Reachability Queries on Large Dynamic Graphs: A Total Order Approach

Andy Diwen Zhu[†] Wenqing Lin[†] Sibo Wang[†] Xiaokui Xiao[‡]
School of Computer Engineering
Nanyang Technological University
Singapore
[†]{diwen.zhu, keamoulin, wangsibo.victor}@gmail.com [‡]xkxiao@ntu.edu.sg

ABSTRACT

Reachability queries are a fundamental type of queries on graphs that find important applications in numerous domains. Although a plethora of techniques have been proposed for reachability queries, most of them require that the input graph is *static*, i.e., they are inapplicable to the *dynamic* graphs (e.g., social networks and the Semantic Web) commonly encountered in practice. There exist a few techniques that can handle dynamic graphs, but none of them can scale to sizable graphs without significant loss of efficiency. To address this deficiency, this paper presents a novel study on reachability indices for *large dynamic graphs*. We first introduce a general indexing framework that summarizes a family of reachability indices with the best performance among the existing techniques for static graphs. Then, we propose general and efficient algorithms for handling vertex insertions and deletions under the proposed framework. In addition, we show that our update algorithms can be used to improve the existing reachability techniques on static graphs, and we also propose a new approach for constructing a reachability index from scratch under our framework. We experimentally evaluate our solution on a large set of benchmark datasets, and we demonstrate that our solution not only supports efficient updates on dynamic graphs, but also provides even better query performance than the state-of-the-art techniques for *static* graphs.

Categories and Subject Descriptors

G.2.2 [**Graph Theory**]: Graph Algorithms

General Terms

Algorithms, Experimentation

1. INTRODUCTION

Given a directed graph G and two vertices s and t in G, a reachability query asks whether there exists a path from s to t in G. Reachability queries are a fundamental operation on graphs and have numerous important applications, such as query processing on social networks, the Semantic Web, XML documents, road networks, and program workflows. Devising index structures for reachability queries is non-trivial, as it requires a careful balancing act between pre-computation cost, index size, and query processing overhead. In particular, if we pre-compute and store the reachability results for all pairs of vertices, then we can process any reachability query in $O(1)$ time but suffer prohibitive costs of preprocessing and space. On the other hand, if we omit indexing and process reachability queries directly on G using depth-first search (DFS) or breadth-first search (BFS), then we minimize space and pre-computation overhead, but fail to ensure query efficiency on large graphs.

Previous work [3–14, 16, 19, 22–25, 27–32] has proposed numerous indexing techniques to efficiently support reachability queries without significant space and pre-computation overheads. Most techniques, however, assume that the input graph G is *static*, which makes them inapplicable for the *dynamic* graphs commonly encountered in practice. For example, the social graph of Twitter is constantly changing, with thousands of new users added per day; the Semantic Web is frequently updated with new concepts and relations; even road networks are subject to changes due to road closures and constructions. There exist a few techniques [4, 12, 13, 16, 22, 24, 32] that are designed for dynamic graphs, but as we discuss in Sections 3 and 8, none of those techniques can scale to sizable graphs without significant loss of efficiency. Specifically, the methods in [4, 12, 13, 16, 22, 24] incur prohibitive preprocessing costs on graphs with more than one million vertices. Meanwhile, the approach in [32] can handle million-vertex graphs, but it offers a query performance that is *generally not much better than a simple BFS approach*, as shown in our experiments.

In summary, no existing method is able to effectively handle reachability queries on *large dynamic graphs*. Motivated by this, we present a comprehensive study on scalable reachability indices that support updates. We first introduce a *total order labeling (TOL)* framework, which summarizes three most advanced methods [8, 17, 30] for reachability queries on static graphs. TOL has two important properties: (i) every reachability index under TOL uniquely corresponds to a *total order* of vertices in the input graph, and (ii) the total order *solely* decides the index's performances in terms of preprocessing, space, and queries. Given these properties, we investigate algorithms that enable us to insert or delete a vertex in a TOL index without changing the order of the other vertices, i.e., without significantly degrading the performance of the index. This results in general algorithms for handling insertions and deletions on indices under TOL. In particular, our insertion algorithm is *optimal* in that it leads to the minimum index size after insertion.

Interestingly, we observe that our update algorithms can be utilized to reduce the space consumptions and query costs of a TOL index, by adjusting the total order pertinent to the index. This leads to a general approach for improving any index under TOL, includ-

SIGMOD'14, June 22–27, 2014, Snowbird, UT, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2376-5/14/06 ...$15.00.
http://dx.doi.org/10.1145/2588555.2612181.

ing the state-of-the-art techniques [8, 17, 30]. The effectiveness of our adjusting approach shows that the total orders of the techniques in [8, 17, 30] leave much room for enhancement, which motivates us to devise new methods for deriving improved total orders for TOL indices. As a result, we present a new reachability index, *Butterfly*, which offers reduced preprocessing, space, and query costs than any existing indices under TOL [8, 17, 30]. We experimentally evaluate TOL using a large variety of benchmark datasets with up to twenty million vertices, and we demonstrate the superiority of TOL against alternative solutions for static and dynamic graphs.

In summary, this paper makes the following contributions:

- We propose general and efficient algorithms that enable *any* index under the TOL framework to support *large dynamic graphs* (Section 5).
- We develop a technique that can postprocess the state-of-the-art reachability indices [8, 17, 30] to significantly enhance their performances in terms of space overheads and query efficiency (Section 6).
- We devise algorithms to derive improved vertex ordering under TOL, based on which we propose Butterfly, a new reachability index that dominates the states of the art (Section 7).
- We evaluate our solution on a large set of real and synthetic graphs, and we demonstrate that our solution not only supports efficient updates on large dynamic graphs, but also provides even better query performance than the state-of-the-art techniques for *static* graphs (Section 8).

2. PRELIMINARIES

Let $G = (V, E)$ be a directed graph with a set V of vertices and a set E of edges. For any two vertices s and t in V, we say that s can *reach* t (denoted as $s \to t$), iff there exists a directed path in G that starts from s and ends at t. Given s and t, a reachability query returns TRUE if $s \to t$, and FALSE otherwise. We refer to s and t as the *source vertex* and *terminal vertex* of the query, respectively.

If $s \to t$ and $t \to s$ both holds, then s and t are *strongly connected*. Accordingly, a *strongly connected component (SCC)* of G is defined as a maximal subset of V where any two vertices are strongly connected. Observe that a vertex u can reach another vertex v, iff either of the following conditions holds: (i) u and v belong to the same SCC, or (ii) there is a path that starts from the SCC containing u to the SCC containing v. Given this observation, there exists a simple method that reduces G into a smaller graph G^* to improve the efficiency of reachability queries:

1. Compute all SCCs of G. (This can be done in $O(|V| + |E|)$ time [26].
2. Map each SCC C to a vertex $f(C)$. For any two SCCs C and C', if G contains an edge that starts at a vertex in C and ends at a vertex in C', then construct an directed edge from $f(C)$ to $f(C')$. Denote the resulting graph as G^*.
3. Given a reachability query from s to t on G, we first retrieve the SSC S (resp. T) of G that contains s (resp. t). Then, we return TRUE for the query, if and only if (i) S and T are the same or (ii) $f(S)$ can reach $f(T)$ in G^*.

In the remainder of the paper, we assume that G has been preprocessed with the above reduction method, i.e., G does not contain any strongly connected component with more than one vertex. (The same assumption is made in existing work [].) Under this assumption, G should be a *directed acyclic graph (DAG)*. In addition, there exists a total order o on V, such that for any two vertices u and v in G, if $u \to v$ then $o(u) < o(v)$ (but not necessarily vice versa). Such a total order can be derived in linear time using a DFS on G [8]. We refer to o as a *topological order*, and $o(u)$ as the *topological rank* of u. For ease of reference, Table 1 lists the notations that will be frequently used in this paper.

Notation	Description
$G = (V, E)$	a DAG with a vertex set V and an edge set E
$o(v)$	the topological rank of a vertex v (see Section 2)
$N_{in}(v)$	the set of v's in-neighbors
$N_{out}(v)$	the set of v's out-neighbors
$l(v)$	the level of v (see Section 4)
$\mathcal{L}$	a total order labeling of G
$L_{in}(v)$	the in-label set of vertex v (see Definition 1)
$L_{out}(v)$	the out-label set of vertex v (see Definition 1)
$I_{in}(v)$	the inverted index for v's in-labels (see Equation 3)
$I_{out}(v)$	the inverted index for v's out-labels (see Equation 4)

Table 1: Table of notations.

3. RELATED WORK

The existing work on reachability queries can be roughly divided into three categories (based on their query processing schemes): *pruned depth-first search*, *transitive closure retrieval*, and *two-hop label matching*. In the following, we survey the techniques in each category for static graphs, and then discuss the existing methods on dynamic graphs.

Pruned Depth-first Search. Techniques [6, 27, 31] in this category process each reachabilty query using DFS on G, but they precompute certain auxiliary information on G to prune the search space of DFS, which helps improve query efficiency. The state-of-the-art algorithm in this category is GRAIL [31]. It preprocesses G by assigning an interval to each vertex, such that, for any two vertices u and v, if the interval of u does not fully contain the interval of v, then $u \not\to v$ must hold. Given such intervals, GRAIL answers any reachability query using a DFS from the source vertex, and it avoids visiting any vertex whose interval does not cover the terminal vertex's interval. GRAIL is shown to incur small overheads of preprocessing and space, but its query efficiency is generally much worse than methods in the other two categories.

Transitive Closure Retrieval. The *transitive closure* of a vertex u is defined as the set of all vertices that u can reach in G. Methods based on transitive closure retrieval [3, 7, 8, 10, 14, 19, 25, 28, 29] pre-compute and compress the transitive closures of each vertex in G. Given any reachability query, they first retrieve the transitive closure of the source vertex, decompress it, and then check whether the terminal vertex is contained in the transitive closure. Such methods are generally efficient for query processing, but they cannot scale to large graphs due to the significant pre-computation and space overheads in deriving and storing transitive closures.

2-Hop Label Matching. Techniques in this category [5, 8–11, 23, 30] preprocess G by constructing two sets of *labels* for each vertex v, namely, an *out-label set* $L_{out}(v)$ and an *in-label set* $L_{in}(v)$. Specifically, each label in $L_{out}(v)$ is a vertex in G that v can reach, while $L_{in}(v)$ contains a subset of the vertices in G that can reach v. The label sets are created in a way such that, for any two vertices s and t, we have $s \to t$ if and only if $L_{out}(s) \cap L_{in}(t) \neq \emptyset$. In other words, any reachability query can be simply answered by taking the intersection of the source vertex's out-label set and the terminal vertex's in-label set. This leads to high query efficiency, *when the label sets are small*.

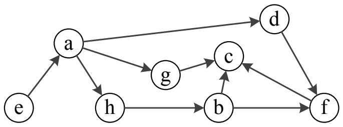

Figure 1: DAG G.

