted list of v
$[V_1, V_2, \ldots, V_g]$	batch sequence of G
$\mathsf{L^{V_i}_{in}}, \mathsf{L^{V_i}_{out}}$	batch in-label, out-label sets regarding V_i

TABLE II: The Index L

Vertex	$\mathsf{L_{in}}$	$\mathsf{L_{out}}$
v_1	$\{v_1\}$	$\{v_1\}$
v_2	$\{v_2\}$	$\{v_1, v_2\}$
v_3	$\{v_2\}$	$\{v_1, v_2\}$
v_4	$\{v_2\}$	$\{v_1, v_2\}$
v_5	$\{v_1\}$	$\{v_1\}$
v_6	$\{v_2\}$	$\{v_1, v_2\}$
v_7	$\{v_1\}$	$\{v_1\}$
v_8	$\{v_1, v_8\}$	$\{v_8\}$
v_9	$\{v_1, v_8, v_9\}$	$\{v_9\}$
v_{10}	$\{v_2, v_{10}\}$	$\{v_{10}\}$
v_{11}	$\{v_2, v_{11}\}$	$\{v_{11}\}$

Given an index L, the largest label size, denoted as Δ, is defined as $\Delta = \max_{v \in V}(\max(|\mathsf{L_{in}}(v)|, |\mathsf{L_{out}}(v)|))$. To answer the reachability query $q(s, t)$ between s, t, we check whether there are overlapping vertices between $\mathsf{L_{out}}(s)$ and $\mathsf{L_{in}}(t)$.

$$q(s, t) = \begin{cases} true, & \text{if } \mathsf{L_{out}}(s) \cap \mathsf{L_{in}}(t) \neq \varnothing; \\ false, & \text{otherwise.} \end{cases}$$

If the vertices in $\mathsf{L_{out}}(s)$ and $\mathsf{L_{in}}(t)$ are sorted by IDs, answering $q(s, t)$ takes $O(|\mathsf{L_{out}}(s)| + |\mathsf{L_{in}}(t)|)$ time [14]. To ensure that index L correctly answers all reachability queries on G, L needs to satisfy the cover constraint.

Definition 3 (Cover Constraint)**.** For $\forall s, t \in V$, $\mathsf{L_{out}}(s) \cap \mathsf{L_{in}}(t) \neq \varnothing$ if and only if ($\Leftrightarrow$) $s \to t$.

Example 2. Consider the graph G in Fig. 1, where Table II lists an index L for G. For v_2, $\mathsf{L_{out}}(v_2) = \{v_1, v_2\} \subseteq \mathsf{DES}(v_2)$; for v_3, $\mathsf{L_{in}}(v_3) = \{v_2\} \subseteq \mathsf{ANC}(v_3)$. Since $\mathsf{L_{out}}(v_2) \cap \mathsf{L_{in}}(v_3) = \{v_2\} \neq \emptyset$, the query $q(v_2, v_3)$ returns "true".

B. Total Order Labeling

Many labeling methods have been proposed to create reachability indexes for graphs [2], and one of the best-known is Total Order Labeling (TOL). TOL works for a total of n rounds, where one vertex is selected for labeling in each round. TOL gives each vertex v an order $\mathsf{ord}(v)$ and selects the vertex with the i-th largest order in round i. TOL uses degree to determine the order and vertex IDs to break the tie: we can define $\mathsf{ord}(v) = (d_{in}(v)+1) \cdot (d_{out}(v)+1) + \frac{ID(v)}{n+1}$ for a vertex v, where $ID(v)$ is the ID of v. There are other ways to define $\mathsf{ord}(v)$, but this way is cheap to calculate and works well in practice [3], [17].

Example 3. Consider the graph G in Fig. 1, which has $n = 11$ vertices. For v_1, $\mathsf{ord}(v_1) = (d_{in}(v_1) + 1) \cdot (d_{out}(v_1) + 1) + \frac{1}{12} = 12.08$; for v_{10}, $\mathsf{ord}(v_{10}) = (d_{in}(v_{10}) + 1) \cdot (d_{out}(v_{10}) + 1) + \frac{10}{12} = 2.83$. Thus, $\mathsf{ord}(v_1) > \mathsf{ord}(v_{10})$, which means that the order of v_1 is higher than v_{10}.

Algorithm 1: TOL

```
Input: Graph $G(V, E)$
Output: Index $L$
1  $\mathsf{L}^1_{\mathsf{in}}(v) \leftarrow \emptyset, \mathsf{L}^1_{\mathsf{out}}(v) \leftarrow \emptyset$, for each vertex $v \in V$;
2  $G_1 \leftarrow G$;
3  foreach $i \in [1, n]$ do
4      $v_i \leftarrow$ the vertex whose order is the $i$-th largest;
5      $\mathsf{DES}^{G_i}(v_i) \leftarrow$ a $v_i$-sourced BFS on $G_i$;
6      $\mathsf{ANC}^{G_i}(v_i) \leftarrow$ a $v_i$-sourced BFS on $\overline{G_i}$;
7      foreach $w \in \mathsf{DES}^{G_i}(v_i)$ do
           // pruning operation
8          if $\mathsf{L}^i_{\mathsf{out}}(v_i) \cap \mathsf{L}^i_{\mathsf{in}}(w) = \emptyset$ then
9              $\mathsf{L}^{i+1}_{\mathsf{in}}(w) \leftarrow \mathsf{L}^i_{\mathsf{in}}(w) \cup \{v_i\}$;
10     foreach $w \in \mathsf{ANC}^{G_i}(v_i)$ do
           // pruning operation
11         if $\mathsf{L}^i_{\mathsf{in}}(v_i) \cap \mathsf{L}^i_{\mathsf{out}}(w) = \emptyset$ then
12             $\mathsf{L}^{i+1}_{\mathsf{out}}(w) \leftarrow \mathsf{L}^i_{\mathsf{out}}(w) \cup \{v_i\}$;
13     $G_{i+1} \leftarrow G_i \setminus \{v_i\}$;
14 return $L = \{\mathsf{L}^{n+1}_{\mathsf{in}}(v) \cup \mathsf{L}^{n+1}_{\mathsf{out}}(v) | v \in V\}$;
```

TOL **Algorithm.** Algorithm 1 shows how TOL creates an index L for graph G. We first initialize the in-label set $\mathsf{L}^1_{\mathsf{in}}(v)$ and out-label set $\mathsf{L}^1_{\mathsf{in}}(v)$ of all vertices v to an empty set (Line 1), and then copy G to G_1 (Line 2). Here, $\mathsf{L}^i_{\mathsf{in}}(v)$ (resp. $\mathsf{L}^i_{\mathsf{out}}$) refers to the label sets created by vertices $[v_1, v_2, \cdots, v_{i-1}]$ of order higher than v_i. The labeling process works in n rounds (Line 3). In round i, the vertex v_i with the i-th largest order starts labeling (Line 4).

<u>*Labeling v_i (Line 5-11).*</u> First, the descendants $\mathsf{DES}^{G_i}(v_i)$ and ancestors $\mathsf{ANC}^{G_i}(v_i)$ of v_i are obtained using the v_i-sourced BFS in G_i and $\overline{G_i}$, respectively (Line 5-6). Then, v_i is added to in-label/out-label sets of other vertices when it passes a **pruning operation**: for each vertex $w \in \mathsf{DES}^{G_i}(v_i)$, when $\mathsf{L}^i_{\mathsf{out}}(v_i)$ and $\mathsf{L}^i_{\mathsf{in}}(w)$ have no overlapping, v_i is appended to $\mathsf{L}^i_{\mathsf{in}}(w)$ to form $\mathsf{L}^{i+1}_{\mathsf{in}}(w)$ (Line 8); For each vertex $w \in \mathsf{ANC}^{G_i}(v_i)$, when $\mathsf{L}^i_{\mathsf{in}}(v_i)$ and $\mathsf{L}^i_{\mathsf{out}}(w)$ have no overlapping, v_i is appended to $\mathsf{L}^i_{\mathsf{out}}(w)$ to form $\mathsf{L}^{i+1}_{\mathsf{out}}(w)$ (Line 10). Then, v_i and the incident edges are removed from G_i (denoted as $G_i \setminus v_i$) to form G_{i+1} for the next round (Line 11). After n rounds, the label sets of all vertices are returned as index L (Line 12).

Example 4. Consider the graph G in Fig. 1. We show how to create in-label sets for G, and out-label sets can be created similarly. In round 1, G is copied to G_1, and v_1 (with the highest order) is inserted into the in-label sets of its descendants $\mathsf{DES}^{G_1}(v_1) = \{v_1, v_5, v_7, v_8, v_9\}$ in G_1, as no pruning occurs. Then, v_1 and its adjacent edges are removed from G_1 to form G_2. In round 2, v_2 (with the second highest order) finds its descendants $\mathsf{DES}^{G_2}(v_2) = \{v_2, v_3, v_4, v_5, v_6, v_7, v_{10}, v_{11}\}$ in G_2. Then, v_2 performs a pruning operation (Line 8 of Algorithm 1) to test whether v_2 is added to the in-label sets of vertices in $\mathsf{DES}^{G_2}(v_2)$. For example, since $\mathsf{L}^2_{\mathsf{out}}(v_2) \cap \mathsf{L}^2_{\mathsf{in}}(v_5) = \{v_1\}$, v_2 is not inserted into $\mathsf{L}^3_{\mathsf{in}}(v_5)$ — pruning occurs. After pruning, v_2 is inserted into the in-label sets of $\{v_2, v_3, v_4, v_6, v_{10}, v_{11}\}$. After processing v_{11}, TOL ends.

Limitation. TOL is a centralized algorithm [14], which means that the graph needs to be stored on one machine for processing. To make matters worse, TOL is non-trivial to be parallelized. To illustrate why, we focus on the process of labeling v_i. When labeling v_i, each vertex w has an in-label set $\mathsf{L}^i_{\mathsf{in}}(w)$ and an out-label set $\mathsf{L}^i_{\mathsf{out}}(w)$ created by vertices $[v_1, v_2, \cdots, v_{i-1}]$ of order higher than v_i. We collect the label sets of all vertices w and get the index $L^i = \bigcup_{w \in V}\{\mathsf{L}^i_{\mathsf{in}}(w) \cup \mathsf{L}^i_{\mathsf{out}}(w)\}$ generated by vertices of order higher than v_i. The index L^i is necessary when labeling v_i: L^i determines whether or not v_i is added to the label set of another vertex $w \in V$.

Lemma 1. *For $\forall w \in V$, whether or not v_i is in the label set of w depends on L^i, specifically,*

- *$v_i \in \mathsf{L}_{\mathsf{in}}(w) \Leftrightarrow v_i \rightarrow w$, $\mathsf{L}^i_{\mathsf{out}}(v_i) \cap \mathsf{L}^i_{\mathsf{in}}(w) = \emptyset$;*
- *$v_i \in \mathsf{L}_{\mathsf{out}}(w) \Leftrightarrow w \rightarrow v_i$, $\mathsf{L}^i_{\mathsf{out}}(w) \cap \mathsf{L}^i_{\mathsf{in}}(v) = \emptyset$.*

Proof. See Appendix. □

Lemma 1 shows that L^i is essential for labeling v_i. However, L^i is generated only after vertices $[v_1, v_2, \cdots, v_{i-1}]$ of order higher than v_i have completed labeling. This suggests that labeling v_i *cannot* begin until those vertices with higher orders have finished labeling. Such a strong *order dependency* prevents TOL from being parallelized. Motivated by this, we aim to design novel labeling methods that can work in parallel while obtaining the same indexes as TOL.

Remark. [14] maintains TOL's index for dynamic graphs, but we try to generate the same indexes as TOL for distributed graphs. We consider maintaining indexes on distributed dynamic graphs as future work.

C. Problem Statement

We plan to use a **vertex-centric system** [18] to implement the proposed labeling methods. The vertex-centric system performs the tasks in a super-step fashion [19]. In each super-step, each active vertex v calls a user-defined function, compute(), to: 1) compute based on v's current state and the messages it received in the previous super-step; 2) update v's state; 3) send messages to other vertices (for the next super-step); and 4) (optionally) vote v to make it inactive. The whole computation terminates when there are no messages in the system, or all vertices become inactive. To avoid ambiguity, we use the term "vertex" to denote v as $v \in V$, and the term "node" to denote a computation unit in a cluster.

The problem addressed in this paper is:

> Given a distributed graph G, design reachability labeling methods and implement them using a vertex-centric system to create the index as TOL.

Throughout this paper, we do not assume that G is acyclic. This treatment is also used in [16], [20]. We treat it this way for two reasons: 1) our methods are general enough to handle both acyclic and non-acyclic graphs; 2) it is non-trivial to obtain and merge strongly connected components to make graphs acyclic in a distributed environment.

III. Distributed Reachability Labeling

We present the concept of backward label sets in Section III-A, and then propose a filtering-and-refinement framework to find backward label sets in Section III-B. New labeling algorithms based on this framework are designed in Section III-C, followed by the algorithm implementation in a distributed system in Section III-D.

A. TOL *Revisited*

As shown in Algorithm 1, TOL works in n rounds, where a vertex v is selected for labeling in each round. The process of labeling v is to use the pruning operation to determine some

TABLE III: The Backward Label Sets

Vertex	$\mathsf{L}^-_{\mathsf{in}}$	$\mathsf{L}^-_{\mathsf{out}}$
v_1	$\{v_1, v_5, v_7, v_8, v_9\}$	$\{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$
v_2	$\{v_2, v_3, v_4, v_6, v_{10}, v_{11}\}$	$\{v_2, v_3, v_4, v_6\}$
v_3	$\emptyset$	$\emptyset$
v_4	$\emptyset$	$\emptyset$
v_5	$\emptyset$	$\emptyset$
v_6	$\emptyset$	$\emptyset$
v_7	$\emptyset$	$\emptyset$
v_8	$\{v_8, v_9\}$	$\{v_8\}$
v_9	$\{v_9\}$	$\{v_9\}$
v_{10}	$\{v_{10}\}$	$\{v_{10}\}$
v_{11}	$\{v_{11}\}$	$\{v_{11}\}$

vertices (in v's descendants/ancestors) such that v is added to their label sets. We define these determined vertices as the backward label set of v.

Definition 4. Given $v \in V$ and index L of G, the **backward in-label set** of v is $\mathsf{L}^-_{\mathsf{in}}(v) = \{w | v \in \mathsf{L}_{\mathsf{in}}(w)\}$; the **backward out-label set** of v is $\mathsf{L}^-_{\mathsf{out}}(v) = \{w | v \in \mathsf{L}_{\mathsf{out}}(w)\}$.

Example 5. Consider the graph G in Fig. 1, where Table II shows the index L. In Table III we list the backward in-label/out-label sets for all vertices. For v_2, $\mathsf{L}^-_{\mathsf{in}}(v_2) = \{v_2, v_3, v_4, v_6, v_{10}, v_{11}\}$ since vertices $\{v_2, v_3, v_4, v_6, v_{10}, v_{11}\}$ contain v_2 in their in-label sets; For v_3, $\mathsf{L}^-_{\mathsf{in}}(v_3) = \emptyset$ since no vertex contains v_3 in its in-label set. Also, labeling v_2 is to add v_2 to the in-label sets of vertices in $\mathsf{L}^-_{\mathsf{in}}(v_2)$ and add v_2 to the out-label sets of vertices in $\mathsf{L}^-_{\mathsf{out}}(v_2)$.

We re-describe the working process of TOL from the perspective of backward label sets: TOL chooses a vertex v to label in each round. The process of labeling v is to determine the vertices in backward in-label/out-label sets where v joins their labels. Recall that TOL applies the pruning operation to determine its backward label sets. According to the analysis in Section II, the pruning operation causes TOL not to work in parallel. Therefore, the main contribution of our paper is to replace the pruning operation of TOL but still get the backward label sets of each vertex $v \in V$. This allows all vertices to work in parallel and get the same index as TOL.

Remark. Label sets and backward label sets are symmetric concepts — v is in $\mathsf{L}_{\mathsf{in}}(w)$ (resp. $\mathsf{L}_{\mathsf{out}}(w)$) implies that w is in $\mathsf{L}^-_{\mathsf{in}}(v)$ (resp. $\mathsf{L}^-_{\mathsf{out}}(v)$). For this reason, we aim to find the backward label sets $\mathsf{L}^-_{\mathsf{in}}(v)$ and $\mathsf{L}^-_{\mathsf{out}}(v)$ of each vertex v to create the index L. Also, since finding $\mathsf{L}^-_{\mathsf{out}}(v)$ on $\overline{G}$ is similar to finding $\mathsf{L}^-_{\mathsf{in}}(v)$ on G, we only discuss how to obtain $\mathsf{L}^-_{\mathsf{in}}(v)$ in the sequel. The discussions for $\mathsf{L}^-_{\mathsf{in}}(v)$ can be naturally extended to $\mathsf{L}^-_{\mathsf{out}}(v)$.

B. Filtering-and-refinement Framework

To determine the backward label set $\mathsf{L}^-_{\mathsf{in}}(v)$ for each vertex v without relying on the pruning operation of TOL, we give the condition for a certain vertex w to lie in $\mathsf{L}^-_{\mathsf{in}}(v)$.

Theorem 1. *For* $\forall v, w \in V$, $w \in \mathsf{L}^-_{\mathsf{in}}(v) \Leftrightarrow w \in \mathsf{DES}(v)$ *and* v *is the highest-order vertex on all paths from* v *to* w.

Proof. We prove $w \in \mathsf{L}^-_{\mathsf{in}}(v)$, or equivalently $v \in \mathsf{L}_{\mathsf{in}}(w)$.

- $\Leftarrow$: If v is the highest-order vertex on all paths from v to w, then there is no vertex u such that $v \to u \to w$ and $\mathsf{ord}(u) > \mathsf{ord}(v)$. Because TOL processes vertices in a non-increasing sequence of vertex order, this means at the moment v starts labeling: 1) $w \in \mathsf{DES}^{G_i}(v)$ since no vertex on a path from v to w can be removed before labeling v; 2) $\mathsf{L}_{\mathsf{out}}(v) \cap \mathsf{L}_{\mathsf{in}}(w) = \emptyset$ since no vertex on a path from v to w finishes labeling. Therefore, by Line 8 of Algorithm 1, $v \in \mathsf{L}_{\mathsf{in}}(w)$. With similar logic, we can prove that $v \in \mathsf{L}_{\mathsf{out}}(w)$ if v is the highest-order vertex on all paths from w to v.
- $\Rightarrow$: If $v \in \mathsf{L}_{\mathsf{in}}(w)$, let $u \neq v$ be the highest-order vertex on all paths from v to w. This means that u's order is the highest on all sub-paths from v to u and from u to w. We reuse the proof in $\Leftarrow$: 1) u's order is the highest on all u-w paths, then $u \in \mathsf{L}_{\mathsf{in}}(w)$; 2) u's order is the highest on all v-u paths, then $u \in \mathsf{L}_{\mathsf{out}}(v)$. Thus, $u \in \mathsf{L}_{\mathsf{out}}(v) \cap \mathsf{L}_{\mathsf{in}}(w) \neq \emptyset$ when labeling v. By Line 8 of Algorithm 1, $v \notin \mathsf{L}_{\mathsf{in}}(w)$, contradiction. □

Example 6. Consider the graph G in Fig. 1. For vertex v_2, $v_3 \in \mathsf{L}^-_{\mathsf{in}}(v_2)$ since v_2 has the highest order on all paths from v_2 to v_3; $v_5 \notin \mathsf{L}^-_{\mathsf{in}}(v_2)$ because v_1, with order higher than v_2, lies on a path from v_2 to v_5.

Theorem 1 paves the way for obtaining $\mathsf{L}^-_{\mathsf{in}}(v)$ of each vertex $v \in V$ in parallel. Specifically, by Theorem 1, $w \in \mathsf{DES}(v)$ is a necessary condition for $w \in \mathsf{L}^-_{\mathsf{in}}(v)$. In other words, $\mathsf{DES}(v)$ is a super-set of $\mathsf{L}^-_{\mathsf{in}}(v)$: $\mathsf{L}^-_{\mathsf{in}}(v) \subseteq \mathsf{DES}(v)$. To obtain $\mathsf{L}^-_{\mathsf{in}}(v)$, we need to remove invalid elements from $\mathsf{DES}(v)$. Thus, we define the higher-order descendants of v.

Definition 5. The **higher-order descendants** of v, denoted as $\mathsf{DES}_{\mathsf{hig}}(v)$, are vertices $u \in \mathsf{DES}(v)$ whose order is higher than v, that is, $\mathsf{DES}_{\mathsf{hig}}(v) = \{u | u \in \mathsf{DES}(v), \mathsf{ord}(u) > \mathsf{ord}(v)\}$.

Theorem 1 states that only when v is the highest-order vertex on all paths from v to w, then w is in $\mathsf{L}^-_{\mathsf{in}}(v)$. In other words, if there is a high-order vertex $u \in \mathsf{DES}_{\mathsf{hig}}(v)$ to reach w, w must not be in $\mathsf{L}^-_{\mathsf{in}}(v)$. Hence, we can use $\mathsf{DES}_{\mathsf{hig}}(v)$ to refine the super-set $\mathsf{DES}(v)$ by removing invalid elements.

Based on this idea, we propose the filtering-and-refinement framework to obtain $\mathsf{L}^-_{\mathsf{in}}(v)$: the filtering phase generates the super-set $\mathsf{DES}(v)$, and then invalid vertices that can be reached by vertices in $\mathsf{DES}_{\mathsf{hig}}(v)$ are removed for refinement. The correctness of this framework is given in Theorem 2.

Theorem 2. $\mathsf{L}^-_{\mathsf{in}}(v) = \mathsf{DES}(v) - \bigcup_{u \in \mathsf{DES}_{\mathsf{hig}}(v)} \mathsf{DES}(u)$.

Proof. We denote the right-hand side of the equation by RHS and verify that $\mathsf{L}^-_{\mathsf{in}}(v) = \mathsf{RHS}$.

- $\mathsf{L}^-_{\mathsf{in}}(v) \subseteq \mathsf{RHS}$: If $w \in \mathsf{L}^-_{\mathsf{in}}(v) \backslash \mathsf{RHS}$, then there are two possibilities: 1) $w \notin \mathsf{DES}(v)$, which contradicts $w \in \mathsf{L}^-_{\mathsf{in}}(v) \subseteq \mathsf{DES}(v)$; 2) $w \in \bigcup_{u \in \mathsf{DES}_{\mathsf{hig}}(v)} \mathsf{DES}(u)$, but by Theorem 1, $w \notin \mathsf{L}^-_{\mathsf{in}}(v)$, contradiction.
- $\mathsf{RHS} \subseteq \mathsf{L}^-_{\mathsf{in}}(v)$: If $w \in \mathsf{RHS}$, then there is no vertex u on any path from v to w for which $\mathsf{ord}(u) > \mathsf{ord}(v)$. By Theorem 1, $w \in \mathsf{L}^-_{\mathsf{in}}(v)$. □

Example 7. Consider the graph G in Fig. 1. We show how to obtain $\mathsf{L}^-_{\mathsf{in}}(v_3)$ of v_3. We first find $\mathsf{DES}(v_3) = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}, v_{11}\}$; next we get $\mathsf{DES}_{\mathsf{hig}}(v_3) = \{v_1, v_2\}$, and $\bigcup_{u \in \mathsf{DES}_{\mathsf{hig}}(v_3)} \mathsf{DES}(u) = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}, v_{11}\}$. Thus, $\mathsf{L}^-_{\mathsf{in}}(v_3) = \mathsf{DES}(v_3) - \bigcup_{u \in \mathsf{DES}_{\mathsf{hig}}(v_3)} \mathsf{DES}(u) = \emptyset$.

C. Two Labeling Methods

C-1. Basic Labeling Method

If we apply Theorem 2 to obtain $\mathsf{L}^-_{\mathsf{in}}(v)$ for each vertex $v \in V$, we need to perform a v-sourced breadth-first search (BFS) to

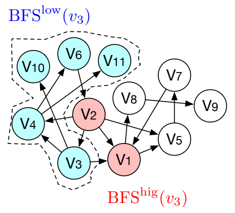

$\text{BFS}^{\text{low}}(v_3)$
V10
V6
V11
V7
V8
V9
V4
V2
V5
V3
V1
$\text{BFS}^{\text{hig}}(v_3)$

Fig. 3: The v_3-sourced Trimmed BFS

obtain $\mathsf{DES}(v)$ and $\mathsf{DES}_{\mathsf{hig}}(v)$ in the filtering phase, and then perform $|\mathsf{DES}_{\mathsf{hig}}(v)|$ BFSs, one BFS for one vertex in $\mathsf{DES}_{\mathsf{hig}}(v)$ in the refinement phase. The refinement phase requires a large number of BFSs, rendering this solution inefficient. To improve efficiency, we find that not all vertices in $\mathsf{DES}_{\mathsf{hig}}(v)$ are useful for refinement. For example, for vertices $a, b \in \mathsf{DES}_{\mathsf{hig}}(v)$, b is unnecessary when $\mathsf{DES}(b)$ is a subset of $\mathsf{DES}(a)$ — vertex a can reach all descendants of b.