Cohen et al. [11] present the first study on 2-hop labelling techniques. They prove that it is generally NP-hard to minimize the total size of label sets, and they propose a $(\log |V|)$-approximate solution to the problem. However, the approximate solution requires $O(|V| \cdot |E| \log(|V|^2/|E|))$ preprocessing time, which renders it inapplicable for sizable graphs. Motivated by this, numerous heuristic techniques [2, 5, 8–10, 15, 18, 23, 30] have been proposed to improve the practical efficiency of 2-hop labelling, albeit discarding the approximation guarantee of Cohen et al.'s method). Among those techniques, the most advanced ones are *TF-Label* [8], *Distribution Labeling (DL)* [17], and *Pruned Landmark Labeling (PLL)* [30], as they are shown to provide better overall performance than other existing methods (including the ones based on pruned DFS or transitive closure retrieval). In Section 4, we will present a framework that summarizes TF-Label, DL, and PLL.

Methods for Dynamic Graphs. While the aforementioned techniques all focus on static graphs, there also exist a few studies [4, 12, 13, 16, 21, 22, 24] on reachability indices for dynamic graphs. In particular, [12, 13, 21, 22] propose algorithms for incrementally maintaining transitive closures on dynamic graphs. Those algorithms, however, cannot scale to graphs with more than a few thousand vertices, as shown by Krommidias et al. [20]. There also exist two methods [4, 24] that extend Cohen et al.'s 2-hop labeling approach [11] to handle updates. Nevertheless, the method in [24] is restricted to XML graphs, while [4] is unable to handle any of the million-node graphs used in our experiments. In addition, [16] presents an algorithm for performing updates on a reachability index, but the index itself incurs tremendous preprocessing costs on large graphs. Very recently, Yildirim et al. propose *Dagger* [32], an extension of GRAIL [31] that supports dynamic graphs. As we show in Section 8, however, Dagger's query performance is up to 10^7 times worse than the solution in this paper, and is generally not much better than a simple BFS approach.

4. TOTAL ORDER LABELING

This section presents *total order labeling (TOL)*, a reachability indexing framework that can be instantiated into various 2-hop labeling indices. The instantiation of TOL requires two input parameters, namely, a DAG $G = (V, E)$ and a *strict* total order l on V. We refer to l as a *level order*. For any vertex v, we define $l(v) \in [1, |V|]$ as the rank of v in l, and refer to $l(v)$ as the *level* of v. In addition, we say that v has a *higher* (resp. lower) level than another vertex u, if $l(v) < l(u)$ (resp. $l(v) > l(u)$).

Definition of TOL Indices. Given G and l, TOL uniquely defines a 2-hop labeling index $\mathcal{L}$ on G as follows:

DEFINITION 1 (TOTAL ORDER LABELING $\mathcal{L}$). *Given a DAG $G = (V, E)$ and a level order l, a TOL index $\mathcal{L}$ is a 2-hop labeling index where each vertex is associated with an in-label set $L_{in}(v)$ and an out-label set $L_{out}(v)$, such that $L_{in}(v)$ (resp. $L_{out}(v)$) contains every vertex u satisfying all of the following constraints:*

- *Reachability Constraint: $u \to v$ (resp. $v \to u$);*
- *Level Constraint: $l(u) < l(v)$;*
- *Path Constraint: No simple path from u to v (resp. from v to u) in G contains a vertex w with $l(w) < l(u)$.*

	TOL index $\mathcal{L}_1$			TOL index $\mathcal{L}_2$		
v	$l_1(v)$	$L_{in}(v)$	$L_{out}(v)$	$l_2(v)$	$L_{in}(v)$	$L_{out}(v)$
a	1	∅	∅	7	∅	{b,c,d,f,g,h}
b	2	{a}	∅	4	∅	{c,f}
c	3	{a,b}	∅	2	{g}	∅
d	4	{a}	{c}	5	∅	{c,f}
e	5	∅	{a}	8	∅	{a,b,c,d,f,g,h}
f	6	{a,b,d}	{c}	3	∅	{c}
g	7	{a}	{c}	1	∅	∅
h	8	{a}	{b}	6	∅	{b,c,f}

Table 2: Two TOL indices for the DAG G in Figure 1.

We illustrate Definition 1 with the following example.

EXAMPLE 1. Given the DAG G shown in Figure 1, Table 2 shows two TOL indices on G (i.e., $\mathcal{L}_1$ and $\mathcal{L}_2$) with level orders l_1 and l_2, respectively. Consider vertex g in G. Its in-label set in $\mathcal{L}_1$ contains a since (i) a can reach g, (ii) a has a higher level than g, and (iii) G contains only one simple path from a to g, and the path does not contain any vertex with a higher level than a. In addition, g's in-label set does not contain any vertex other than a, since the path from any other vertex to g must pass through a, whereas a has the smallest level in l_1, i.e., adding any other vertex to $L_{in}(g)$ violates the Path Constraint in Definition 1.

In contrast, g's in-label set in $\mathcal{L}_2$ is ∅. This is because g has the smallest level in l_2, due to which we cannot add any vertex into g's in-label set without violating the Level Constraint in Definition 1. In general, the label sets in $\mathcal{L}_1$ are drastically different from their counterparts in $\mathcal{L}_2$, due to the differences between l_1 and l_2. □

As demonstrated in Example 1, the level order l used to instantiate TOL has a profound effect on resulting reachability index. Therefore, if we are to obtain a TOL index with high efficiency, it is essential that we choose an appropriate level order l. In Sections 5 and 7, we will discuss how a good level order can be derived and incrementally maintained for dynamic graphs. For convenience, we define the size of a TOL index $\mathcal{L}$ as the total size of the label sets in $\mathcal{L}$, i.e.,

$$|\mathcal{L}| = \sum_{v \in V} \big(|L_{in}(v)| + |L_{out}(v)|\big).$$

Query Algorithm. Given a reachability query from a vertex s to another vertex t, a TOL index $\mathcal{L}$ processes the query in the same way as other 2-hop labeling methods do. In particular, $\mathcal{L}$ first retrieves the out-label sets of s and the in-label set of t, and then computes a *witness set* as follows:

$$W(s,t) = \big(L_{out}(s) \cup \{s\}\big) \cap \big(L_{in}(t) \cup \{t\}\big). \quad (1)$$

If the witness set is empty, then $\mathcal{L}$ returns FALSE for the query; otherwise, $\mathcal{L}$ returns TRUE. The following lemma shows the correctness of the above query processing approach.

LEMMA 1. *Given any two vertices s and t, we have $W(s,t) \neq \emptyset$ iff $s \to t$ in G.*

PROOF. We first prove that $W(s,t) \neq \emptyset$ implies $s \to t$. Observe that, when $W(s,t) \neq \emptyset$, at least one of the following three cases must hold: (i) $t \in L_{out}(s)$, (ii) $s \in L_{in}(t)$, and (iii) $L_{in}(t) \cap L_{out}(s) \neq \emptyset$. By Definition 1, both $t \in L_{out}(s)$ and $s \in L_{in}(t)$ indicate that $s \to t$. Meanwhile, if $L_{out}(s) \cap L_{in}(t) \neq$

$\emptyset$, then for any $u \in L_{out}(s) \cap L_{in}(t)$, we have $s \rightarrow u$ and $u \rightarrow t$, which leads to $s \rightarrow t$.

Next, we show that if $s \rightarrow t$, then $W(s,t) \neq \emptyset$. Consider the set of all simple paths from s to t in G. Let v be the vertex with the highest level among all vertices on those paths. We differentiate three cases: (i) $v = t$, (ii) $v = s$, and (iii) $v \neq s$ and $v \neq t$. If $v = t$, then by Definition 1, we have $t \in L_{out}(s)$, in which case $W(s,t)$ contains t, i.e., $W(s,t) \neq \emptyset$. Meanwhile, if $v = s$, then $s \in L_{in}(t)$ holds, which leads to $W(s,t) \supseteq \{s\} \neq \emptyset$. Finally, if $v \neq s$ and $v \neq t$, then v must appear in both $L_{out}(s)$ and $L_{in}(t)$, in which case $W(s,t) \supseteq \{v\} \neq \emptyset$. □

Interestingly, the label sets in any TOL index $\mathcal{L}$ are *minimal*, in the sense that no label can be removed without affecting the correctness of TOL's query processing algorithm:

LEMMA 2. *Let $\mathcal{L}$ be a TOL index on G with a level ordering l. For any vertex u, if we remove a vertex v_1 from $L_{out}(u)$, then $W(u, v_1) = \emptyset$ but $u \rightarrow v_1$. Meanwhile, if we remove a vertex v_2 from $L_{in}(u)$, then $W(v_2, u) = \emptyset$ but $v_2 \rightarrow u$.*

PROOF. Consider vertex v_1. Given that $v_1 \in L_{out}(u)$, we have $u \rightarrow v_1$ and $l(u) > l(v_1)$ by Definition 1. Since $l(u) > l(v_1)$, we have $u \notin L_{in}(v_1)$ by the Level Constraint in Definition 1.

Assume on the contrary that, after v_1 is removed from $L_{out}(u)$, $W(u, v_1) \neq \emptyset$. Then, by Equation 1 and $u \notin L_{in}(v_1)$, there must exist a vertex $w \in L_{out}(u) \cap L_{in}(v_1)$. In that case, G must contain a simple path from u to v_1 via w, and w must have a higher level than both u and v_1. This contradicts the Path Constraint in Definition 1 since $v_1 \in L_{out}(u)$ initially holds. The case for vertex v_2 can be established in a similar manner. □

Existing Instantiations of TOL. By Definition 1, TOL defines a family of 2-hop labeling approaches that satisfy the Reachability, Level, and Path Constraints. This index family does not include all existing 2-hop labeling methods (as many of them violate the aforementioned constraints), but it captures the three most advanced 2-hop labeling techniques, i.e., TF-Label [8], DL [17], and PLL [30]. In particular, TF-Label utilizes a topological order o of the vertices in G (see Section 2 for the definition of o). It constructs indices in way such that (i) a label set of a vertex v only contains vertices u with $o(v) < o(u)$, and (ii) the label sets are minimal. It can be shown the TF-Label corresponds to a TOL index that uses o as the level order (with ties broken arbitrarily when multiple vertices have the same rank in o).

Meanwhile, DL sorts the vertices in G in descending order of their degrees, and it follows the sorted order to inspect vertices in G and constructs label sets accordingly. Specifically, each time it examines a vertex v, it uses two constrained BFSs on G to identify a number of vertices that (i) are connected to v and (ii) rank lower than v in the sorted order; then, it adds v into the label sets of those vertices. It can be proved that DL is equivalent to a TOL index where the level order ranks vertices in descending order of their degrees. Finally, as PPL is shown to be equivalent to DL [17], it is also an instantiation of TOL.

It is noteworthy that, if we modified the vertex order in DL, and use the modified order to construct a reachability index based on DL's preprocessing algorithm, then the resulting index is equivalent to a TOL index adopting the same modified order. In other words, any TOL index can be obtained using a modified version of DL's pre-computation algorithm. Nevertheless, the paper that describes DL [17] does not summarize the Reachability, Level, and Path Constraints (see Definition 1) that underpin TOL. The summarization of those constraints is crucial in the context of our paper, as they are imperative in our analysis on how TOL indices can be incrementally updated (see Section 5).

5. INCREMENTAL UPDATES

In this section, we study how a TOL index $\mathcal{L}$ can be incrementally updated when a vertex is inserted into or deleted from G. Our objective is twofold:

1. *$\mathcal{L}$ should remain a TOL index after any update*, i.e., it should always satisfy the constraints in Definition 1. This is to ensure the correctness of $\mathcal{L}$'s query algorithm and the minimality of $\mathcal{L}$'s label sets
2. *Inserting or deleting a vertex should not change the level order l on the other vertices.* Intuitively, this reduces the amount of changes required in the label sets of $\mathcal{L}$, and helps retain the performance of $\mathcal{L}$ after the update, since a TOL index's label sets (and thus, its performance) are solely decided by its level order.

5.1 Insertion Algorithm

Consider that we insert a new vertex v into G and connect v with some existing vertices in G. Let $G' = (V', E')$ be the graph obtained after the insertion. Following previous work [4, 32], we assume that G' is also a DAG. The case when G' is not a DAG can be handled by incrementally maintaining the strongly connected components in G, as discussed in [32]. Let $\mathcal{L}$ be a TOL index on G with a level order l. As mentioned, our objective is to update $\mathcal{L}$ into a TOL index $\mathcal{L}'$ on G' with a level order l', such that for any two vertices $u_1, u_2 \in V$, $l(u_1) < l(u_2)$ iff $l'(u_1) < l'(u_2)$.

In a nutshell, our insertion algorithm runs in two steps: In Step 1, it decides the value of $l'(v)$, and then sets $l'(u)$ for any vertex u in G as follows:

$$l'(u) = \begin{cases} l(u) & \text{if } l(u) < l'(v) \\ l(u) + 1 & \text{otherwise} \end{cases} \tag{2}$$

Then, in Step 2, it updates the label sets in $\mathcal{L}$ according to l', which transforms $\mathcal{L}$ into $\mathcal{L}'$. For ease of exposition, we will first elaborate Step 2, assuming that l' has been constructed.

5.1.1 Step 2: Updating Label Sets

Given G', $\mathcal{L}$, and l', the second step of our insertion algorithm is further divided into two sub-steps. In Step 2.1, we create the label sets $L'_{in}(v)$ and $L'_{out}(v)$ for the new vertex v, and insert v into the label sets of other vertices. Then, in Step 2.2, we further refine the label sets of the vertices in V, so as to convert $\mathcal{L}$ into $\mathcal{L}'$. Throughout the algorithm, for each vertex u, we maintain two *inverted lists* $I_{in}(u)$ and $I_{out}(u)$, such that

$$I_{in}(u) = \{w \mid u \in L_{in}(w)\}, \tag{3}$$

$$I_{out}(u) = \{w \mid u \in L_{out}(w)\}. \tag{4}$$

In other words, if u appears in the in-label (resp. out-label) set of a vertex w, then w is recorded in the inverted list $I_{in}(u)$ (resp. $I_{out}(u)$). These inverted lists enable us to efficiently identify the label sets that are affected by any vertex u. In addition, they can be easily maintained with respect to changes in the label sets.

Step 2.1. Algorithm 1 shows the pseudo-code for the first sub-step. The algorithm first creates a *candidate set* $C_{in}(v)$ (Lines 1-4), and then refines it into the in-label set $L'_{in}(v)$ of v (Lines 5-10). In particular, the candidate set $C_{in}(v)$ contains all in-neighbors of v (i.e., the starting vertices of the edges pointing to v), as well as the in-label sets of those in-neighbors (Line 3-4).

By Definition 1, $C_{in}(v)$ is a superset of v's in-labels in $\mathcal{L}'$. To explain, consider a vertex u that is an in-label of v in $\mathcal{L}'$. By the Reachability Constraint, there exists a path P from u to v in G'. Let w be the in-neighbor of v on P. Then, u can reach w. In addition,

Algorithm 1: INSERT-STEP-2.1

```
input  : $G'$, $\mathcal{L}$, $l'$, and $v$
output: $L'_{in}(v)$, $L'_{out}(v)$, and a modified version of $\mathcal{L}$
1  create two candidate label sets $C_{in}(v)$ and $C_{out}(v)$ ;
2  set $L'_{in}(v) = L'_{out}(v) = C_{in}(v) = C_{out}(v) = \emptyset$ ;
3  for each of $v$'s in-neighbors $u$ do
4      $C_{in}(v) = C_{in}(v) \cup L_{in}(u) \cup \{u\}$ ;
5  for each $u \in C_{in}(v)$ in ascending order of $l'(u)$ do
6      if $L_{out}(u) \cap L'_{in}(v) = \emptyset$ then
7          if $l'(u) < l'(v)$ then
8              add $u$ into $L'_{in}(v)$ ;
9          else
10             add $v$ into $L_{out}(u)$ ;
11 for each of $v$'s out-neighbors $u$ do
12     $C_{out}(v) = C_{out}(v) \cup L_{out}(u) \cup \{u\}$ ;
13 Repeat Lines 5-10 with subscripts "in" and "out" exchanged ;
14 return $L'_{in}(v)$, $L'_{out}(v)$, and the label sets in $\mathcal{L}$ ;
```

Algorithm 2: INSERT-STEP-2.2

```
input  : $G'$, $l'$, $v$, and the output of Algorithm 1
output: A TOL index $\mathcal{L}'$ for $G'$ with a level order $l'$
1  for each vertex $u \in L'_{in}(v)$ in ascending order of $l'(u)$ do
2      for each vertex $w \in L'_{out}(v) \cup \{v\}$ with $l'(w) > l'(u)$ do
3          if $L_{out}(u) \cap L_{in}(w) = \emptyset$ then
4              $L_{in}(w) = L_{in}(w) \cup \{u\}$;
5          for each vertex $x \in I_{in}(w)$ do
6              if $L_{out}(u) \cap L_{in}(x) = \emptyset$ then
7                  $L_{in}(x) = L_{in}(x) \cup \{u\}$;
8      for each vertex $x' \in I_{in}(u)$ do
9          for each vertex $y' \in I_{out}(u)$ do
10             if $y' \in L_{in}(x')$ then
11                 remove $y'$ from $L_{in}(x')$;
12             if $x' \in L_{out}(y')$ then
13                 remove $x'$ from $L_{out}(y')$;
14 Repeat Lines 1-14 with subscripts "in" and "out" exchanged ;
15 return the revised label sets ;
```

$l(u) < l(w)$; otherwise, w is a vertex on P with a higher level than u, which violates the Path Constraint. Finally, all paths from u to w should contain only vertices with levels lower than u; otherwise, u should not be an in-label of v in the first place. All of the above indicates that u is an in-label of w, as it fulfills the three constraints in Definition 1. Therefore, $C_{in}(v)$ is superset of v's in-labels in $\mathcal{L}'$.

To refine $C_{in}(v)$ into $L'_{in}(v)$, Algorithm 1 removes the vertices in $C_{in}(v)$ that violate any of the Level and Path Constraints in Definition 1 (Lines 5-10). (The Reachability Constraint is ignored as all vertices in $C_{in}(v)$ can reach v.) Specifically, the algorithm examines the vertices in $C_{in}(v)$ in ascending order of their level values. For each vertex u with $l'(u) < l'(v)$ (i.e., u satisfies the Level Constraint), the algorithm adds u into $L'_{in}(v)$ if $L_{out}(u) \cap L'_{in}(v) = \emptyset$. The rationale is that, if $L_{out}(u) \cap L'_{in}(v) = \emptyset$, then no vertex with a higher level than $l(u)$ can connect u to v, in which case u fulfills the Path Constraint. Meanwhile, if $L_{out}(u) \cap L'_{in}(v) = \emptyset$ but $l'(u) > l'(v)$, then we add v into u's out-label set $L_{out}(u)$ instead.

After $L'_{in}(v)$ is created, Algorithm 1 constructs $L'_{out}(v)$ (Lines 11-13) and then terminates. We omit the discussion on $L'_{out}(v)$ as it is similar to the case of $L'_{in}(v)$.

Step 2.2. Given Step 2.1's output, Step 2.2 of our algorithm proceeds to refine the label sets in $\mathcal{L}$, as shown in Algorithm 2. The rationale is that, since the insertion of v creates new paths among the vertices in V, we may need to adjust the label sets in $\mathcal{L}$ to ensure that $\mathcal{L}$ remains a TOL index. Specifically, Algorithm 2 first inspects the vertices in $L'_{in}(v)$ in ascending order of their level values (Line 1). For each vertex u, the algorithm examines each vertex w in $L'_{out}(v) \cup \{v\}$ with lower levels than u (Line 2). Notice that, for every such pair of vertices u and w, the insertion of v creates a new path from u to w via v. Accordingly, the algorithm adds u into w's in-label set $L_{in}(w)$ if $L_{out}(u) \cap L_{in}(w) = \emptyset$, i.e., if inserting u into $L_{in}(w)$ does not violate the Path Constraint in Definition 1 (Lines 3-4). Similarly, the algorithm also inserts u into $L_{in}(x)$ for any vertex $x \in I_{in}(w)$, i.e., any vertex x that has w as an in-label (Line 5-7).

After that, the algorithm proceeds to check whether the operations in Lines 2-7 have resulted in unnecessary labels (Lines 8-13). In particular, it examines each pair of vertices $x' \in I_{out}(u)$ and $y' \in I_{in}(u)$, i.e., x' has u as an out-label and y' has u as an in-label. For each pair of x' and y', the algorithm checks whether y' appears in the out-label of x'; if so, it removes y' from $L_{out}(x')$, since (i) u has higher level than both x' and y' and (ii) u connects x' to y', which leads to a violation of the Path Constraint. Similarly, if x' is in the out-label set of y', and it is removed.

Once the above nested loop is finished, Algorithm 2 enters another nested-loop, where (i) the outer loop linearly scans each vertex u in $L'_{out}(v)$ in ascending order of level values, and (ii) the inner loop examines each vertex w in $L'_{in}(v) \cup \{v\}$ with $l'(u) < l'(w)$. This nested loop complements the previous nested loop, in that the former processes vertex pairs in $L'_{in}(v)$ and $L'_{out}(v)$ that are ignored by the latter. Finally, Algorithm 2 terminates and returns the revised label sets, which constitute $\mathcal{L}'$.