So, how to identify unnecessary vertices in $\mathsf{DES}_{\mathsf{hig}}(v)$? A simple rule is that it is safe to delete vertex b if vertex $a \in \mathsf{DES}_{\mathsf{hig}}(v)$ can reach b: a can reach all descendants of b. Based on this rule, we propose to use a v-sourced BFS but block the expansion branch upon meeting a vertex $a \in \mathsf{DES}_{\mathsf{hig}}(v)$, thus implicitly deleting vertices $b \in \mathsf{DES}_{\mathsf{hig}}(v)$ that can be reached by a. We denote this BFS as a trimmed BFS.

Algorithm 2: Trimmed BFS

```
   Input: Graph $G(V,E)$, $v$
   Output: $\mathsf{BFS}_{\mathsf{low}}(v)$, $\mathsf{BFS}_{\mathsf{hig}}(v)$
1  queue $Q \leftarrow \emptyset$;
2  status$(u) \leftarrow \textcircled{?}$, for all vertices $u \in V$;
3  push $v \rightarrow Q$ and $\mathsf{BFS}_{\mathsf{low}}(v)$;
4  status$(v) \leftarrow \textcircled{\rightarrow}$;
5  while $Q$ is not empty do
6      $u \leftarrow$ pop from $Q$;
7      foreach $w \in N_{out}(u)$ do
8          if status$(w) \neq \textcircled{?}$ then continue;
9          if ord$(w) <$ ord$(v)$ then
10             status$(w) \leftarrow \textcircled{\rightarrow}$, push $w \rightarrow Q$ and $\mathsf{BFS}_{\mathsf{low}}(v)$;
11         else
               // block the expansion via $w$
12             push $w \rightarrow \mathsf{BFS}_{\mathsf{hig}}(v)$;
13 return $\mathsf{BFS}_{\mathsf{low}}(v)$, $\mathsf{BFS}_{\mathsf{hig}}(v)$;
```

Trimmed BFS. Algorithm 2 describes the v-sourced trimmed BFS. We initialize an empty queue Q and set the status of all vertices to unvisited (denoted as $\textcircled{?}$) (Line 1-2). Then, v is inserted into Q and $\mathsf{BFS}_{\mathsf{low}}(v)$, and the status of v is set to visited (denoted as $\textcircled{\rightarrow}$) (Line 3-4). Afterward, we pop a vertex u from Q and check each neighbor w of u (Line 6-7). If w is visited before, we do nothing (Line 8). Otherwise, depending on the order of w, we have two cases: 1) if w is lower in order than v, we continue expanding via w by setting the status of w as visited and inserting w both in Q and $\mathsf{BFS}_{\mathsf{low}}(v)$ (Line 9-10); 2) otherwise, we block the expansion via w and insert w into $\mathsf{BFS}_{\mathsf{hig}}(v)$ (Line 12). When the queue Q is empty, the BFS terminates, and $\mathsf{BFS}_{\mathsf{low}}(v)$ and $\mathsf{BFS}_{\mathsf{hig}}(v)$ are returned.

Lemma 2. *The time cost of Algorithm 2 is $O(|E| + |V|)$.*

Example 8. Fig. 3 shows the v_3-sourced trimmed BFS. First, v_3 is inserted into both Q and $\mathsf{BFS}_{\mathsf{low}}(v_3)$. Then, v_3 is popped from Q, and for v_3's out-neighbors $\{v_1, v_4, v_{10}\}$: v_4 and v_{10} are inserted into both $\mathsf{BFS}_{\mathsf{low}}(v_3)$ and Q because they are of lower order than v_3; the expansion via v_1 is pruned since $\mathsf{ord}(v_1) > \mathsf{ord}(v_3)$, and v_1 is inserted into $\mathsf{BFS}_{\mathsf{hig}}(v_3)$. Then, v_4 is popped from Q, and v_4's out-neighbors $\{v_6, v_{11}\}$ are examined. The BFS terminates when Q is empty, and we get $\mathsf{BFS}_{\mathsf{low}}(v_3) = \{v_3, v_4, v_{10}, v_6, v_{11}\}$, $\mathsf{BFS}_{\mathsf{hig}}(v_3) = \{v_1, v_2\}$.

Modified Framework. During the trimmed BFS sourced from v, we obtain $\mathsf{BFS}_{\mathsf{low}}(v)$ (vertices visited by BFS and of order lower than v) and $\mathsf{BFS}_{\mathsf{hig}}(v)$ (vertices with higher order that block the expansion). Using $\mathsf{BFS}_{\mathsf{low}}(v)$ and $\mathsf{BFS}_{\mathsf{hig}}(v)$, we optimize the original filtering-and-refinement framework.

Refinement. We first show that in the refinement phase, $\mathsf{BFS}_{\mathsf{hig}}(v)$ can replace $\mathsf{DES}_{\mathsf{hig}}(v)$ since vertices in $\mathsf{BFS}_{\mathsf{hig}}(v)$ reach all the descendants of vertices in $\mathsf{DES}_{\mathsf{hig}}(v)$.

Lemma 3. $\bigcup_{u \in \mathsf{BFS}_{\mathsf{hig}}(v)} \mathsf{DES}(u) = \bigcup_{u \in \mathsf{DES}_{\mathsf{hig}}(v)} \mathsf{DES}(u)$.

Proof. Let LHS be $\bigcup_{u \in \mathsf{BFS}_{\mathsf{hig}}(v)} \mathsf{DES}(u)$ and RHS be $\bigcup_{u \in \mathsf{DES}_{\mathsf{hig}}(v)} \mathsf{DES}(u)$.

- LHS $\subseteq$ RHS follows from the fact $\mathsf{BFS}_{\mathsf{hig}}(v) \subseteq \mathsf{DES}_{\mathsf{hig}}(v)$.
- RHS $\subseteq$ LHS: If $\exists s \in$ RHS $\setminus$ LHS, $s \in$ RHS implies there is a path from v to s containing some vertex $w \in \mathsf{DES}_{\mathsf{hig}}(v)$. On all paths from v to s, we collect the inner vertices in $\mathsf{DES}_{\mathsf{hig}}(v)$ and insert them in set S (S is not empty as $w \in \mathsf{DES}_{\mathsf{hig}}(v)$ is such a vertex). We choose the vertex $u \in S$ with the smallest distance to v: there is no other higher-order vertex on the v-u path. By Algorithm 2, $u \in \mathsf{BFS}_{\mathsf{hig}}(v)$. Thus, $s \in \bigcup_{u \in \mathsf{BFS}_{\mathsf{hig}}(v)} \mathsf{DES}(u) =$ LHS, contradiction. □

Example 9. Consider the graph G in Fig. 1. For v_3, $\mathsf{BFS}_{\mathsf{hig}}(v_3)$ can replace $\mathsf{DES}_{\mathsf{hig}}(v_3)$ for refinement since $\mathsf{BFS}_{\mathsf{hig}}(v_3) = \{v_1, v_2\}$ reaches all descendants of vertices in $\mathsf{DES}_{\mathsf{hig}}(v_3)$: $\bigcup_{u \in \mathsf{BFS}_{\mathsf{hig}}(v_3)} \mathsf{DES}(u) = \bigcup_{u \in \mathsf{DES}_{\mathsf{hig}}(v_3)} \mathsf{DES}(u) = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}, v_{11}\}$.

Filtering. Next, we show $\mathsf{BFS}_{\mathsf{low}}(v)$ is a super-set of $\mathsf{L}^-_{\mathsf{in}}(v)$, meaning that $\mathsf{BFS}_{\mathsf{low}}(v)$ can replace $\mathsf{DES}(v)$ for filtering.

Lemma 4. $\mathsf{L}^-_{\mathsf{in}}(v) \subseteq \mathsf{BFS}_{\mathsf{low}}(v)$.

Proof. Suppose there is $s \in \mathsf{L}^-_{\mathsf{in}}(v) \setminus \mathsf{BFS}_{\mathsf{low}}(v)$, then there must be a path from v to s through a high-order vertex $u \in \mathsf{BFS}_{\mathsf{hig}}(v)$. By Theorem 1, $s \notin \mathsf{L}^-_{\mathsf{in}}(v)$, contradiction. □

Example 10. Consider the graph G in Fig. 1. For v_3, $\mathsf{BFS}_{\mathsf{low}}(v_3)$ can replace $\mathsf{DES}(v_3)$ for filtering, because $\mathsf{BFS}_{\mathsf{low}}(v_3) = \{v_3, v_4, v_6, v_{10}, v_{11}\}$ includes all vertices in $\mathsf{L}^-_{\mathsf{in}}(v_3) = \emptyset$.

Basic Labeling Method. Lemma 4 shows that $\mathsf{BFS}_{\mathsf{low}}(v)$ is a super-set of $\mathsf{L}^-_{\mathsf{in}}(v)$, while Lemma 3 shows that $\mathsf{BFS}_{\mathsf{hig}}(v)$ is sufficient to eliminate invalid elements not in $\mathsf{L}^-_{\mathsf{in}}(v)$. Thus, we give the basic labeling method for labeling $v \in V$.

Step 1. In the filtering phase, we use a v-sourced BFS to find $\mathsf{BFS}_{\mathsf{low}}(v)$ and $\mathsf{BFS}_{\mathsf{hig}}(v)$;
Step 2. In the refinement phase, we perform a BFS for each vertex in $\mathsf{BFS}_{\mathsf{hig}}(v)$.
Step 3. Return $\mathsf{BFS}_{\mathsf{low}}(v) - \bigcup_{u \in \mathsf{BFS}_{\mathsf{hig}}(v)} \mathsf{DES}(u)$ as $\mathsf{L}^-_{\mathsf{in}}(v)$.

Combing Lemma 3, Lemma 4, and Theorem 2, the correctness of this method is given in Theorem 3.

Theorem 3. $\mathsf{L}^-_{\mathsf{in}}(v) = \mathsf{BFS}_{\mathsf{low}}(v) - \bigcup_{u \in \mathsf{BFS}_{\mathsf{hig}}(v)} \mathsf{DES}(u)$.

C-2. Improved Labeling Method

Compared with the framework given in Theorem 2, the basic method based on Theorem 3 reduces the number of BFSs needed in the refinement phase from $|\mathsf{DES}_{\mathsf{hig}}(v)|$ to $|\mathsf{BFS}_{\mathsf{hig}}(v)|$. But the number $|\mathsf{BFS}_{\mathsf{hig}}(v)|$ may still be very large. Therefore, can we avoid using a large number of BFSs in the refinement phase?

To answer this question, we revisit the refinement phase of the basic method (i.e., Lemma 3): when labeling v, w is eliminated when a vertex in $\mathsf{BFS}_{\mathsf{hig}}(v)$ can reach w. We focus on a specific vertex $u \in \mathsf{BFS}_{\mathsf{hig}}(v)$, which has the highest order on paths from v to w: when performing a u-sourced trimmed BFS in G, u can reach w, so w is in $\mathsf{BFS}_{\mathsf{low}}(u)$[1]; when performing a u-sourced trimmed BFS in the inverse graph $\overline{G}$, u can reach v, so v is in $\mathsf{BFS}^{\overline{\mathsf{G}}}_{\mathsf{low}}(u)$. Thus, by examining whether there exist vertices u of order higher than v such that $w \in \mathsf{BFS}_{\mathsf{low}}(u)$ and $v \in \mathsf{BFS}^{\overline{\mathsf{G}}}_{\mathsf{low}}(u)$, we can eliminate w to complete the refinement without using any BFSs: the existence of higher-order vertices u on the v-w paths implies that w can be eliminated.

Example 11. Consider the graph G in Fig. 1. For v_3 in G, v_4 can be eliminated because $\exists v_2 \in \mathsf{BFS}_{\mathsf{hig}}(v_3)$ s.t., 1) in G (Fig. 1), v_2-sourced BFS visits v_4 and hence $v_4 \in \mathsf{BFS}_{\mathsf{low}}(v_2)$; 2) in $\overline{G}$ (Fig. 2), v_2-sourced BFS visits v_3, and hence $v_3 \in \mathsf{BFS}^{\overline{\mathsf{G}}}_{\mathsf{low}}(v_2)$.

Improved Refinement. Based on the above idea, in the refinement phase of labeling v, to check whether some vertex $w \in \mathsf{BFS}_{\mathsf{low}}(v)$ should be removed, we need to check if there exists $u \in \mathsf{BFS}_{\mathsf{hig}}(v)$ such that $w \in \mathsf{BFS}_{\mathsf{low}}(u)$ and $v \in \mathsf{BFS}^{\overline{\mathsf{G}}}_{\mathsf{low}}(u)$. Determining $w \in \mathsf{BFS}_{\mathsf{low}}(u)$ can be done intuitively in G, since $\mathsf{BFS}_{\mathsf{low}}(u)$ is known; but determining $v \in \mathsf{BFS}^{\overline{\mathsf{G}}}_{\mathsf{low}}(u)$ is not so simple, since the information on $\overline{G}$ needs to be used.

To make it feasible to determine $v \in \mathsf{BFS}^{\overline{\mathsf{G}}}_{\mathsf{low}}(u)$, we create an inverted list $\mathsf{IBFS}_{\mathsf{low}}(v)$ for vertex $v \in V$.

Definition 6. If v is visited by the u-sourced trimmed BFS in $\overline{G}$, vertex u is in the **inverse list** $\mathsf{IBFS}_{\mathsf{low}}(v)$ of v, i.e., $\mathsf{IBFS}_{\mathsf{low}}(v) = \{u | v \in \mathsf{BFS}^{\overline{\mathsf{G}}}_{\mathsf{low}}(u)\}$.

With $\mathsf{IBFS}_{\mathsf{low}}(v)$, we can eliminate w by Lemma 5.

Lemma 5. *For a vertex $w \in \mathsf{BFS}_{\mathsf{low}}(v)$, $w \notin \mathsf{L}^{-}_{\mathsf{in}}(v)$ if $\exists u \in \mathsf{IBFS}_{\mathsf{low}}(v)$, and $w \in \mathsf{BFS}_{\mathsf{low}}(u)$.*

Proof. A vertex $u \in \mathsf{IBFS}_{\mathsf{low}}(v)$ with $w \in \mathsf{BFS}_{\mathsf{low}}(u)$ means there is a higher-order vertex u on the path from v to w. Then, $w \notin \mathsf{L}^{-}_{\mathsf{in}}(v)$ by Theorem 1. □

Improved Labeling Method. With the refinement given in Lemma 5, we give an improved labeling method for labeling v.

- Step 1. In the filtering phase, we use a v-sourced BFS in G to find $\mathsf{BFS}_{\mathsf{low}}(v)$ and $\mathsf{BFS}_{\mathsf{hig}}(v)$, for $\forall v \in V$;
- Step 2. We use a v-sourced BFS in $\overline{G}$ to find $\mathsf{BFS}^{\overline{\mathsf{G}}}_{\mathsf{low}}(v)$, and then get $\mathsf{IBFS}_{\mathsf{low}}(v)$ by Definition 6, for $\forall v \in V$;
- Step 3. In the refinement phase, if $\exists u \in \mathsf{IBFS}_{\mathsf{low}}(v)$, and $w \in \mathsf{BFS}_{\mathsf{low}}(u)$, the vertex w can be eliminated;
- Step 4. Return the non-eliminated vertices as $\mathsf{L}^{-}_{\mathsf{in}}(v)$.

Combing Lemma 5 and Theorem 3, the correctness of this method is given below.

Theorem 4. *$\mathsf{L}^{-}_{\mathsf{in}}(v) = \mathsf{BFS}_{\mathsf{low}}(v) - S$, where $S = \{w | w \in \mathsf{BFS}_{\mathsf{low}}(v), \exists u \in \mathsf{IBFS}_{\mathsf{low}}(v), w \in \mathsf{BFS}_{\mathsf{low}}(u)\}$.*

[1]Without ambiguity, $\mathsf{BFS}_{\mathsf{low}}(u)$ and $\mathsf{BFS}^{\mathsf{G}}_{\mathsf{low}}(u)$ refers to the same thing.

Note that in Step 2 of the improved labeling method, we need a v-sourced trimmed BFS on $\overline{G}$ to obtain $\mathsf{L}^{-}_{\mathsf{in}}(v)$. This step does not introduce additional costs because $\mathsf{BFS}^{\overline{\mathsf{G}}}_{\mathsf{low}}(v)$ is needed to obtain $\mathsf{L}^{-}_{\mathsf{out}}(v)$. In other words, only trimmed BFSs are required to obtain both $\mathsf{L}^{-}_{\mathsf{in}}(v)$ and $\mathsf{L}^{-}_{\mathsf{out}}(v)$.

So far, in addition to finding the backward label sets using the framework given in Theorem 2, we have proposed a basic method based on Theorem 3 and an improved method based on Theorem 4. We give in Table IV the number of BFSs required in the filtering and refinement phases for each method.

TABLE IV: The Comparison Between Labeling Methods

Method	Filtering	Refinement
Theorem 2	1	$\lvert\mathsf{DES}_{\mathsf{hig}}(v)\rvert$
Theorem 3 (Basic)	1	$\lvert\mathsf{BFS}_{\mathsf{hig}}(v)\rvert \leq \lvert\mathsf{DES}_{\mathsf{hig}}(v)\rvert$
Theorem 4 (Improved)	1	1

D. Distributed Implementation

To handle distributed graphs, we implement the improved labeling method using a vertex-centric system, which is denoted as DRL. We omit the distributed implementation of the basic labeling method, as this can be implemented in a similar way.

Algorithm 3: Compute() for DRL

```
Data: in-msgs ← messages from in-neighbors;
      out-msgs ← messages to out-neighbors;
if super-step = 1 then
    // w is vertex to perform computations
    $w = vertex\_id()$;
    $w$.status($z$) ← $\textcircled{?}$, for each vertex $z \in V$;
    $w$.status($w$) ← $\textcircled{\rightarrow}$;
    // message format:{ID, order}
    message ← $\{w, \mathsf{ord}(w)\}$;
    send message to out-neighbors;
foreach message ∈ in-msgs do
    // v is the source to do trimmed BFS
    $v$ ← message.ID;
    ord($v$) ← messge.order;
    if $w$.status($v$) = $\textcircled{\rightarrow}$ then continue;
    if ord($v$) > ord($w$) then
        if Check($v, w$)=true then continue;
        $w$.status($v$) ← $\textcircled{\rightarrow}$;
        message ← $\{v, \mathsf{ord}(v)\}$;
        send message to out-neighbors;
        // works on $\overline{G}$
        insert $v$ into $\mathsf{IBFS}_{\mathsf{low}}(w)$;
// only run after the final super-step
foreach v, s.t., $w$.status($v$) = $\textcircled{\rightarrow}$ do
    if Check($v, w$)=true then $w$.status($v$) ← $\textcircled{?}$;

Procedure Check(v, w)
    foreach $u \in \mathsf{IBFS}_{\mathsf{low}}(v)$ do
        if $w$.status($u$) = $\textcircled{\rightarrow}$ then return true
    return false
```

Algorithm. Algorithm 3 describes DRL, where the compute() function is executed on each vertex $w \in V$ in super-steps. We record the visited status of w using a status array[2] w.status. Specifically, if the value of w.status(v) is $\textcircled{?}$, then w is not visited by the vertex v; if the value is $\textcircled{\rightarrow}$, then w is visited by v. By reading the values of status arrays, the backward in-label sets of all vertices can be obtained.

[2]In the implementation, the hash table can be used to replace the array because of the sparsity of the array.

In the first super-step (Line 3), vertex w initializes its status array by assigning the unvisited status $\textcircled{?}$ to all vertices (Line 5), except for w itself, which is assigned as $\textcircled{\rightarrow}$ (Line 6). Then, w sends the message containing its vertex ID (w) and vertex order ($\mathsf{ord}(w)$) to out-neighbors (Line 7-8). In subsequent super-steps, once vertex w receives the message from in-neighbors (Line 9), w extracts vertex ID v (Line 10) and order $\mathsf{ord}(v)$ (Line 11) of the message. If $w.\mathsf{status}(v)$ is $\textcircled{\rightarrow}$, we do nothing as v visited w before (Line 12).

If $w.\mathsf{status}(v)$ is $\textcircled{?}$ and the order v is higher than w, we continue the v-sourced trimmed BFS via w (Line 13). We mark the status of $w.\mathsf{status}(v)$ as $\textcircled{\rightarrow}$ (Line 15), and we send the message $\{v, \mathsf{ord}(v)\}$ to w's out-neighbors to continue the v-sourced BFS. Also, on the inverse graph $\overline{G}$, if v can reach w, then v is inserted in $\mathsf{IBFS}_{\mathsf{low}}(w)$ (Line 18). Note that we will call the procedure $\mathsf{Check}(v, w)$ (Line 21-24) for an expansion pruning (Line 14): if $\mathsf{IBFS}_{\mathsf{low}}(v)$ contains a vertex u that can reach w, it follows from Lemma 5 that w is not in $\mathsf{L}^{-}_{\mathsf{in}}(v)$, and we prune the expansion of v-sourced BFS via w.

In the final super-step, we check for w the vertices v for which $w.\mathsf{status}(v)$ is $\textcircled{\rightarrow}$: if the procedure $\mathsf{Check}(v, w)$ returns true, we reset $w.\mathsf{status}(v)$ to $\textcircled{?}$ (Line 19-20). After this check, the vertices w for which $w.\mathsf{status}(v)$ is $\textcircled{\rightarrow}$ form the backward in-label set $\mathsf{L}^{-}_{\mathsf{in}}(v)$ of v. Finally, we can collect the backward label sets of each vertex on one machine to obtain an index the same as TOL to support reachability queries.

Analysis. We give the correctness analysis of DRL, i.e., the vertices w whose $w.\mathsf{status}(v)$ value is $\textcircled{\rightarrow}$ form $\mathsf{L}^{-}_{\mathsf{in}}(v)$ of v.

Theorem 5. *Given a graph G and a vertex v, $\mathsf{L}^{-}_{\mathsf{in}}(v) = \{w | w.\mathsf{status}(v) = \textcircled{\rightarrow}\}$ for Algorithm 3.*

Proof. We denote RHS by $\{w | w.\mathsf{status}(v) = \textcircled{\rightarrow}\}$ and we prove $\mathsf{L}^{-}_{\mathsf{in}}(v) = \mathsf{RHS}$.

- $\mathsf{RHS} \subseteq \mathsf{L}^{-}_{\mathsf{in}}(v)$: $w.\mathsf{status}(v) = \textcircled{\rightarrow}$ means that v can reach w and $\mathsf{ord}(w) < \mathsf{ord}(v)$. Then we verify that there are no higher-order vertices on any path from v to w, thus deriving $w \in \mathsf{L}^{-}_{\mathsf{in}}(v)$ by Theorem 1. Suppose there are high-order vertices on paths from v to w, we choose the highest-order vertex u. Since u's order is the highest on all v-w paths, u can reach v in $\overline{G}$, thus $u \in \mathsf{IBFS}_{\mathsf{low}}(v)$; u can reach w in G, thus $w.\mathsf{status}(u) = \textcircled{\rightarrow}$. So the procedure $\mathsf{Check(v, w)}$ will set $w.\mathsf{status}(v) = \textcircled{?}$, contradiction.
- $\mathsf{L}^{-}_{\mathsf{in}}(v) \subseteq \mathsf{RHS}$: Suppose there exists a vertex $w \in \mathsf{L}^{-}_{\mathsf{in}}(v)$ and $w.\mathsf{status}(v) = \textcircled{?}$, then there are two cases: 1) if v cannot reach w, then $w \notin \mathsf{L}^{-}_{\mathsf{in}}(v)$ by Theorem 1, contradiction; 2) there exists a higher-order vertex u in $\mathsf{IBFS}_{\mathsf{low}}(v)$ to block the expansion branch from v to w to set $w.\mathsf{status}(v)$ to $\textcircled{?}$ (Line 14) or to reset $w.\mathsf{status}(v)$ to $\textcircled{?}$ (Line 19-20), which contradicts with Theorem 1. □

We then analyze the computation and communication costs.