5.1.2 Step 1: Deciding Vertex Level

Our algorithms in Section 5.1.1 require that the level $l'(v)$ of the new vertex v is decided. A straightforward approach is to set $l'(v) = |V| + 1$, i.e., give v the lowest possible level. This leads to relatively small update overheads because, when $l'(v)$ is maximized, we do not need to insert v into the label sets of any other vertex (due to the Level Constraint in Definition 1). In terms of space overhead and query efficiency, however, setting $l'(v) = |V| + 1$ could be highly sub-optimal than other choices $l'(v)$. To address this issue, we present an alternative solution that sets $l'(v)$ to a value that minimizes the total size of the label sets. Such a $l'(v)$ is also likely to improve query efficiency, since the cost of a reachability query on a TOL index is linear to the sizes of the source and terminal vertices' label sets.

Let $\mathcal{L}_k$ be the TOL index obtained by inserting v into $\mathcal{L}$ with $l'(v) = k$. To identify the value of k that minimizes $\mathcal{L}$, we examine all possible $k \in [1, |V|+1]$, but avoid repeatedly using Algorithms 1 and 2 to construct all $\mathcal{L}_k$. Instead, we propose a lightweight approach for deriving

$$\Delta_k = |\mathcal{L}_k| - |\mathcal{L}_{k+1}| \tag{5}$$

for any $k \in [1, |V|]$. Once Δ_k is computed, we can easily determine the optimal value of $l'(v)$.

The key observation behind our approach is as follows. When we change $l'(v)$ from k to $k-1$, the level order of all vertices remain unchanged, except for v and the vertex u with $l(u) = k-1$ (since the order between u and v would be reversed). As a consequence, the size difference between $\mathcal{L}_k$ and $\mathcal{L}_{k-1}$ only depends on the label sets that concern u and v. Intuitively, tracking the changes in those label sets is much simpler than creating a TOL index from

Algorithm 3: INSERT-STEP-1

```
    input  : $G = (V, E)$, $\mathcal{L}$, $l$, and $v$
    output: the value for $l'(v)$ that minimizes $|\mathcal{L}'|$
1   set $l'(v) = |V| + 1$ ;
2   construct $L'_{in}(v)$, $L'_{out}(v)$, $I'_{in}(v)$, and $I'_{out}(v)$ as in Algorithm 1
    (without materializing any changes to $\mathcal{L}$) ;
3   for $k = |V|, |V| - 1, \cdots, 1$ do
4       $\Delta_k = 0$ ;
5       let $u$ be the vertex with $l(u) = k$ ;
6       if $u \in L'_{in}(v)$ then
7           remove $u$ from $L'_{in}(v)$ and add it into $I'_{out}(v)$ ;
8           for each vertex $w \in I'_{in}(v)$ such that $u \in L_{in}(w)$ do
9               $\Delta_k = \Delta_k - 1$ ;
10          for each vertex $w' \in I_{out}(u)$ such that $v \notin L_{out}(w)$ do
11              $\Delta_k = \Delta_k + 1$ ;
12              add $w'$ into $I'_{out}(v)$ ;
13      repeat Lines 6-12 with subscripts "in" and "out" exchanged ;
14  initialize variables $\theta_1, \theta_2, \ldots, \theta_{|V|+1}$ ;
15  $\theta_{|V|+1} = 0$ ;
16  for $k = |V|, |V| - 1, \cdots, 1$ do
17      $\theta_k = \theta_{k+1} + \Delta_k$ ;
18  return $\arg\min_k\{\theta_k\}$ ;
```

scratch, and hence, deriving Δ_k can be much more efficient than constructing $\mathcal{L}_k$.

Algorithm 3 shows the pseudo-code of our approach. It first sets $l'(v) = |V| + 1$, and applies Algorithm 1 to compute, for v, two label sets $L'_{in}(v)$ and $L'_{out}(v)$ and two inverted lists $I'_{in}(v)$ and $I'_{out}(v)$ (Lines 1-2). The subsequent part of the algorithm consists of $|V|$ iterations (Lines 3-14). In the $(|V| - k + 1)$-th iteration, the algorithm considers the case when $l'(v)$ changes from $k + 1$ to k, and evaluates the corresponding changes in the label sets, based on which it derives Δ_k.

Specifically, the algorithm first sets $\Delta_k = 0$ and inspects the vertex u with $l(u) = k$, i.e., the vertex whose level is to be exchanged with v when $l'(v)$ is decreased from $k + 1$ to k. Observe that, if the exchange between u and v leads to changes in some label sets, then $u \in L'_{in}(v) \cup L'_{out}(v)$ should hold. The reason is that, when $u \notin L'_{in}(v) \cup L'_{out}(v)$, either (i) there is no path between u and v or (ii) all paths between u and v contain at least one vertex with higher level than u and v. In either case, switching levels between u and v would not lead to violations of the Reachability, Level, or Path Constraint in any label sets. Therefore, if $u \notin L'_{in}(v) \cup L'_{out}(v)$, then no label set would be affected by swapping u and v's levels. Based on this analysis, Algorithm 3 sets the final value of Δ_k to 0, whenever $u \notin L'_{in}(v) \cup L'_{out}(v)$ (Lines 4-13).

Now consider that $u \in L'_{in}(v)$. After we exchange u and v's levels, u should be removed from $L'_{in}(v)$, and v should become an out-label of u. This explains Line 7 in Algorithm 3. Meanwhile, for any vertex $w \in I'_{in}(v)$ (i.e., w has v as an in-label), we check if u is an in-label of w (Line 8). If $u \in L_{in}(w)$, then after the levels of u and v are swapped, u should be removed from $L_{in}(w)$ due to the Path Constraint, which reduces the size of $L_{in}(w)$ by one. Accordingly, Algorithm 3 decreases Δ_k by 1 for each such vertex w (Line 9). In addition, for any vertex $w' \in I_{out}(u)$ (i.e., u is an out-label for w'), we examine if v is not out-label of w' (Line 10). If $v \notin L_{out}(w')$, then after we swap u and v's levels, v will become an out-label of w', i.e., the size of $L_{out}(w')$ is increased by one. Therefore, for each such vertex w', Algorithm 3 increases Δ_k by 1 and inserts w' into $I'_{out}(v)$ (Lines 11 and 12). It can be verified that, apart from the label sets mentioned above, no other label sets would be affected by the exchange between u and v.

Although the above discussion assumes $u \in L'_{in}(v)$, it can be easily extended to the case when $u \in L'_{out}(v)$ (Line 13). Once all Δ_k are obtained, Algorithm 3 derives $|V| + 1$ variables $\theta_1, \theta_2, \ldots, \theta_{|V|+1}$, such that $\theta_{|V|+1} = 0$ and $\theta_k = \theta_{k+1} + \Delta_k$ ($k \in [1, |V|]$). By the definition of Δ_k, the value of k that minimizes θ_k should also minimize $|\mathcal{L}_k|$. Accordingly, Algorithm 3 terminates by returning $\arg\min_k\{\theta_k\}$ (Line 18).

5.1.3 Correctness and Complexity

We first show the correctness of our insertion algorithm in Lemma 3, and then analyze its complexity.

LEMMA 3. *Given G and a new vertex v, Algorithms 1 and 2 produces a TOL index on G′, and Algorithm 3 computes a level for v that minimizes the label size of $\mathcal{L}'$.*

PROOF. We start by proving that the index $\mathcal{L}'$ produced by Algorithm 1 and 2 is a TOL index on G'. In particular, we first show that Algorithm 1 and Lines 1-7 in Algorithm 2 create label sets that are supersets of corresponding label sets in the TOL on G', and then show that Lines 8-13 in Algorithm 2 remove all redundant labels.

Recall that we have shown the correctness of $L'_{in}(v)$ and $L'_{out}(v)$ created by Algorithm 1 in Section 5.1.1. In the following, we prove that, for any vertex u other than v, Lines 1-7 in Algorithm 2 create an in-label set that is a superset of u's in-label set in the TOL on G'. Given two vertices u and x, according to Definition 1, the insertion of v causes u to become an in-label of x, only if all of the following conditions hold: (i) $l'(u) < l'(x)$; (ii) u can reach v in G' and v can reach x in G'; (iii) no simple path from u to x contains a vertex that has a higher level than u. Accordingly, in Line 1, we omit any vertex u that is not in $L'_{in}(v)$. This is because if u is not in $L'_{in}(v)$, then for the vertex z with the highest level among all the paths from u to v, we have $z \neq u$. In addition, z is also on the path from u to any x that can be reached by v. In that case, condition (iii) is violated, and hence, u will never become an in-label of any vertex in G'.

Next, consider any vertex x eliminated by Line 2. If x has a higher level than u, then it should be eliminated as it violates condition (i). On the other hand, if x has lower level than u, then x is not in $L'_{out}(v) \cup \{v\}$. In that case, for the vertex z with the highest level in all the paths from v to x be z, we have $z \in L'_{out}(v)$. If $l'(z) > l'(u)$, then x is examined in the loop when $w = z$. If $l'(z) < l'(u)$, then there is a path from u to x that contains a vertex (i.e, z) that has higher level than u, violating condition (iii). In addition, Line 3 and Line 6 also guarantee that the newly created labels do not violate condition (iii). In summary, Lines 1-7 only omits vertices that will never lead to a label creation, and hence, the set of in-label sets created in Lines 1-7 are supersets of the corresponding in-label sets in the TOL on G'.

After adding u into $L_{in}(x)$, some existing labels related to x may become redundant. By Definition 1, neither the Level Constraint nor the Reachability Constraint would be affected for an existing label, i.e., the only possible violation is on the Path Constraint. In particular, u may be an out-label of a vertex y, such that y is an in-label of x or x is an out-label of y, either of which leads to a violation of the Path Constraint since (i) there is a path from y to x with u on it, and (ii) u has higher level than both y and x. To address this issue, Algorithm 2 enumerates all relevant x and y in Lines 8 and 9, and remove redundant labels in Lines 10-13.

After that, we repeat the above procedure (Line 14) with "in" and "out" exchanged to update the in-label sets of the vertices that can reach v, as well as the out-label sets of the vertices that v can reach. Summarizing the above discussion, Algorithms 1 and 2 result in a TOL on G'.

Finally, as Algorithm 3 precisely evaluates the size differences between consecutive levels for v, i.e., $\Delta_i, i = 1, ..., |V|$ (as shown in Section 5.1.2), it identifies a level k for v, such that the size of the resulting index is minimized. □

Complexity Analysis. To analyze our algorithm, we consider the complexity step by step. In Step 1, given a vertex v for insertion, Algorithm 3 inspects all the vertices $u \in L'_{in}(v) \cup L'_{out}(v)$, and derives Δ_k by computing all vertices $w \in I_{in}(u) \cup I_{out}(u)$, each with one set operation, i.e., Lines 8 and 10. Let β be the cost of one set operation, then the complexity of Step 1 is bounded by $O(|V|^2\beta)$. In Step 2.1, Algorithm 1 incurs $|C_{in}(v)| + |C_{out}(v)|$ number of set operation in Line 6, therefore, its cost is bounded by $C_2 = O((|C_{in}(v)| + |C_{out}(v)|)\beta)$. In Step 2.2, Algorithm 2 performs the set operations for each pair of vertices in the worse case, resulting a complexity of $O(|V|^2\beta)$. Note that, Lines 9-14 inspects the vertices in $I_{in}(u) \cap L_{in}(x')$ and $I_{out}(u) \cap L_{out}(y')$, which can be implemented in set operations as well. Hence, the complexity for our entire insertion algorithm is $O(|V|^2\beta)$.

5.2 Deletion Algorithm

Next, we discuss our algorithm handling deletion in G. Let $G' = (V', E')$ be the graph obtained by removing a vertex v from G. As with the case of insertion, we assume that G' is a DAG, and we aim to transform $\mathcal{L}$ into a TOL index $\mathcal{L}'$ on G', such that the level orders of $\mathcal{L}$ and $\mathcal{L}'$ sort the vertices in V' in the same order.

Algorithm 4 presents the pseudo-code of our deletion method. It first removes v from every label set that it appears (Lines 1-4). Then, it refines the label sets in $\mathcal{L}$ to convert it into $\mathcal{L}'$ (Lines 5-22). In particular, it first retrieves the set $B^+(v)$ of all vertices that v can reach, using a BFS from v that follows the outgoing edges of each vertex. Then, it inspects the vertices in $B^+(v)$ in ascending order of their topological ranks (see Section 2), and reconstructs the in-label set of each vertex. (Note that the out-label sets of those vertices are not affected by the deletion of v.)

For each vertex $u \in B^+(v)$, the algorithm first creates a candidate set $C_{in}(u)$ (Line 8). Then, for each of u's in-neighbors z such that $z \neq v$, the algorithm inserts the in-labels of z into $C_{in}(u)$ (Lines 9-10). It can be proved that $C_{in}(u)$ is a superset of the in-labels of u in $\mathcal{L}'$. To refine $C_{in}(u)$ into u's in-label set, the algorithm examines each vertex w in $C_{in}(u)$ in ascending order of $l(w)$ (Line 12). If $l(w) < l(u)$ and $L_{out}(w) \cap L_{in}(u) = \emptyset$, then we identify w as an in-label of u in $\mathcal{L}'$ (Line 14). Subsequently, we remove the labels that become redundant due to the insertion of w into $L_{in}(u)$ (Lines 15-17). In particular, for each vertex s having w as an out-label, if u is also in the out-label set of s, then we remove u from the out-label of s.

Once all vertices in $B^+(v)$ are processed, the algorithm derives the set $B^-(v)$ of vertices that can reach v, by applying a BFS from v that follows the incoming edges of each vertex. After that, it reconstructs the out-label set of each vertex in $B^-(v)$, in a way similar to the case of $B^+(v)$ (Lines 18-21). Finally, it returns the modified label sets for all vertices expect v, i.e., the label sets that form $\mathcal{L}'$.

Correctness Proof. Lemma 4 proves the correctness of our deletion algorithm.

LEMMA 4. *Given G and a vertex v to be deleted, the updated labeling $\mathcal{L}'$ produced by Algorithm 4 is a TOL on G'.*

PROOF. We first prove that the candidate set $C_{in}(u)$ generated in Lines 8-10 is a superset of $L'_{in}(u)$. Then, we show that redundant labels in $L'_{in}(u)$ are removed in Lines 12-19.

Algorithm 4: DELETE

```
input  : $G = (V, E)$, $\mathcal{L}$, and $v \in V$
output: $\mathcal{L}'$
1  for each $x \in I_{out}(v)$ do
2      remove $v$ from $L_{out}(x)$ ;
3  for each $y \in I_{in}(v)$ do
4      remove $v$ from $L_{in}(y)$ ;
5  identify the set $B^+(v)$ of vertices that $v$ can reach, using a BFS from
   $v$ that follows the outgoing edges of each vertex ;
6  for each $u \in B^+(v)$ in ascending order of $o(u)$ do
7      let $N_{in}(u)$ be the set of in-neighbors of $u$ ;
8      create a candidate set $C_{in}(u) = \emptyset$ ;
9      for each $z \in N_{in}(u)$ such that $z \neq v$ do
10         $C_{in}(u) = C_{in}(u) \cup L_{in}(z) \cup \{z\}$;
11     $L_{in}(u) = \emptyset$ ;
12     for each $w \in C_{in}(u)$ in ascending order of $l(w)$ do
13         if $l(w) < l(u)$ and $L_{out}(w) \cap L_{in}(u) = \emptyset$ then
14             add $w$ into $L_{in}(u)$ ;
15             for each $s \in I_{out}(w)$ do
16                 if $u \in L_{out}(s)$ then
17                     remove $u$ from $L_{out}(s)$ ;
18 identify the set $B^-(v)$ of vertices that can reach $v$, using a BFS from
   $v$ that follows the incoming edges of each vertex ;
19 for each $u \in B^-(v)$ in descending order of $o(u)$ do
20     let $N_{out}(u)$ be the set of out-neighbors of $u$ ;
21     repeat Lines 8-17 with subscripts "in" and "out" exchanged ;
22 return the label sets of all vertices except $v$ ;
```

Consider each vertex x that is an updated in-label of u (i.e. $x \in L'_{in}(u)$). By the Reachability Constraint and the Path Constraint in Definition 1, we know that there is a path from x to u in G', and x is the highest-level vertex on the path. Let z be the in-label of u on the path. Then, either $x = z$ or x is an updated in-label of z (i.e., $x \in L'_{in}(z)$). To understand this, assume on the contrary that $x \neq z$ and $x \notin L'_{in}(z)$. Then, there must be a path P from x to z that contains a vertex with a higher level than both x and z. Consider a path P' that goes from x to z via P, and then from z to u. Observe that P' connects x to u and contains a vertex with a level higher than x. This contradicts the Path Constraint since $x \in L'_{in}(u)$. In summary, for any vertex x in $L'_{in}(u)$, it is either an in-neighbor of u or an in-label of an in-neighbor of u. Accordingly, Lines 8-10 constructs the candidate set $C_{in}(u)$ by combining all the in-neighbors of u as well as the in-labels of those in-neighbors. In addition, since we construct $L'_{in}(u)$ in the ascending order of $o(u)$, we ensure that the in-label sets of the in-neighbors of u are updated before the construction of $C_{in}(u)$, which guarantees that $C_{in}(u)$ is a superset of $L'_{in}(u)$.

Next, for each vertex w in $C_{in}(u)$, we add it into $L'_{in}(u)$ if $l(w) < l(u)$ and $L_{out}(w) \cap L'_{in}(u) = \emptyset$. This guarantees the Level and Path Constraints. Since the Reachability Constraint is already ensured in the construction of the candidate sets, we know that adding w to $L'_{in}(u)$ does not affect the TOL properties. However, some existing labels related to w and u may become redundant. By Definition 1, neither the Level Constraint nor the Reachability Constraint will be affected, and the only possible violation is on the Path Constraint. In particular, if w is also an out-label of a vertex s and if u is in the out-label of s, then u violates the Path Constraint, since there is a path from s to u that contains w, and w has higher level than u. On the other hand, the case that s is an in-label of u never occurs, since we add w into $L'_{in}(u)$ in ascending order of $l(w)$.

Summarizing the above discussion, for each vertex u, we first generate a superset of $L'_{in}(u)$, and then construct the $L'_{in}(u)$ by adding vertices from the superset. In addition, for each w that is added into $L'_{in}(u)$, we remove, on the fly, all redundant labels caused by w. As such, Algorithm 4 results in a TOL on G'. □

Complexity Analysis. The complexity of computing $B^+(v)$ and $B^-(v)$ is equivalent to the complexity of BFS in G, which is $O(|E|)$. Next, consider the computation cost for a vertex u in $B^+(v)$. In Lines 9-10, as we merge the updated in-label sets of the in-neighbors of u to form the candidate set for u, the complexity is bounded by $|V|\beta$, where β is the complexity of a set operation. Then, for each vertex x in the candidate set $C_{in}(u)$, it takes a constant number of set operations to add x to $L_{in}(u)$ and remove redundant labels. Thus, the total computation cost on u is bounded by $O(|V|\beta)$. As such, the complexity of the deletion algorithm is $O(|V|^2\beta)$.

6. ITERATIVE LABEL REDUCTION

The update algorithms in Section 5 retains the level order on the vertices in G, which, as mentioned, helps ensure that the performance of $\mathcal{L}$ does not significantly degrade after updates. As the initial level order l is retained during updates, however, it is essential that l is chosen carefully. Otherwise, if l renders $\mathcal{L}$ inefficient, then this inefficiency is likely to persist even after updates.

A straightforward solution to choose the initial l is to enumerate all possible level orders (i.e., all permutations of vertices in V), then construct a TOL index based on each order, and finally select the one that optimizes performance. However, this approach is far from practical due to the enormous number of possible level orders. As an alternative solution, one may select an initial level order l using some heuristic approach (e.g., using the existing TOL instantiations [8, 17, 30]), and then adjusts the level order l to improve $\mathcal{L}$. Interestingly, our update algorithms can be utilized for such adjustments of l.

Specifically, given $\mathcal{L}$, we can first remove a vertex v using Algorithm 4, and then insert v back using the insertion algorithm in Section 5.1. By the properties of the insertion algorithm, when v is re-inserted, its level $l(v)$ is set to a value that minimizes $|\mathcal{L}|$, i.e., the total size of the label sets in $\mathcal{L}$. Therefore, $|\mathcal{L}|$ is likely to decrease (and will never increase) after the *deletion and re-insertion* of v. By repeating this process for each vertex v, we can obtain an improved version of $\mathcal{L}$ with a (much) reduced total size. This decrease in $|\mathcal{L}|$ not only reduces space consumption, but also improves query efficiency (as $\mathcal{L}$ process queries by scanning label sets). In Section 8, we experimentally show that this label reduction approach can significantly enhance the performance of existing TOL instantiations [8, 17, 30].

7. CONSTRUCTION OF INITIAL $\mathcal{L}$

Although the label reduction algorithm improves the performance of $\mathcal{L}$, we observe from our experiments that it incurs substantial computation costs on large graphs. This motivates us to investigate more efficient methods for choosing a good initial level order l. In the following, we first present a new algorithm for construct an initial l (Section 7.1), and then discuss the construction of $\mathcal{L}$ given l (Section 7.2).

7.1 Deciding Vertex Level

Given a G and a vertex $v \in V$, let $S_{in}(v, G)$ be the set of vertices that can reach v in G, and $S_{out}(v, G)$ be the set of vertices that v can reach in G. Suppose that we set the level of v higher than the level of any vertex in $S_{in}(v, G) \cup S_{out}(v, G)$. Then, in the corresponding TOL index, we need to add v to the in-label sets of the vertices in $S_{in}(v, G)$, as well as the out-label set of the vertices in $S_{out}(v, G)$. In that case, v contributes $|S_{in}(v, G)| + |S_{out}(v, G)|$ labels in $\mathcal{L}$.