Lemma 6. *The computation cost of labeling a vertex $v \in V$ using Algorithm 3 is $O(|E| + |\mathsf{IBFS}_{\mathsf{low}}(v)| \cdot |V|)$, where $|\mathsf{IBFS}_{\mathsf{low}}(v)|$ is the inverted list size of v.*

Proof. The time required to perform a v-sourced trimmed BFS is $O(|E|)$. Also, Algorithm 3 triggers at most $|V|$ times of the $\mathsf{Check}()$ procedure for v, each requiring $|\mathsf{IBFS}_{\mathsf{low}}(v)|$ time. □

Lemma 7. *The communication cost of labeling a vertex $v \in V$ using Algorithm 3 is $O(|E| + |\mathsf{IBFS}_{\mathsf{low}}(v)|)$.*

Proof. Each vertex needs to send/read a message to its neighbors at most once for labeling a certain vertex v. In addition, we need to share $\mathsf{IBFS}_{\mathsf{low}}(v)$ to implement the $\mathsf{Check}()$ procedure. Hence, the communication cost is $\sum_{v \in V} d_{out}(v) + \sum_{v \in V} d_{in}(v) + |\mathsf{IBFS}_{\mathsf{low}}(v)| = |E| + |\mathsf{IBFS}_{\mathsf{low}}(v)|$. □

Remark. Although each vertex v needs to share its inverted list $\mathsf{IBFS}_{\mathsf{low}}(v)$, the size of $\mathsf{IBFS}_{\mathsf{low}}(v)$ is pretty small (empirical studies show that the average size of $\mathsf{IBFS}_{\mathsf{low}}(v)$ of each vertex v is less than one), so the communication overhead associated with sharing the inverted list is not significant. The efficiency of DRL is validated in Section VI.

IV. Batch Labeling Optimization

DRL creates backward label sets for all vertices in parallel. However, DRL misses the opportunity provided by the serial execution of TOL — the already processed high-order vertices strongly prune the search space when labeling the current vertex. As a remedy, we further improve the labeling efficiency by splitting vertices into batches to trade-off between pruning power and parallelization.

Batch Sequence. We split the vertices into a batch sequence for *batch labeling*: we label all the vertices within a batch simultaneously, while vertices in different batches perform the labeling process sequentially.

Definition 7. $[V_1, V_2, \ldots, V_g]$ is a **batch sequence** when

- $\bigcup_{i \in g} V_i = V$ and $V_i \cap V_j = \emptyset$, for $\forall i \neq j$;
- for vertex $u \in V_i$ and vertex $v \in V_j$ with $i < j$, it must be ensured that $\mathsf{ord}(u) > \mathsf{ord}(v)$.

The batch sequence $[V_1, V_2, \ldots, V_g]$ is a graph partition since it disjointly covers all vertices. Also, the vertices with high order are placed before the vertices with low order in the sequence. When the batch size $|V_i|$ $(1 \leq i \leq g)$ is fixed to one, we get $|V|$ batches of vertices for labeling. This fully serial execution is howTOL works. When the batch size is fixed to $|V|$, we get 1 batch of vertices for labeling. This fully parallel execution is how DRL works. By setting the batch size flexibly, we make a trade-off between TOL and DRL.

To obtain a valid batch sequence, we need two parameters: an initial batch size variable b (ranging from 1 to $|V|$) and an increment factor k. The specific procedure is given below.

Step 1. Sort vertices V in a non-increasing order of ord values, and then copy sorted vertices into the set S;
Step 2. In iteration i, remove b vertices with the highest order from S to form V_i (i.e., $S \leftarrow S \setminus V_i$), and then multiply b by k for the next iteration (i.e., $\mathsf{b} \leftarrow \mathsf{b} \cdot \mathsf{k}$);
Step 3. Stop at round $g + 1$ when $S = \emptyset$ and return $[V_1, V_2, \cdots, V_g]$; otherwise, increase i by 1 and go to Step 2.

The number of vertices in the last batch V_g may not exceed b. We set the values of both b and k to 2. The effect of b and k on the labeling efficiency is discussed in Section 5.

Example 12. Consider the graph G in Fig. 1. Suppose $\mathsf{b} = 2$ and $\mathsf{k} = 2$. In the first round ($\mathsf{b} = 2$), we get $V_1 = \{v_1, v_2\}$; in the second round ($\mathsf{b} = 4$), we get $V_2 = \{v_3, v_4, v_5, v_6\}$; in the third round ($\mathsf{b} = 8$), we get $V_3 = \{v_7, v_8, v_9, v_{10}, v_{11}\}$. $[V_1, V_2, V_3]$ is a batch sequence of G.

Algorithm 4: Compute() for $\mathsf{DRL_b}$

```
1  Data: $in\text{-}msgs \leftarrow$ messages from in-neighbors;
2        $out\text{-}msgs \leftarrow$ messages to out-neighbors;
3  Input: $V_i$;
4  if $super\text{-}step$ = 1 then
       // $w$ is vertex to perform computations
5      $w = vertex\_id()$;
6      if $w \notin V_i$ or $\mathsf{L}^{V_i}_{\mathsf{out}}(w) \cap \mathsf{L}^{V_i}_{\mathsf{in}}(w) \neq \emptyset$ then Return;
7      the same as Line 5-8 of Algorithm 3;
8      broadcast $\mathsf{L}^{V_i}_{\mathsf{out}}(w)$ and $\mathsf{L}^{V_i}_{\mathsf{in}}(w)$ to all computation nodes;
9  for each message $\in$ in-msgs do
       // $v$ is source to do trimmed BFS
10     $v \leftarrow message.ID$;
11     $\mathsf{ord}(v) \leftarrow messge.order$;
12     if $\mathsf{L}^{V_i}_{\mathsf{out}}(w) \cap \mathsf{L}^{V_i}_{\mathsf{in}}(w) \neq \emptyset$ then Continue;
13     the same as Line 12-20 in Algorithm 3;
   // only run after the final super-step
14 $\mathsf{L}^{V_{i+1}}_{\mathsf{in}}(w) \leftarrow \{v | w.\mathsf{status}(v) = \textcircled{\rightarrow}\}$;
```

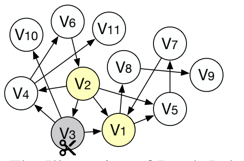


Fig. 4: The Illustration of Batch Labeling

Batch Label Sets. Since we process vertices in batches, the vertices in previous batches $[V_1, V_2, \cdots, V_{i-1}]$ completed labeling before the current batch V_i begins. We define label sets generated by vertices in batches $[V_1, V_2, \cdots, V_{i-1}]$ as batch label sets regarding V_i.

Definition 8. Given the batch V_i, the **batch in-label set** $\mathsf{L}^{V_i}_{\mathsf{in}}(w)$ of a vertex $w \in V$ is defined as $\mathsf{L}^{V_i}_{\mathsf{in}}(w) = \{u | u \in \mathsf{L}_{\mathsf{in}}(w), \mathsf{ord}(u) > \mathsf{ord}(V_i)\}$; the **batch out-label set** $\mathsf{L}^{V_i}_{\mathsf{out}}(w)$ of a vertex $w \in V$ is defined as $\mathsf{L}^{V_i}_{\mathsf{out}}(w) = \{u | u \in \mathsf{L}_{\mathsf{out}}(w), \mathsf{ord}(u) > \mathsf{ord}(V_i)\}$, where $\mathsf{ord}(V_i) = \max\{\mathsf{ord}(v) | v \in V_i\}$.

Similar to TOL, we can use the batch label sets to perform the pruning operation during the current batch, thereby optimizing the efficiency of DRL.

Example 13. Consider the graph G in Fig. 1. Suppose v_1 and v_2 finished labeling in the previous batch V_1 and only v_3 is in current batch V_2. Before V_2 starts labeling, the batch in-label set $\mathsf{L}^{V_2}_{\mathsf{in}}(v_9)$ of v_9 is $\{v_1\}$: v_1 in $\mathsf{L}_{\mathsf{in}}(v_9) = \{v_1, v_8, v_9\}$ has a higher order than v_3; the batch out-label set $\mathsf{L}^{V_2}_{\mathsf{out}}(v_9)$ of v_9 is $\emptyset$: $\mathsf{L}_{\mathsf{out}}(v_9) = \{v_9\}$ has no vertex of higher order than v_3.

Algorithm. We incorporate the idea of batch labeling into DRL and implement it on a vertex-centric system to obtain algorithm $\mathsf{DRL_b}$ (Algorithm 4). $\mathsf{DRL_b}$ resembles DRL (Algorithm 3), and we only list the differences. First, only vertices in the current batch V_i are selected for labeling (Line 6); Also, if there is a higher-order vertex on the path from w to w ($\mathsf{L}^{V_i}_{\mathsf{out}}(w) \cap \mathsf{L}^{V_i}_{\mathsf{in}}(w) \neq \emptyset$), w is pruned (note that the graph is unnecessary to be acyclic) (Line 6). The batch label sets are then sent to all computation nodes (Line 8). The batch label sets are also used for pruning in Line 12. Then, at the end of batch V_i, vertices v with $w.\mathsf{status}(v) = \textcircled{\rightarrow}$ form batch in-label set of w for the next round (Line 14).

Example 14. Fig. 4 shows how batch labeling works. When labeling v_3 in the current batch V_i, as $\mathsf{L}^{V_i}_{\mathsf{in}}(v_3) = \{v_2\}$ intersects with $\mathsf{L}^{V_i}_{\mathsf{out}}(v_3) = \{v_1, v_2\}$, v_3 is pruned immediately — the search space for labeling v_3 is dramatically reduced.

Analysis. We analyze the correctness of $\mathsf{DRL_b}$.

Theorem 6. *Given a graph G and a vertex v, $\mathsf{L}^{-}_{\mathsf{in}}(v) = \{w | w.\mathsf{status}(v) = \textcircled{\rightarrow}\}$ for Algorithm 4.*

Proof. We denote $\{w | w.\mathsf{status}(v) = \textcircled{\rightarrow}\}$ by RHS and prove that $\mathsf{L}^{-}_{\mathsf{in}}(v) = \mathsf{RHS}$.

- $\mathsf{RHS} \subseteq \mathsf{L}^{-}_{\mathsf{in}}(v)$: Suppose there is a vertex w with $w.\mathsf{status}(v) = \textcircled{\rightarrow}$ but $w \notin \mathsf{L}^{-}_{\mathsf{in}}(v)$, $w \notin \mathsf{L}^{-}_{\mathsf{in}}(v)$ implies 1) $v \not\to w$, which shows that $w.\mathsf{status}(v) = \textcircled{?}$, or 2) there is a vertex with order higher than v on a path from v to w, so we select the highest-order vertex s. If s is in the previous batches and the current batch is denoted as V_i, since there are no vertices to prune s, $s \in \mathsf{L}^{V_i}_{\mathsf{out}}(v)$ and $s \in \mathsf{L}^{V_i}_{\mathsf{in}}(w)$. Hence, $w.\mathsf{status}(v) = \textcircled{?}$ by Line 12 of Algorithm 4; or s is in the current batch, then $w.\mathsf{status}(v) = \textcircled{?}$ by the correctness of DRL, contradiction.
- $\mathsf{L}^{-}_{\mathsf{in}}(v) \subseteq \mathsf{RHS}$: We prove this by induction on the batch number. When $v \in V_1$, since no pruning occurs, then $\mathsf{DRL_b}$ is correct by the correction of DRL. Suppose V_{i-1} finishes labeling and $\mathsf{DRL_b}$ is correct, we prove $\mathsf{DRL_b}$ is correct for $v \in V_i$. Since $w \in \mathsf{L}^{-}_{\mathsf{in}}(v)$, then v can reach w and no pruning occurs at Line 12 of Algorithm 4. Therefore, by the correction of DRL, $\mathsf{DRL_b}$ is correct for $v \in V_i$. □

We then provide its computation and communication costs.

Lemma 8. *The computation cost of labeling a vertex $v \in V$ using Algorithm 4 is $O(|E'| + (|\mathsf{IBFS_{low}}(v)| + \Delta) \cdot |V|)$, where $E' \subseteq E$, and Δ is the largest label size.*

Proof. For each vertex v, Algorithm 4 needs to explore the reduced search space (denoted as E', $E' \subseteq E$) due to the pruning operation. Moreover, Algorithm 4 requires at most $|V|$ times of Check() procedure (each costing $O(|\mathsf{IBFS_{low}}(v)|)$) and $|V|$ label queries (each costing $O(\Delta)$). □

Lemma 9. *The communication cost of labeling a vertex $v \in V$ using Algorithm 4 is $O(|E'| + |\mathsf{IBFS_{low}}(v)| + \Delta)$.*

Proof. The cost comes from: 1) sharing label sets with other computation nodes, which incurs $O(\Delta)$ cost; 2) sending $\mathsf{IBFS_{low}}(v)$ for refinement; 3) reduced search space E'. □

Remark. Compared to DRL (Algorithm 3), $\mathsf{DRL_b}$ (Algorithm 4) requires additional costs to share and query batch label sets. However, empirical studies in Section VI show that the benefit of reducing the search space from E to $E' \subseteq E$ outweighs the additional overhead.

V. Related Work

Index-free Approaches. The online search [16] such as breadth-first [21] or depth-first search [22] can be used to process reachability queries. However, because the graph needs to be used at the query time, the query latency can be large.

Index-assisted Approaches. Index-assisted methods speed up the online search by using auxiliary structures. The auxiliary structures can be subgraphs [6], multiple intervals [7], independent permutations [8], bloom filters [9], or partial label sets [15]. One of the best-known approaches in this category is BFL [9]. The basic idea of BFL is that if vertex s can reach vertex t, then s can reach all descendants $\mathsf{DES}(t)$ of t, i.e., $\mathsf{DES}(t) \subseteq \mathsf{DES}(s)$. Through a Bloom filter, BFL maps the descendants $\mathsf{DES}(v)$ of each vertex v to a subset as the out-label set of v [23]. At query time, the labels alone can be used to determine that s cannot reach t: if t's out-label set is not fully contained in s's out-label set, then $\mathsf{DES}(t) \not\subseteq \mathsf{DES}(s)$ and thus $s \not\rightsquigarrow t$. However, if s can reach t, then BFL needs to perform a graph search to report the answer. Similarly, BFL uses the Bloom filter to map each vertex's ancestors to generate its in-label set. For BFL, as the index cannot answer all queries, the graph needs to be loaded into memory at query time (whereas the index-only approach does not need to rely on the graph at query time). BFL is undesirable for distributed graphs because (1) its index construction strictly follows the post-order of depth-first search (DFS), and thus requires performing distributed DFS, and (2) it needs to traverse distributed graphs during query processing. These two operations incur high costs in the distributed environment, as will be validated in Section VI.

Index-only Approaches. Reachability relations between all pairs of vertices can be stored as an index using a transit closure (TC) in $O(n^2)$ space. Due to its huge size, TC is compressed by intervals [24], bit vectors [25], or other structures (e.g., a spanning tree and additional intervals [26], multiple chains [27], and a path-tree [28]). The limitation of TC is the huge space required. Another type of index is created by reachability labeling methods. Reachability labeling methods were pioneered by Cohen et al. [10]. After that, [11] provided a divide-and-conquer strategy for labeling; geometric-based and graph partition-based labeling methods were proposed in [12] and [13], respectively. The state-of-the-art index-only approach TOL is introduced in Section II.

Parallelized Distance Labeling. Parallelized distance labeling methods are proposed for shortest distance queries. Li et al. [29] designed a parallel distance labeling algorithm for unweighted graphs; Lakhotia et al. [30] provided a distributed distance labeling algorithm for weighted graphs. For distance labeling, as shown in [29], v is inserted into the label set of w if and only v is the highest-order vertex on *all shortest paths* from v to w. But according to Theorem 1 in Section III, v is inserted into the label set of w if and only if v is the highest-order vertex on *all paths* from v to w. Current distance labeling methods cannot find a higher-order vertex *not* on the shortest path, thus resulting in much larger indexes than the reachability labeling methods. Therefore, there is still a need to investigate new techniques for parallelizing reachability labeling, which is the focus of this paper.

VI. Experiments

A. Settings

Algorithms. We aim to propose distributed labeling algorithms that produce the same index as TOL. Our methods include:

- DRL (Algorithm 3), a distributed labeling algorithm based on Theorem 4.
- DRL^-, a basic labeling algorithm based on Theorem 3. Since its distributed implementation is similar to DRL, we omit its implementation details.
- $\mathsf{DRL_b}$ (Algorithm 4), a distributed algorithm obtained by applying batch labeling to DRL.

Datasets. The experiments were conducted on 18 real-world directed graphs that are widely used in recent work related to reachability queries [3], [6], [31]. The properties of the graphs are shown in Table V. The largest graph has more than 3.7 billion edges. All the datasets are from Stanford Large Network Dataset Collection[3] [32], Koblenz Network Collection[4] [33], Laboratory for Web Algorithms[5] [34], [35], Network Data Repository[6] [36], and the links in [37]. Note that, to verify the generality of our algorithms for processing distributed graphs, we do not transform the graphs into acyclic graphs, but build the indexes directly on the original graphs.

TABLE V: Datasets

Name	Dataset	$\lvert V\rvert$	$\lvert E\rvert$	Type
WEBW	Web-wikipedia	1,864,433	4,507,315	Web
DBPE	Dbpedia	3,365,623	7,989,191	Knowledge
CITE	Citeseerx	6,540,401	15,011,260	Citation
CITP	Cit-patent	3,774,768	16,518,947	Citation
TW	Twitter	18,121,168	18,359,487	Social
GO	Go-uniprot	6967956	34,770,235	Biology
SINA	Soc-sinaweibo	58,655,849	261,321,071	Social
LINK	Wikipedia-link	13,593,032	437,217,424	Web
WEBB	Webbase-2001	118,142,155	1,019,903,190	Web
GRPH	Graph500	17,043,780	1,046,934,896	Synthetic
TWIT	Twitter-2010	41,652,230	1,468,365,182	Social
HOST	Host-linkage	57,383,985	1,643,624,227	Web
GSH	Gsh-2015-host	68,660,142	1,802,747,600	Web
SK	Sk-2005	50,636,154	1,949,412,601	Web
TWIM	Twitter-mpi	52,579,682	1,963,263,821	Social
FRIE	Friendster	68,349,466	2,586,147,869	Social
UK	Uk-2006-05	77,741,046	2,965,197,340	Web
WEBS	Webspam-uk	105,896,555	3,738,733,648	Web

Environment. We implement all algorithms in C++ and compile them using GNU GCC 4.8.5. Our distributed algorithms ($\mathsf{DRL}^-, \mathsf{DRL}, \mathsf{DRL_b}$) are designed to run on a vertex-centric system. To eliminate the dependence on specific features provided by the vertex-centric system, we implement this system ourselves using MPI. We map graph vertices to different computation nodes via vertex IDs to make it suitable for distributed setups.

Our algorithms are executed on a cluster of 32 computation nodes — each node contains an Intel Xeon 2.7 GHz CPU, 32 GB main memory, and runs Linux (Red Hat Linux 4.8.5, 64 bits). In contrast, centralized algorithms such as TOL [14] and BFL [9] are executed on only one node with the same system settings. If not explicitly stated, we only run one thread on each computation node. We set the cut-off time to 2 hours. If the algorithm runs out of memory or cannot complete the computation within the cut-off time, the execution time is marked as "INF".

B. Comparison with Competitor Methods

Exp 1: Comparison with TOL. TOL is an index-only algorithm [14]. To illustrate the necessity of the proposed methods, we compare our best method $\mathsf{DRL_b}$ with TOL. The results of the comparison are given in Table VI. When a method cannot complete index creation because it exceeds the memory limit, we mark its results with the notation "-" in the table.

On index time. The time of $\mathsf{DRL_b}$ includes both computation time and communication time. On a medium-sized graph that can be accommodated by a single computation node, $\mathsf{DRL_b}$'s index time can be at most 9.37 times faster than TOL. Note that TOL is a centralized algorithm and cannot handle distributed graphs. When a single computation node cannot accommodate a graph (e.g., WEBS), TOL fails to work. In contrast, $\mathsf{DRL_b}$ can index all graphs within half an hour. This shows that our method can efficiently handle large-scale graphs that are beyond the ability of TOL.

[3]http://snap.stanford.edu/data/
[4]http://konect.uni-koblenz.de/
[5]http://law.di.unimi.it
[6]http://networkrepository.com/

TABLE VI: The Comparison with Competitor Methods

	Index Time (sec)					Index Size (MB)					Query Time (sec)				
Name	BFL^{C}	BFL^{D}	TOL	DRL$_{b}$	DRL$_{b}^{M}$	BFL^{C}	BFL^{D}	TOL	DRL$_{b}$	DRL$_{b}^{M}$	BFL^{C}	BFL^{D}	TOL	DRL$_{b}$	DRL$_{b}^{M}$
WEBW	1.51	59.21	61.84	9.08	7.31	85.35	85.35	432.06	432.06	432.06	8.58E-07	5.39E-05	2.09E-07	2.09E-07	2.09E-07
DBPE	2.10	110.17	2.21	0.92	0.64	154.07	154.07	63.91	63.91	63.91	2.25E-07	4.63E-05	1.51E-07	1.51E-07	1.51E-07
CITE	3.39	195.62	4.95	2.34	1.42	299.40	299.40	138.80	138.80	138.80	1.24E-07	4.52E-05	1.78E-07	1.78E-07	1.78E-07
CITP	5.10	138.30	125.21	13.36	11.17	172.80	172.80	622.04	622.04	622.04	5.68E-07	5.07E-05	3.12E-07	3.12E-07	3.12E-07
TW	3.74	469.34	7.27	1.13	1.40	829.52	829.52	271.60	271.60	271.60	1.95E-07	6.72E-05	1.84E-07	1.84E-07	1.84E-07
GO	3.56	365.38	7.40	1.76	2.03	318.97	318.97	274.43	274.43	274.43	1.11E-07	4.41E-05	2.02E-07	2.02E-07	2.02E-07
SINA	41.35	2,822.48	–	136.32	–	2,685.05	2,685.05	–	13,691.20	–	2.82E-06	8.64E-05	–	6.76E-07	–
LINK	16.29	213.43	55.16	15.64	9.38	622.24	622.24	239.76	239.76	239.76	2.29E-07	7.31E-05	1.35E-07	1.35E-07	1.35E-07
WEBB	–	1,181.08	–	103.98	–	–	5,408.12	–	2,578.85	–	–	2.37E-04	–	1.84E-07	–
GRPH	46.30	6.36	76.31	24.00	16.44	780.20	780.20	325.01	325.01	325.01	9.61E-08	7.03E-05	8.71E-08	8.71E-08	8.71E-08
TWIT	57.44	304.55	134.87	62.82	35.79	1,906.69	1,906.69	766.17	766.17	766.17	1.55E-07	1.43E-04	1.06E-07	1.06E-07	1.06E-07
HOST	–	1,655.19	–	66.87	–	–	2,626.83	–	926.77	–	–	5.30E-04	–	2.26E-07	–
GSH	–	512.04	–	77.93	–	–	3,143.01	–	1,266.78	–	–	2.85E-04	–	1.37E-07	–
SK	–	219.47	–	82.85	–	–	2,317.94	–	975.83	–	–	1.80E-04	–	1.01E-07	–
TWIM	–	359.05	–	68.36	–	–	2,406.91	–	958.17	–	–	2.60E-04	–	2.76E-07	–
FRIE	–	688.47	–	112.77	–	–	3,128.79	–	1,240.01	–	–	3.58E-04	–	4.38E-07	–
UK	–	296.52	–	217.94	–	–	3,558.70	–	1,567.59	–	–	2.25E-04	–	2.81E-07	–
WEBS	–	747.92	–	188.58	–	–	4,847.56	–	2,063.97	–	–	4.81E-04	–	4.15E-07	–

On index size. For DRL$_{b}$, we aggregate the index distributed on different computation nodes on one node. Hence, our algorithms have the same index size as TOL. The index size of DRL$_{b}$ (and TOL) on all graphs is very small. For example, the largest graph WEBS has an index size of 2.06 GB. This indicates that although distributed graphs may be large, the generated index can be accommodated on an ordinary machine to support in-memory queries. So, it is feasible to create the index of TOL for a distributed graph because the index size is not so large.

On query time. DRL$_{b}$ creates the same index as TOL, so the query time is the same for TOL and DRL$_{b}$. The query time of DRL$_{b}$ (and TOL) on all graphs is less than one microsecond. This result reinforces our research motivation that efficient reachability queries on a distributed graph can be supported by proposing new labeling methods to obtain the same index as TOL.