On the other hand, if we set the level of v to a lower level than all vertices in $S_{in}(v, G) \cup S_{out}(v, G)$, then we need to (i) add each vertex in $S_{in}(v, G)$ into $L_{in}(v)$, and (ii) add each vertex in $S_{out}(v, G)$ into $L_{out}(v)$. Furthermore, if v happens to be the only vertex that connects vertices in $S_{in}(v, G)$ to those in $S_{out}(v, G)$, then in the worst case, we have to add every vertex in $S_{out}(v, G)$ to the out-label set of every vertex in $S_{in}(v, G)$. In that scenario, v contributes $|S_{in}(v, G)| \cdot |S_{out}(v, G)| + |S_{in}(v, G)| + |S_{out}(v, G)|$ labels in $\mathcal{L}$.

We define $|S_{in}(v, G)|$ and $|S_{out}(v, G)|$ as the *in-score* and *out-score* of v, respectively. In addition, we define a *score function f* as follows:

$$f(v, G) = \frac{|S_{in}(v, G)| \cdot |S_{out}(v, G)| + |S_{in}(v, G)| + |S_{out}(v, G)|}{|S_{in}(v, G)| + |S_{out}(v, G)|}.$$

In the pathological case when $|S_{in}(v, G)| + |S_{out}(v, G)| = 0$, we define $f(v, G) = 0$. Intuitively, if $f(v, G)$ is large, then v should be given a higher order than the vertices in $S_{in}(v, G) \cup S_{out}(v, G)$, so as to avoid the worst-case space cost of $|S_{in}(v, G)| \cdot |S_{out}(v, G)| + |S_{in}(v, G)| + |S_{out}(v, G)|$.

Based on the above intuition, we can design an algorithm to derive a good level order l as follows. Given G, we first identify the vertex v_1 that maximizes $f(v_1, G)$, and then set $l(v_1) = 1$, i.e., we assign v_1 to the highest level. After that, we remove v_1 from G, and proceed to identify the vertex v_2 that maximizes $f(v_2, G)$ in the modified G, then set $l(v_2) = 2$. We repeat this process until all vertices are removed from G, i.e., until each vertex is given a level.

Although the above algorithm is intuitively, it is difficult to implement efficiently, as (i) the computation of $f(v, G)$ requires us to derive the in-score and out-score of v using BFS (or DFS) on G, and (ii) the in-score and out-score of a vertex need to be recomputed whenever another vertex is removed from G. To address this deficiency, we propose to *approximate* the in-scores and out-scores of the vertices in G.

Let $S^{\top}_{in}(v)$ and $S^{\top}_{out}(v)$ be the approximate in-score and out-score of a vertex v, respectively. For each vertex w in G with no in-neighbor (resp. out-neighbor), we set $S^{\top}_{in}(w) = 0$ (resp. $S^{\top}_{out}(w) = 0$) to zero; note that this is also the exact in-score (resp. out-score) of w. After that, based on those vertices w, we recursively compute the approximate in-score and out-score of each remaining vertex v as follows:

$$S^{\top}_{in}(v) \quad = \quad \begin{cases} \sum_{u \in N_{in}(v)} \left(S^{\top}_{in}(u) + 1\right), & \text{if } N_{in}(v) \neq \emptyset; \\ 0, & \text{otherwise.} \end{cases}$$

$$S^{\top}_{out}(v) \quad = \quad \begin{cases} \sum_{u \in N_{out}(v)} \left(S^{\top}_{out}(u) + 1\right), & \text{if } N_{out}(v) \neq \emptyset; \\ 0, & \text{otherwise.} \end{cases}$$

where $N_{in}(v)$ and $N_{out}(v)$ denote the sets of in-neighbors and out-neighbors of v, respectively. It can be verified that $S^{\top}_{in}(v)$ and $S^{\top}_{out}(v)$ are upperbounds of v's in-score and out-score, respectively. As an alternative solution, we also consider using a lowerbound of v's in-score (resp. out-score), denoted as $S^{\perp}_{in}(v)$ (resp. $S^{\perp}_{out}(v)$), for approximation. In particular, $S^{\perp}_{in}(v) = 0$ if v has no in-neighbor, and $S^{\perp}_{out}(v) = 0$ if v has no out-neighbor. For any other vertex, we have:

$$S^{\perp}_{in}(v) \quad = \quad \begin{cases} \sum_{u \in N_{in}(v)} \frac{S^{\perp}_{in}(u) + 1}{|N_{out}(u)|}, & \text{if } N_{in}(v) \neq \emptyset; \\ 0, & \text{otherwise.} \end{cases}$$

$$S^{\perp}_{out}(v) \quad = \quad \begin{cases} \sum_{u\in N_{out}(v)} \frac{S^{\perp}_{out}(u)+1}{|N_{out}(u)|}, & \text{if } N_{out}(v)\neq\emptyset;\\ 0, & \text{otherwise.}\end{cases}$$

Note that we can compute $S^{\top}_{in}(v)$ for all vertices v in G, using a linear scan of the vertices. Specifically, we inspect the vertices v in ascending order of their topological ranks $o(v)$ (see Section 2). As G is a DAG, the vertices with the smallest ranks must have no in-neighbors, and hence, we have $S^{\top}_{in}(u) = 0$ for any of those vertices. By the definition of topological ranks, when we inspect any other vertex w, the $S^{\top}_{in}$ values of w's in-neighbor must have been computed. Therefore, $S^{\top}_{in}(w)$ can be easily derived. In summary, we can compute $S^{\top}_{in}(v)$ for all vertices v in $O(|V|+|E|)$ time. The same algorithm can be easily extended to compute $S^{\perp}_{in}(v)$. Meanwhile, we can derive $S^{\top}_{out}(v)$ and $S^{\perp}_{out}(v)$ for all vertices v, using a linear scan of the vertices in G in descending order of their topological ranks.

7.2 Labeling Algorithm

As discussed in Section 4, a level order l uniquely decides a TOL index $\mathcal{L}$. In this section, we introduce an algorithm for constructing $\mathcal{L}$ given a level order l computed using the methods in Section 7.1. Similar approaches have been proposed in [2, 17] for specific instantiations of TOL (with specific level orderings), but the correctness analysis therein is not immediately applicable under the general TOL framework. Therefore, we present our algorithm and analysis for the sake of completeness.

Algorithm. Algorithm 5 presents our method (referred to as *Butterfly* for constructing $\mathcal{L}$ given a level order l on G. The algorithm first creates a copy of G (referred to as G_1), and then it runs in $|V|$ iterations. In the k-th iteration, it removes from G_k the vertex v with $l(v) = k$, and inserts v into the label sets of other vertices. In particular, we first obtain the set $B^+(v)$ of vertices in G_k that v can reach, using a BFS from v that follows the outgoing edges of each vertex. Then, for each vertex $w \in B^+(v)$, if $L_{out}(v) \cap L_{in}(w) = \emptyset$, then we add v into $L_{in}(w)$. After that, we perform a BFS on G_k from v following the incoming edges of each vertex, to identify the set $B^-(v)$ of vertices that can reach v in G_k. For each $u \in B^-(v)$, we insert v into $L_{out}(u)$ if $L_{out}(u) \cap L_{in}(v) = \emptyset$. At the end of the iteration, we remove v from G_k, and denote the resulting graph as G_{k+1}. After that, we proceed to the $(k+1)$-th iteration. Once all iterations are finished, Algorithm 5 returns the label sets constructed, which form a TOL index $\mathcal{L}$ on G.

Correctness and Complexity. We prove the correctness of Algorithm 5 by the following lemma.

LEMMA 5. *Given a DAG $G = (V, E)$ and a vertex level, Algorithm 5 outputs a TOL index of G.*

PROOF. Observe that, for any two vertices u and v, Algorithm 5 inserts u into the in-label (resp. out-label) set of v, only when $l(u) < l(v)$ and u can reach v (v can reach u). Therefore, the index constructed by Algorithm 5 never violates the Rechability or Level Constraint in Definition 1.

By contradiction, assume that the output of Algorithm 5 is not a TOL index. Then, there exist two vertices u and v such that (i) u is in the in-label or out-label of v, and (ii) u violates the Path Constraint. We first discuss the case when u is in the in-label of v, i.e., $u \to v$, and the same discussion can be extended to the alternative case.

Since u does not fulfill the Path Constraint, there exists a vertex w in the path from u to v, such that the level of w is higher than the levels of u and v. In case that there exist several such paths, we let w be the vertex with highest level in all those paths. Since the level of w is higher than the levels of u and v, w is removed before the removals of u and v. Then, when Algorithm 5 removes w, it must add w into the out-label set of u and the in-label set of v. Now observe that, when Algorithm 5 removes u, it adds u into the in-label set of v, only if $L_{out}(u) \cap L_{in}(v) = \emptyset$. However, since w has been inserted into both $L_{out}(u)$ and $L_{in}(v)$, $L_{out}(u) \cap L_{in}(v) \neq \emptyset$. This indicates that u should not be in $L_{in}(v)$, leading to a contradiction. □

Algorithm 5: BUTTERFLY

```
input  : $G$ and a level order $l$
output: a TOL index $\mathcal{L}$
let $G_1 = G$ ;
for $k = 1, \cdots, |V|$ do
    let $v$ be the vertex whose level is $k$ ;
    identify the set $B^+(v)$ of vertices that $v$ can reach in $G_k$, using a
        BFS from $v$ that follows the outgoing edges of each vertex ;
    identify the set $B^-(v)$ of vertices that can reach $v$ in $G_k$, using a
        BFS from $v$ that follows the incoming edges of each vertex ;
    for each vertex $u$ in $B^+(v)$ do
        if $L_{out}(v) \cap L_{in}(u) = \emptyset$ then
            add $v$ to $L_{in}(u)$ ;
    for each vertex $u$ in $B^-(v)$ do
        if $L_{out}(u) \cap L_{in}(v) = \emptyset$ then
            add $v$ to $L_{out}(u)$ ;
    remove $v$ and $G_k$ and denote the resulting graph as $G_{k+1}$ ;
return the label sets of all vertices in $G$ ;
```

We now discuss the complexity of Algorithm 5. Let $G_k = (V_k, E_k)$ be the input graph to the k-th iteration, and denote the vertex with highest vertex level in G_k by v. The complexity of forward and backward BFS is bounded by $|E_k|$, and total size of $B^-(v)$ and $B^+(v)$ is bounded by $|V_k|$. Besides, the size of in-label or out-label of any vertex at k-th iteration is at most k since only the vertices whose levels are higher than v are added the current labels. As such, the complexity of each operation, intersection or addition, can be bounded by $O(k)$. In sum, we bound the complexity at k-iteration as $O(|E_k| + k|V_k|)$. Note that, as we remove a vertex from G_k at the end of the k-th iteration, the graph for computing becomes smaller and smaller, that is, the complexity at the k-th iteration decreases significantly when k increases.