Exp 2: Comparison with BFL**.** We compare our best method DRL$_{b}$ with BFL, which is an index-assisted method [9]. BFL uses DFS to create the index and may also use the online search (e.g. DFS) at query time. We use the code provided in [9] to implement the *centralized* BFL algorithm, which is denoted as BFL^{C}, where all parameters are set by default. Also, we implemented a distributed version of DFS (which is a core operation of BFL) to obtain the *distributed* BFL algorithm, which is denoted as BFL^{D}. BFL^{C} runs on one computation node, while BFL^{D} runs on 32 nodes. We compare DRL$_{b}$, BFL^{C}, and BFL^{D} and record the results in Table VI.

On index time. (1) First, we compare DRL$_{b}$ with the centralized algorithm BFL^{C}. On medium-sized graphs, the index time of BFL^{C} is normally better than that of DRL$_{b}$. But the index time of DRL$_{b}$ is comparable to that of BFL^{C}: the index time of DRL$_{b}$ is within seven times that of BFL^{C}. Moreover, DRL$_{b}$ can handle large-scale graphs for which BFL^{C} cannot create indexes. (2) Then, we compare DRL$_{b}$ with the distributed algorithm BFL^{D}. BFL^{D} can process large-scale graphs by partitioning them to different computation nodes. But because BFL^{D} requires the distributed DFS to create indexes, its index time is very high: BFL^{D} runs on average 52.54 times slower than DRL$_{b}$.

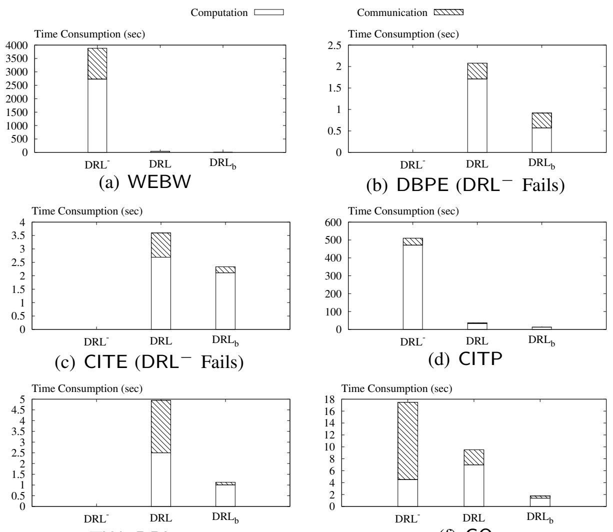
On index size. Since the index sizes of BFL^{C} and BFL^{D} are the same on all graphs, we only compare the index sizes of BFL^{D} and DRL$_{b}$. It can be seen that the index size of BFL^{D} is small on all graphs (no larger than 6 GB), and the index of BFL^{D} is smaller than that of DRL$_{b}$ on WEBW, CITP, and SINA. However, the index size of DRL$_{b}$ is smaller than that of BFL^{D} on the other graphs: the index size of DRL$_{b}$ is on average 2.38 times smaller than that of BFL^{D} on these graphs.

On query time. (1) We first compare the query time of BFL^{C} and DRL$_{b}$. Both BFL^{C} and DRL$_{b}$ can answer queries in microseconds, but the performance of DRL$_{b}$ is better than BFL^{C}: on average, the query time of DRL$_{b}$ is 1.8 times faster than that of BFL^{C} on graphs that BFL^{C} can process. (2) Then we compare the query time of BFL^{D} and DRL$_{b}$. Because BFL^{D} cannot avoid traversing the distributed graph to answer queries, the performance of BFL^{D} can be very bad: the query time of BFL^{D} is on average 867.6 times longer than that of DRL$_{b}$.

Overall, BFL performs well only when index creation can be done on a single node (see performance of BFL^{C}); when dealing with large graphs, BFL has high index time and query time due to the high cost of distributed DFS and graph search (see performance of BFL^{D}). This clarifies why we parallelize the index-only method TOL instead of BFL. Note that the graphs we use are not converted to be acyclic because of the high cost of performing such conversions in a distributed environment using DFS. This partly explains why there are some minor inconsistencies between our conclusions and [9].

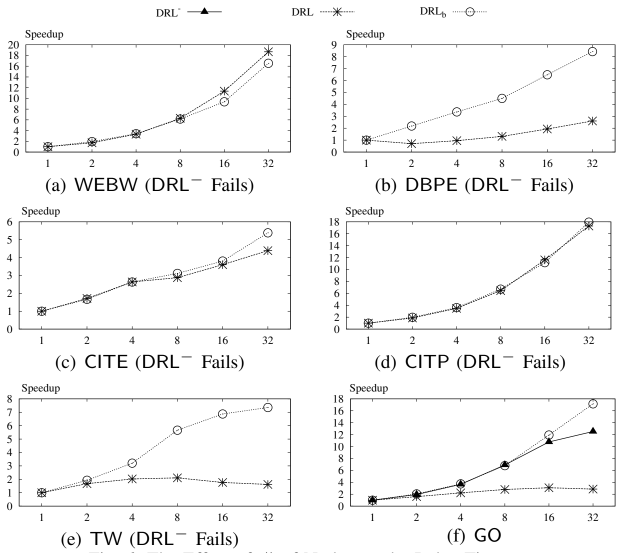
Exp 3: Comparison with Multi-core Version. Our algorithm DRL$_{b}$ (see Algorithm 4) achieves parallelism among multiple computation nodes in a distributed environment. Besides, we can also achieve parallelism among multiple threads (instead of multiple nodes), resulting in a multi-core version of DRL$_{b}$, which is denoted as DRL$_{b}^{M}$. For a fair comparison, we test DRL$_{b}^{M}$ in a machine configured similarly to the single computation node used by DRL$_{b}$, and this machine contains 32 cores and has a memory size of 32 GB. OpenMP [38] is used to implement DRL$_{b}^{M}$. Since DRL$_{b}$ and DRL$_{b}^{M}$ have the same index size and query time, we compare only the index time between them. We record the results in Table VI and have the following findings.

On medium-sized graphs. Because DRL$_{b}^{M}$ can use shared memory for data exchange [39], it avoids the communication cost of DRL$_{b}$. This leads to a better index time for DRL$_{b}^{M}$ than for DRL$_{b}$ in most cases: DRL$_{b}^{M}$ is 1.34 times faster than DRL$_{b}$ on graphs where DRL$_{b}^{M}$ can create indexes. However, this speedup is limited. One possible reason is that the communication cost of DRL$_{b}$ is relatively small compared to the computation cost (see the communication time and the computation time of DRL$_{b}$ in Exp



(a) WEBW (b) DBPE (DRL$^-$ Fails) (c) CITE (DRL$^-$ Fails) (d) CITP (e) TW (DRL$^-$ Fails) (f) GO

Fig. 5: The Comparison of Communication and Computation Time



(a) WEBW (DRL$^-$ Fails) (b) DBPE (DRL$^-$ Fails) (c) CITE (DRL$^-$ Fails) (d) CITP (DRL$^-$ Fails) (e) TW (DRL$^-$ Fails) (f) GO

Fig. 6: The Effect of # of Nodes on the Index Time

4 for details), leading to a less prominent advantage of shared memory.

On large-scale graphs. Since $\mathsf{DRL}_\mathsf{b}^\mathsf{M}$ is a centralized algorithm, the usability of $\mathsf{DRL}_\mathsf{b}^\mathsf{M}$ is limited by memory and therefore cannot build indexes for massive graphs. For example, $\mathsf{DRL}_\mathsf{b}^\mathsf{M}$ ran out of memory when building the index for WEBS. On the other hand, DRL_b can allocate graphs to multiple computation nodes and thus is more suitable for large graph processing.

C. Comparison Between Proposed Algorithms

Exp 4: Communication and Computation Time. We compare our proposed labeling algorithms, DRL$^-$, DRL, and DRL_b. We divide the index time of the proposed algorithms into computation time and communication time. If an algorithm is unable to finish labeling within the cut-off time, we do not report its time and mark the failure at the title of that graph. Due to space constraints, we present in Fig. 5 the results on the first 6 graphs (WEBW, DBPE, CITE, CITP, TW, and GO) with the following findings.

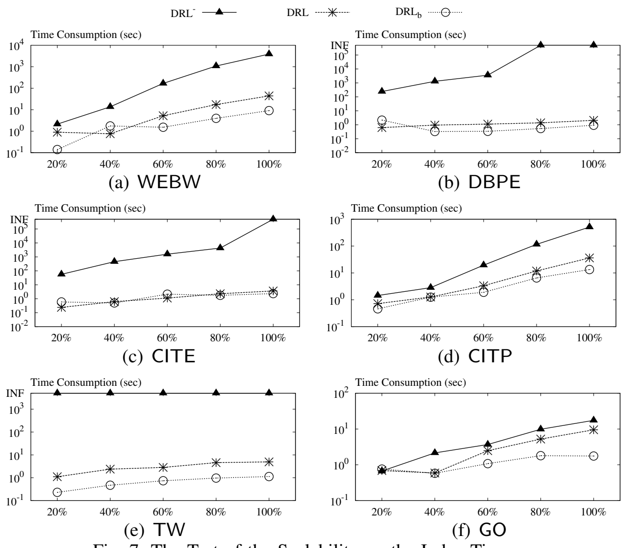
Comparison of DRL$^-$ *and* DRL. Compared to DRL$^-$, DRL uses the inverted list to implement the refinement phase. We find that DRL can index on DBPE, CITE and TW, while DRL$^-$ cannot. In addition, on the other three graphs, DRL has an average of 88.2 times shorter index time than DRL$^-$, thanks to the new refinement technique used by DRL.

Comparison of DRL *and* DRL_b. DRL_b uses batch labeling to further optimize DRL. We find that DRL_b has an average of 3.5 times shorter index time over DRL. Moreover, DRL_b reduces the computation time while substantially reducing the communication cost of DRL, which validates the effectiveness of the optimization strategy used by DRL_b.

Exp 5: Effect of Node Number. We used 32 computation nodes by default. To test the impact of the number of computation nodes on the proposed algorithms, we varied the number of nodes from 1, 2, 4, 8, 16 to 32 and recorded the corresponding index time. We define the **speedup** as the ratio of the index time on one node to the index time on x nodes, i.e., $speedup = \frac{\text{the index time on 1 node}}{\text{the index time on x node}}$. If an algorithm fails to finish labeling within the cut-off time on 1 node, we do not report its speedup ratio and mark the failure at the title of that graph. We show the speedup ratios on the first six graphs in Fig. 6 with the following findings.

DRL_b *has a satisfying speedup ratio.* The maximum speedup ratio for DRL_b with 32 nodes is 17.93 (on CITP) compared to using a single node. Moreover, the speedup ratio of DRL_b shows an increasing trend as the number of nodes increases.

The speedup of DRL$^-$ *and* DRL *has limitations.* Although the maximum speedup ratio of DRL$^-$ is 12.54 (on GO), DRL$^-$ cannot finish labeling on other five graphs using a single node within the cut-off time. On the other hand, although the maximum speedup ratio of DRL is 18.69 (on WEBW), on TW, the ratio of DRL is only 2.86 while that of DRL_b is 17.2, which shows that introducing the batch labeling optimization maintains a better speedup ratio.



(a) WEBW (b) DBPE (c) CITE (d) CITP (e) TW (f) GO

Fig. 7: The Test of the Scalability on the Index Time

Exp 6: Test of Scalability. In testing the scalability of the proposed algorithms, we divide the edges of the graph into five disjoint groups, each group consisting of $\frac{1}{5}$ edges of the

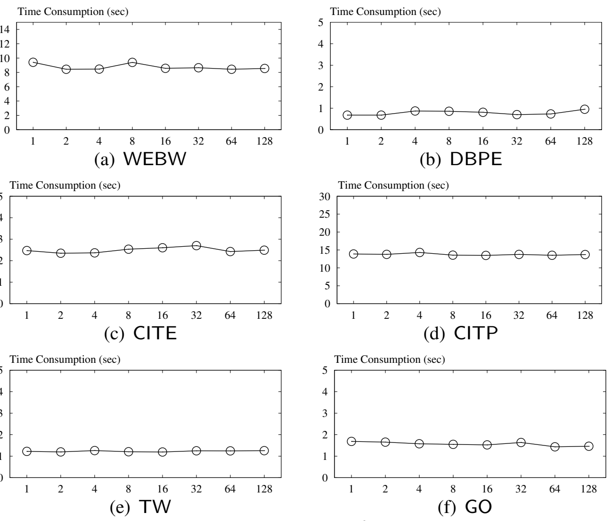


Fig. 8: The Effect of Initial Batch Size b on the Index Time

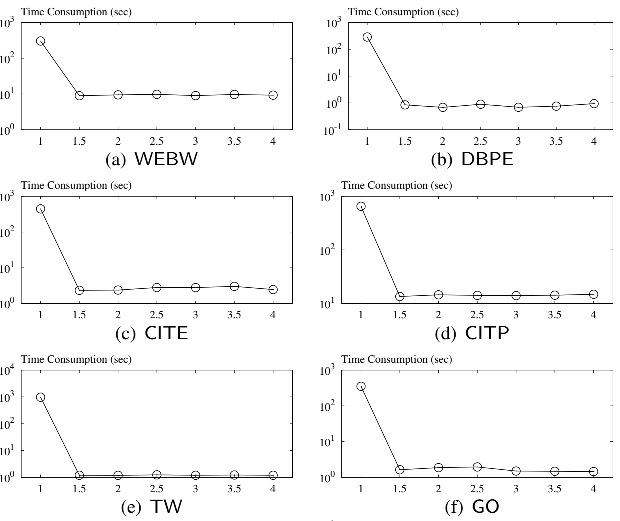


Fig. 9: The Effect of Factor k on the Index Time

original graph. We generate five test graphs, where the i-th test graph contains edges in the first i groups. The experiments are conducted on five test graphs for each dataset. Since the index size and query time are the same for all algorithms, we omit the discussion of them and only provide the effect on the index time in Fig. 7. We have the following finding.

The proposed algorithms exhibit good scalability. The index time of all algorithms improves as the graph size becomes larger. However, the increase of all methods is smooth. For example, the index time of $\mathsf{DRL_b}$ for the test graph with 100% edges is 4.8 times longer than that of the test graph with 20% edges on TW. This indicates the good scalability of the proposed algorithms.

D. Effect of Parameters on Index Time

In Exp-4, we verified that using the batch labeling optimization (forming $\mathsf{DRL_b}$) can speed up the index time of DRL. For $\mathsf{DRL_b}$, two parameters need to be set to generate the batch sequence: the initial batch size b and the incremental factor k. We set both parameters to 2 by default, but we need to further test the effect of these two parameters on the index performance (time) of $\mathsf{DRL_b}$.

Exp 7: Effect of Initial Batch Size b. We first analyze the effect of b. We vary the value of b from 1, 2, 4, 8, 16, 32, 64, to 128. We record the index time of $\mathsf{DRL_b}$ for different values of b and present the results of $\mathsf{DRL_b}$ on the first 6 graphs in Fig 8. We have the following findings.

b *has little effect on the index time.* As the value of b varies, the difference between the maximum index time and the minimum index time on all used graphs is no more than 1.5 times. This indicates that $\mathsf{DRL_b}$ is not sensitive to the parameter b.

The default value of 2 *is a good choice.* On some graphs (e.g., WEBW and DBPE), setting b to 2 leads to a local minimum index time. This explains why 2 is used as the default value.

Exp 8: Effect of Factor k. We analyze the effect of another parameter k on the index time. We vary the value of k from 1, 1.5, 2, 2.5, 3, 3.5, to 4. We report the index time of $\mathsf{DRL_b}$ for different k in Fig. 9 and obtain the following findings.

When k *is not* 1. When k is taken other than 1, the index time does not vary much: the difference between the maximum and minimum index time on all graphs does not exceed 1.4. Also, on some graphs (e.g., DBPE and CITE), the index time reaches a local minimum when k is 2, so 2 is used as the default value.

When k *is* 1. The index time becomes very slow when k is taken as 1: the index time is up to 812 times slower when k is 1 than when k is taken as other. This further explains why k needs to be set to 2 as the default value.

VII. Conclusions

We develop novel labeling methods to produce the same indexes as TOL on distributed graphs. To overcome the limitation that TOL cannot be executed in parallel, we resort to finding the backward label set of each vertex. We propose to use a filtering-and-refinement framework to find backward label sets. Using this framework, we design new labeling algorithms and further improve the efficiency by batch labeling optimization. Experimental results show that our algorithms can efficiently handle distributed graphs. The follow-up work is to maintain the indexes on distributed dynamic graphs. Another important research direction is to further explore the features of distributed and multi-core systems to accelerate the construction of indexes.

Appendix

Proof of Lemma 1. We verify only the case $v_i \in \mathsf{L_{in}}(w)$.

- $\Leftarrow$: If $\mathsf{L}^i_{\mathsf{out}}(v_i) \cap \mathsf{L}^i_{\mathsf{in}}(w) = \emptyset$, then there is no vertex u with $\mathsf{ord}(u) > \mathsf{ord}(v_i)$ such that $v_i \to u \to w$. Then, v_i is the highest-order vertex on all paths from v_i to w. By Theorem 1, $v_i \in \mathsf{L_{in}}(w)$.
- $\Rightarrow$: If $v_i \in \mathsf{L_{in}}(w)$ but $\mathsf{L}^i_{\mathsf{out}}(v_i) \cap \mathsf{L}^i_{\mathsf{in}}(w) = S \neq \emptyset$, we choose $s \in S$ whose order is the highest in S. $s \neq v_i$ since $v_i \notin \mathsf{L}^i_{\mathsf{in}}(v_i)$ by definition. Moreover, the fact that $s \in S$ yields $v_i \to s \to u$ and $\mathsf{ord}(s) > \mathsf{ord}(v_i)$. By Theorem 1, $v_i \notin \mathsf{L_{in}}(w)$, contradiction.

Acknowledgment

Lu Qin is supported by ARC FT200100787 and DP210101347. Ying Zhang is supported by ARC FT170100128 and ARC DP210101393. Xuemin Lin is supported by ARC DP180103096 and ARC DP170101628.

References

[1] S. Sahu, A. Mhedhbi, S. Salihoglu, J. Lin, and M. T. Özsu, "The ubiquity of large graphs and surprising challenges of graph processing: extended survey," *The VLDB Journal*, vol. 29, no. 2, pp. 595–618, 2020.

[2] J. X. Yu and J. Cheng, "Graph reachability queries: A survey," in *Managing and Mining Graph Data*. Springer, 2010, pp. 181–215.

[3] R. Jin and G. Wang, "Simple, fast, and scalable reachability oracle," *Proceedings of the VLDB Endowment*, vol. 6, no. 14, pp. 1978–1989, 2013.

[4] S. Beamer, K. Asanovic, and D. Patterson, "Direction-optimizing breadth-first search," in *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2012, pp. 1–10.

[5] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM journal on computing*, vol. 1, no. 2, pp. 146–160, 1972.

[6] R. Jin, N. Ruan, S. Dey, and J. Y. Xu, "Scarab: scaling reachability computation on large graphs," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, 2012, pp. 169–180.

[7] H. Yildirim, V. Chaoji, and M. J. Zaki, "Grail: Scalable reachability index for large graphs," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 276–284, 2010.

[8] H. Wei, J. X. Yu, C. Lu, and R. Jin, "Reachability querying: An independent permutation labeling approach," *Proceedings of the VLDB Endowment*, vol. 7, no. 12, pp. 1191–1202, 2014.

[9] J. Su, Q. Zhu, H. Wei, and J. X. Yu, "Reachability querying: Can it be even faster?" *IEEE Transactions on Knowledge and Data Engineering*, vol. 29, no. 3, pp. 683–697, 2016.

[10] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick, "Reachability and distance queries via 2-hop labels," *SIAM Journal on Computing*, vol. 32, no. 5, pp. 1338–1355, 2003.

[11] R. Schenkel, A. Theobald, and G. Weikum, "Hopi: An efficient connection index for complex xml document collections," in *International Conference on Extending Database Technology*. Springer, 2004, pp. 237–255.

[12] J. Cheng, J. X. Yu, X. Lin, H. Wang, and S. Y. Philip, "Fast computation of reachability labeling for large graphs," in *International Conference on Extending Database Technology*. Springer, 2006, pp. 961–979.

[13] J. Cheng, J. X. Yu, X. Lin, H. Wang, and P. S. Yu, "Fast computing reachability labelings for large graphs with high compression rate," in *Proceedings of the 11th international conference on Extending database technology: Advances in database technology*. ACM, 2008, pp. 193–204.

[14] A. D. Zhu, W. Lin, S. Wang, and X. Xiao, "Reachability queries on large dynamic graphs: a total order approach," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 2014, pp. 1323–1334.

[15] T. Zhang, Y. Gao, C. Li, C. Ge, W. Guo, and Q. Zhou, "Distributed reachability queries on massive graphs," in *International Conference on Database Systems for Advanced Applications*. Springer, 2019, pp. 406–410.

[16] W. Fan, X. Wang, and Y. Wu, "Performance guarantees for distributed reachability queries," *Proceedings of the VLDB Endowment*, vol. 5, no. 11, pp. 1304–1316, 2012.

[17] Y. Yano, T. Akiba, Y. Iwata, and Y. Yoshida, "Fast and scalable reachability queries on graphs by pruned labeling with landmarks and paths," in *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*. ACM, 2013, pp. 1601–1606.

[18] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[19] X. Feng, L. Chang, X. Lin, L. Qin, and W. Zhang, "Computing connected components with linear communication cost in pregel-like systems," in *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE, 2016, pp. 85–96.

[20] T. Zhang, Y. Gao, L. Chen, W. Guo, S. Pu, B. Zheng, and C. S. Jensen, "Efficient distributed reachability querying of massive temporal graphs," *The VLDB Journal*, vol. 28, no. 6, pp. 871–896, 2019.

[21] A. Buluç and K. Madduri, "Parallel breadth-first search on distributed memory systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011, pp. 1–12.

[22] B. Awerbuch, "A new distributed depth-first-search algorithm," *Information Processing Letters*, vol. 20, no. 3, pp. 147–150, 1985.

[23] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[24] E. Nuutila, "Efficient transitive closure computation in large digraphs." 1998.

[25] S. J. van Schaik and O. de Moor, "A memory efficient reachability data structure through bit vector compression," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. ACM, 2011, pp. 913–924.

[26] R. Agrawal, A. Borgida, and H. V. Jagadish, "Efficient management of transitive relationships in large data and knowledge bases," in *ACM SIGMOD Record*, vol. 18, no. 2. ACM, 1989, pp. 253–262.

[27] Y. Chen and Y. Chen, "An efficient algorithm for answering graph reachability queries," in *2008 IEEE 24th International Conference on Data Engineering*. IEEE, 2008, pp. 893–902.

[28] R. Jin, Y. Xiang, N. Ruan, and H. Wang, "Efficiently answering reachability queries on very large directed graphs," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM, 2008, pp. 595–608.

[29] W. Li, M. Qiao, L. Qin, Y. Zhang, L. Chang, and X. Lin, "Scaling distance labeling on small-world networks," in *Proceedings of the 2019 International Conference on Management of Data*, 2019, pp. 1060–1077.

[30] K. Lakhotia, R. Kannan, Q. Dong, and V. Prasanna, "Planting trees for scalable and efficient canonical hub labeling," *Proceedings of the VLDB Endowment*, vol. 13, no. 4.

[31] J. Cheng, S. Huang, H. Wu, and A. W.-C. Fu, "Tf-label: a topological-folding labeling scheme for reachability querying in a large graph," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM, 2013, pp. 193–204.

[32] J. Leskovec and A. Krevl, "Snap datasets: Stanford large network dataset collection," 2014.

[33] J. Kunegis, "Konect: the koblenz network collection," in *Proceedings of the 22nd International Conference on World Wide Web*. ACM, 2013, pp. 1343–1350.

[34] P. Boldi and S. Vigna, "The WebGraph framework I: Compression techniques," in *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*. Manhattan, USA: ACM Press, 2004, pp. 595–601.

[35] P. Boldi, M. Rosa, M. Santini, and S. Vigna, "Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks," in *Proceedings of the 20th international conference on World Wide Web*, S. Srinivasan, K. Ramamritham, A. Kumar, M. P. Ravindra, E. Bertino, and R. Kumar, Eds. ACM Press, 2011, pp. 587–596.

[36] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *AAAI*, 2015. [Online]. Available: http://networkrepository.com

[37] J. Zhou, S. Zhou, J. X. Yu, H. Wei, Z. Chen, and X. Tang, "Dag reduction: Fast answering reachability queries," in *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 2017, pp. 375–390.

[38] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *IEEE computational science and engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[39] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, and J. McDonald, *Parallel programming in OpenMP*. Morgan kaufmann, 2001.