8. EXPERIMENTS

This section experimentally evaluates our solution against the state of the art. We implement our algorithms in C++, and we adopt the C++ implementations of all competitors provided by their authors. All of our experiments are conducted on a machine with an Intel Xeon 2.4GHz CPU and 48GB RAM, running Ubuntu 12.4. In each experiment, we measure the performance of each method for 5 times, and we report the average measurement. If a method requires more than 24 hours or more than 48GB RAM to preprocess a dataset D, we omit the method from the experiments on D.

Datasets and Queries. Table 3 shows the datasets used in our experiments. Among them, RG5, RG10, RG20, and RG40 are synthetic DAGs generated using the method in experiments in [8], varying the average degree of vertices from 5 to 40, setting the *topological level* to 8 (see [8] for details). The other 11 datasets are the largest DAGs that have been adopted in the literature. In particular, *uni-22m*, *uni-100m*, *uni-150m*, *wiki*, *citeseerx*, *go-uniprot*,

dataset	$\vert V\vert$	$\vert E\vert$	**avg. deg.**
RG5	1.0M	5.0M	5.00
RG10	1.0M	10.0M	10.00
RG20	1.0M	20.0M	20.00
RG40	1.0M	40.0M	40.00
uniprot22m (uni-22m)	1.6M	1.6M	1.00
uniprot100m (uni-100m)	16.1M	16.1M	1.00
uniprot150m (uni-150m)	25.0M	25.0M	1.00
wiki	2.3M	2.3M	1.01
Twitter	16.6M	18.4M	1.10
Yago2	16.1M	25.7M	1.59
Web-UK	20.4M	37.8M	1.85
citeseerx	6.3M	14.8M	2.36
GovWild	8.0M	23.7M	2.95
patent	3.7M	15.7M	4.27
go-uniprot	7.0M	34.8M	4.99

Table 3: Datasets (M = 10^6).

and *patents* are from [8, 17], while *GovWild*, *Yago2*, *Twitter*, and *Web-UK* are used in [25].

On each dataset G, we generate a set of 10^6 reachability queries. In particular, we first derive a topological order on the vertices in G. Then, for each query q, we randomly select two vertices from G, and we choose the vertex with lower (resp. higher) topological rank as the source (resp. terminal) vertex s (resp. t). This method of query generation ensures that none of the queries can be answered by trivially checking whether the terminal vertex has a lower topological rank than the source vertex[1].

For each experiment on updates on G, we randomly remove 10^4 vertices one by one from G, and measure the average deletion time of each method. After that, we insert the deleted vertices back into G, in reverse order of their removal. During this process, we evaluate the average insertion time of each algorithm.

Experiments on Dynamic Graphs. Our first set of experiments evaluates our solution against existing techniques for dynamic graphs. As mentioned in Section 3, there exist a few methods [4, 12, 13, 16, 22, 24, 32] for handling updates on reachability indices. Among them, [12,13,22] are shown to be restricted to small graphs with at most a few thousand vertices [20], while [24] only handles XML graphs. We test the remaining methods, and find that only Dagger [32] is able to run on more than one datasets in our experiments. Therefore, we choose Dagger as our competitor. In addition, we evaluate two versions of our solution, namely, *Butterfly-U (BU)* and *Butterfly-L (BL)*, such that BU adopts $S_{in}^{\top}$ and $S_{out}^{\top}$ as its score functions, and BL adopts $S_{in}^{\perp}$ and $S_{out}^{\perp}$ (see Section 7).

Figure 2 shows the average insertion time of BU, BL, and Dagger. Observe that BU is more efficient than Dagger in most cases. In particular, on Twitter, BU's insertion time is lower than that of Dagger by four orders of magnitude. Meanwhile, BL is evidently less efficient than BU, although it still outperforms Dagger on half of the datasets. (Note that we omit BL on RG40, as it incurs excessive memory consumption on the graph.) The performance gap between BU and BL indicates that the vertex ordering adopted by BU is superior to that by BL. Meanwhile, Dagger considerably outperforms BU and BL on uni-22m, uni-100m, and uni-150m, since (i) each of those three graphs is a tree, and (ii) Dagger is particularly efficient on trees [32].

Figure 3 illustrates the total query time (for processing 10^6 queries) of BU, BL, Dagger, as well as a simple baseline approach. In particular, given a reachability query q on a graph G, the baseline approach performs a BFS from the source vertex of q (following the outgoing edges of each vertex), as well as a BFS from the terminal vertex of q (following the incoming edges of each vertex). The two BFSs take turns to traverse the vertices in G, until a common vertex is visited by both BFSs (i.e., when a path from the source vertex to the terminal vertex is found). As shown in Figure 3, BU consistently outperforms Dagger and the BFS approach, and BL's query time is slightly worse than BU's in general. On the other hand, Dagger is only slightly better than the BFS approach on most datasets, and is more than 900 (resp. 700) times slower than the latter on Wiki (resp. Twitter). This shows that Dagger is *not* a favorable approach for handling updates on dynamic graphs, as it incurs significantly higher update overheads than the BFS approach without providing substantially better query performance. (Note that the BFS approach entails zero update costs as it does not maintain any index.)

Finally, Figure 4 shows the average deletion time of each method. BU and BL's performance is generally comparable to Dagger's, except on RG40 and wiki. The slightly inferior performance of BU on deletion is justified by its superior efficiency on insertions and queries.

Experiments on Static Graphs. Our second set of experiments compares BU and BL with three state-of-the-art methods for static graphs, namely, *TF-label (TF)* [8], *hierarchical labeling (HL)* [17], *distribution labeling (DL)* [17]. For completeness, we also include Dagger in the experiments. Figure 5 illustrates the space consumption of each method. Observe that BU and BL generally outperform both TF and DL. To explain, recall that BU, BL, TF, and DL are all instantiations of the TOL framework. As such, their performance are *solely* decided by the vertex orderings that they adopt. The vertex ordering in DL (resp. TF), however, simply ranks vertices based on their degrees (topological ranks). In contrast, both BU and BL rank vertices based on advanced score functions that take into account the characteristics of the input graphs. As such, the vertex orderings employed by BU and BL are superior to those by TF and DL, and hence, lead to smaller index sizes. In particular, on RG10, the space overhead of BU is 5 times smaller than that of DL. Meanwhile, the space cost of HL is always higher than that of DL, which is consistent with the experimental results in [17].

Figure 6 illustrates the preprocessing time of each method. The relative performance of BU, BL, TF, HL, and DL are similar to the case of Figure 5, since a smaller index size indicates fewer labels in the label sets, and hence, the construction cost for the label sets is generally smaller. We omit HL and DL on RG20 and RG40, since their memory consumptions on those graphs exceed 48GB. On RG10, the preprocessing costs of HL, DL, and TF are at least an order of magnitude higher than that of BU, which indicates that the former are more sensitive to the average degree of the input graph. Figure 7 plots each algorithm's total query time for 10^6 random queries. Again, BU and BL consistently outperform TF and DL, due to their improved vertex orderings. In particular, on RG10, the query time of BU is 4 (resp. 7) times less than that of TF (resp. DL). Meanwhile, the query cost of HL is comparable to that of DL.

Experiments on Label Reduction. Our last set of experiments evaluates the label reduction approach presented in Section 6. Specifically, we first use DL and TF to construct reachability in-

[1] We have also experimented with alternative query sets that disregard the topological ranks of vertices; the experimental results are qualitatively similar to those reported in this paper, and are included in our online technical report [1].

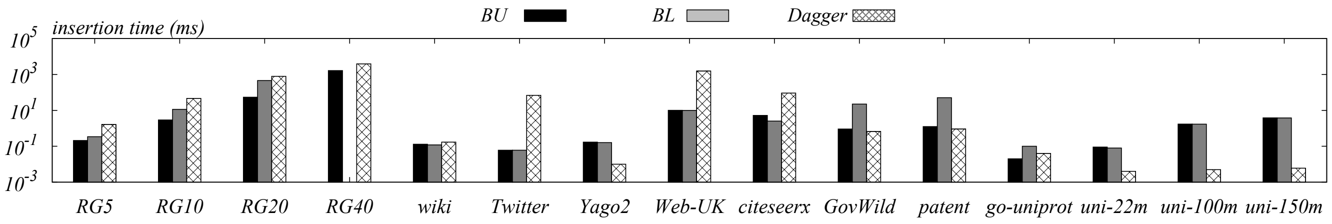

Figure 2: Average insertion time on dynamic graphs.

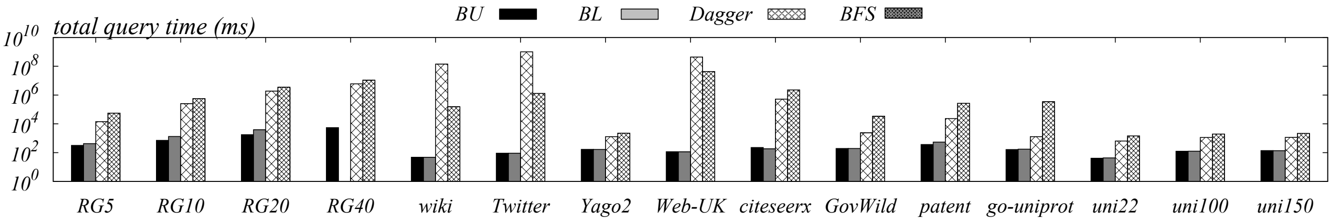

Figure 3: Total query time on dynamic graphs.

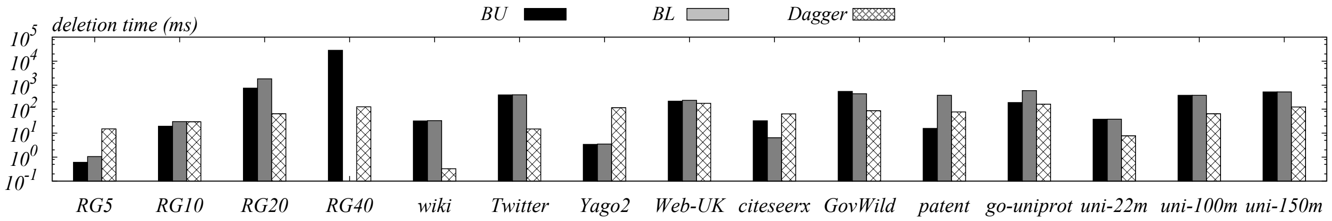

Figure 4: Average deletion time on dynamic graphs.

	DL			TF		
	$\Delta_{\mathcal{L}}$(MB)	$\Delta_{\mathcal{L}}/\lvert\mathcal{L}\rvert$	time (s)	$\Delta_{\mathcal{L}}$(MB)	$\Delta_{\mathcal{L}}/\lvert\mathcal{L}\rvert$	time (s)
RG5	98.91	51.94%	986.3	877.22	82.79%	2.9K
RG10	1.63K	81.61%	98.9K	—	—	—
uni-22m	0.00	0.00%	0.1	0.00	0.01%	0.3
uni-100m	0.71	0.96%	3.6	2.26	3.89%	5.5
uni-150m	3.14	2.46%	201.1	4.77	10.59%	324.2
wiki	0.00	0.00%	0.1	0.01	0.74%	0.3
Twitter	0.00	0.00%	3.7	0.02	8.41%	6.5
Yago2	0.00	0.00%	18.9	0.00	0.00%	18.8
Web-UK	38.69	21.80%	456.3	820.17	47.02%	12.7K
citeseerx	9.14	13.95%	773.2	192.71	96.23%	1.5K
GovWild	20.69	16.24%	844.1	2.40K	94.62%	59.3K
patent	255.09	48.92%	3.0K	3.69K	93.27%	186.5K
go-uniprot	53.24	26.91%	2.9K	104.61	66.38%	5.4K

Table 4: Performance of label reduction (K = 10^3).

dices $\mathcal{L}$ on each graph, and then apply our label reduction algorithm on $\mathcal{L}$ to obtain an improved index $\mathcal{L}^*$. Then, we measure the difference $\Delta_{\mathcal{L}}$ between the sizes of $\mathcal{L}^*$ and $\mathcal{L}$, and we divide $\Delta_{\mathcal{L}}$ by the size of $\mathcal{L}$ to derive the ratio of space reduction. Observe that the ratio of space reduction is up to 81.61% and 96.23% for DL and TF, respectively. This demonstrates the effectiveness of our label reduction approach. Nevertheless, the label reduction process incurs significant computation overheads on some of the graphs (e.g., patent). This indicates that one should not overly rely on the label reduction approach to improve the performance of a TOL index, but should adopt a good initial vertex order (e.g., the ones adopted by BU and BL). Note that no result is presented for TF on RG10, since the label reduction process takes excessive time on the graph. In addition, we omit RG20 and RG40 from the experiments, since both DL and TF incur prohibitive memory consumption on those two datasets.

9. CONCLUSIONS

This paper presents a novel study on reachability queries on *large dynamic graphs*. We propose general and efficient algorithms for processing vertex insertions and deletions on reachability indices, and we show that our algorithms can also be used to improve the performance of existing techniques for static graphs. In addition, we devise a new algorithm for constructing an efficient reachability index on an input graph from scratch. We evaluate our solution on a large set of real and synthetic graphs, and we demonstrate that our solution not only supports efficient updates on large dynamic graphs, but also provides even better query performance than the state-of-the-art techniques for *static* graphs. To our knowledge, we are the first in the literature to present a reachability index that can efficiently handle updates while offering superior query performance. For future work, we plan to investigate how our solutions can be extended to massive graphs that do not fit in main memory.

10. ACKNOWLEDGMENTS

This work was supported by the Nanyang Technological University under SUG Grant M4080094.020, by the Microsoft Research Asia under a gift grant, and by the Ministry of Education (Singapore) under AcRF Tier-2 Grant ARC19/14.

11. REFERENCES

[1] `https://sites.google.com/site/totalorderlabelling`.

[2] I. Abraham, D. Delling, A. V. Goldberg, and R. F. F. Werneck. Hierarchical hub labelings for shortest paths. In *ESA*, pages 24–35, 2012.

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

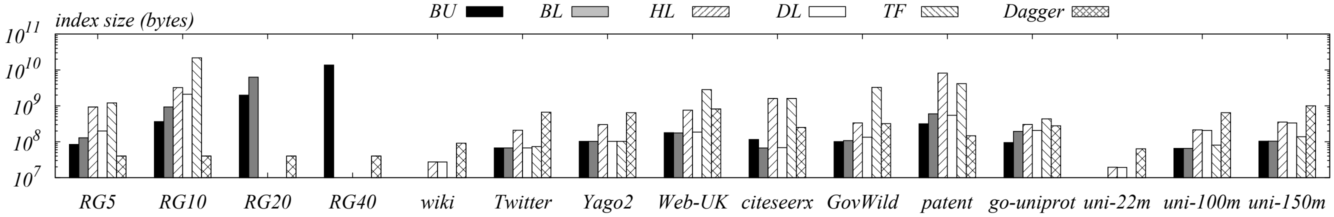

Figure 5: Index sizes on static graphs.

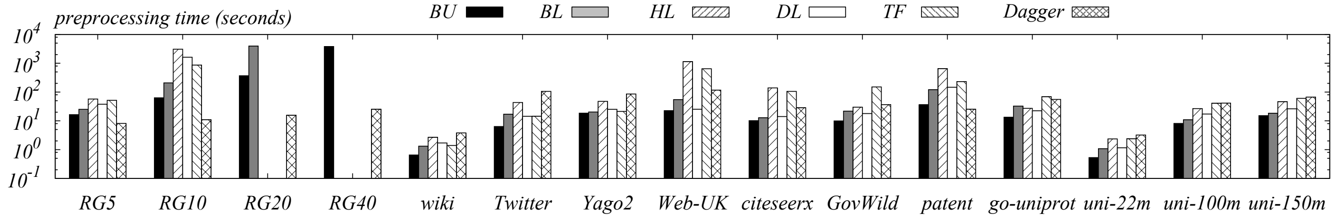

Figure 6: Preprocessing time on static graphs.

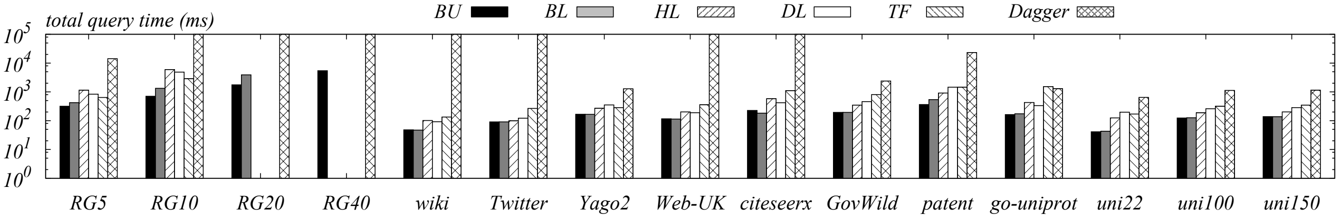

Figure 7: Total query time on static graphs.

[4] R. Bramandia, B. Choi, and W. K. Ng. Incremental maintenance of 2-hop labeling of large graphs. *IEEE Trans. Knowl. Data Eng.*, 22(5):682–698, 2010.

[5] J. Cai and C. K. Poon. Path-hop: efficiently indexing large graphs for reachability queries. In *CIKM*, pages 119–128, 2010.

[6] L. Chen, A. Gupta, and M. E. Kurul. Stack-based algorithms for pattern matching on dags. In *VLDB*, pages 493–504, 2005.

[7] Y. Chen and Y. Chen. An efficient algorithm for answering graph reachability queries. In *ICDE*, pages 893–902, 2008.

[8] 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 *SIGMOD*, pages 193–204, 2013.

[9] J. Cheng, Z. Shang, H. Cheng, H. Wang, and J. X. Yu. K-reach: Who is in your small world. *PVLDB*, 5(11):1292–1303, 2012.

[10] J. Cheng, J. X. Yu, X. Lin, H. Wang, and P. S. Yu. Fast computing reachability labelings for large graphs with high compression rate. In *EDBT*, pages 193–204, 2008.

[11] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. In *SODA*, pages 937–946, 2002.

[12] C. Demetrescu and G. F. Italiano. Fully dynamic all pairs shortest paths with real edge weights. *J. Comput. Syst. Sci.*, 72(5):813–837, 2006.

[13] M. R. Henzinger and V. King. Fully dynamic biconnectivity and transitive closure. In *FOCS*, pages 664–672, 1995.

[14] H. V. Jagadish. A compression technique to materialize transitive closure. *ACM Trans. Database Syst.*, 15(4):558–598, 1990.

[15] R. Jin, N. Ruan, S. Dey, and J. X. Yu. Scarab: scaling reachability computation on large graphs. In *SIGMOD*, pages 169–180, 2012.

[16] R. Jin, N. Ruan, Y. Xiang, and H. Wang. Path-tree: An efficient reachability indexing scheme for large directed graphs. *ACM Trans. Database Syst.*, 36(1):7, 2011.

[17] R. Jin and G. Wang. Simple, fast, and scalable reachability oracle. *PVLDB*, 6(14):1978–1989, 2013.

[18] R. Jin, Y. Xiang, N. Ruan, and D. Fuhry. 3-hop: a high-compression indexing scheme for reachability query. In *SIGMOD*, pages 813–826, 2009.

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

[20] I. Krommidas and C. D. Zaroliagis. An experimental study of algorithms for fully dynamic transitive closure. *ACM Journal of Experimental Algorithmics*, 12, 2008.

[21] L. Roditty. Decremental maintenance of strongly connected components. In *SODA*, pages 1143–1150, 2013.

[22] L. Roditty and U. Zwick. A fully dynamic reachability algorithm for directed graphs with an almost linear update time. In *STOC*, pages 184–191, 2004.

[23] R. Schenkel, A. Theobald, and G. Weikum. Hopi: An efficient connection index for complex xml document collections. In *EDBT*, pages 237–255, 2004.

[24] R. Schenkel, A. Theobald, and G. Weikum. Efficient creation and incremental maintenance of the hopi index for complex xml document collections. In *ICDE*, pages 360–371, 2005.

[25] S. Seufert, A. Anand, S. J. Bedathur, and G. Weikum. Ferrari: Flexible and efficient reachability range assignment for graph indexing. In *ICDE*, pages 1009–1020, 2013.

[26] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[27] S. Trißl and U. Leser. Fast and practical indexing and querying of very large graphs. In *SIGMOD*, pages 845–856, 2007.

[28] S. J. van Schaik and O. de Moor. A memory efficient reachability data structure through bit vector compression. In *SIGMOD*, pages 913–924, 2011.

[29] H. Wang, H. He, J. Yang, P. S. Yu, and J. X. Yu. Dual labeling: Answering graph reachability queries in constant time. In *ICDE*, page 75, 2006.

[30] Y. Yano, T. Akiba, Y. Iwata, and Y. Yoshida. Fast and scalable reachability queries on graphs by pruned labeling with landmarks and paths. In *CIKM*, pages 1601–1606, 2013.

[31] H. Yildirim, V. Chaoji, and M. J. Zaki. Grail: Scalable reachability index for large graphs. *PVLDB*, 3(1):276–284, 2010.

[32] H. Yildirim, V. Chaoji, and M. J. Zaki. Dagger: A scalable index for reachability queries in large dynamic graphs. *CoRR*, abs/1301.0977, 2013.