Fast and Scalable Reachability Queries on Graphs by Pruned Labeling with Landmarks and Paths

Yosuke Yano[‡] Takuya Akiba[‡] Yoichi Iwata[‡] Yuichi Yoshida[†§]
‡The University of Tokyo, †National Institute of Informatics, §Preferred Infrastructure, Inc.
{yyano, t.akiba, y.iwata}@is.s.u-tokyo.ac.jp, yyoshida@nii.ac.jp

ABSTRACT

Answering reachability queries on directed graphs is ubiquitous in many applications involved with graph-shaped data as one of the most fundamental and important operations. However, it is still highly challenging to efficiently process them on large-scale graphs. Transitive-closure-based methods consume prohibitively large index space, and online-search-based methods answer queries too slowly. Labeling-based methods attain both small index size and query time, but previous indexing algorithms are not scalable at all for processing large graphs of the day. In this paper, we propose new labeling-based methods for reachability queries, referred to as *pruned landmark labeling* and *pruned path labeling*. They follow the frameworks of *2-hop cover* and *3-hop cover*, but their indexing algorithms are based on the recent notion of *pruned labeling* and improve the indexing time by several orders of magnitude, resulting in applicability to large graphs with tens of millions of vertices and edges. Our experimental results show that they attain remarkable trade-offs between fast query time, small index size and scalability, which previous methods have never been able to achieve. Furthermore, we also discuss the ingredients of the efficiency of our methods by a novel theoretical analysis based on the graph minor theory.

Categories and Subject Descriptors

E.1 [**Data**]: Data Structures—*Graphs and networks*

Keywords

Graphs; reachability; query processing

1. INTRODUCTION

Answering *reachability queries*, determining whether there is a directed path from a vertex s to a vertex t on a given directed graph $G = (V, E)$, is ubiquitous as one of the most basic and important operations on graphs. For example, in query engines such as SPARQL and XQuery, it is one of the fundamental building blocks for answering user queries [11, 2]. In computational biology, it is employed for representing and analyzing molecular and cellular functions [17]. In program analysis, it enables precise interprocedural dataflow analysis [13, 12].

Due to its importance and recent emergence of large graph-shaped data, many indexing schemes have been proposed in the recent database community [18,8,3,19,20,5,15,4,7]. Nevertheless, efficiently processing reachability queries on large graphs is still a highly challenging task since current state-of-the-art methods suffer from drawback of either scalability or large query time.

One of the most classical approaches is to compress *transitive closure* [16, 18]. However, even with compression, space complexity is still essentially quadratic, and thus this approach is not promising with regard to scalability. In contrast, methods that conduct a graph search guided by precomputed indices for answering each query achieve better scalability due to small indexing time and index size [3, 19, 20]. However, their query time is several orders of magnitude slower than other methods, which is critical for certain applications such as SPARQL engines and XQuery engines, as sometimes answers to thousands or millions of reachability queries are necessary to process one user query [19].

Methods based on *labeling* to vertices have also been studied for a long time [5, 15, 4, 7]. They precompute a *label* for each vertex so that a reachability query can be answered from the labels of two endpoints. This approach is promising since, after obtaining small labels, they attain both fast query time and small index size. However, computing such labels has been challenging and highly expensive, thus limiting the scalability of this approach.

1.1 Contributions

To address these issues, in this paper, we propose new labeling-based methods for reachability queries, referred to as *pruned landmark labeling* and *pruned path labeling*. Since they are labeling-based methods, they achieve fast query time and small index size, but their indexing algorithms are significantly more efficient than previous algorithms. As a result, they overcome the drawback of the scalability of labeling-based methods and attain remarkable trade-offs between query time, index size and scalability, which previous methods have never been able to achieve.

As the labeling framework (i.e., index data structure and query algorithm), our pruned landmark labeling follows 2-hop cover [5], which stores sets of vertices $L_{\mathrm{OUT}}(v)$ and $L_{\mathrm{IN}}(v)$ as the label for each vertex v so that a reachability query (u, v) can be answered by testing whether $L_{\mathrm{OUT}}(u)$

CIKM'13, Oct. 27–Nov. 1, 2013, San Francisco, CA, USA.

ACM 978-1-4503-2263-8/13/10 ...$15.00.
http://dx.doi.org/10.1145/2505515.2505724.

and $L_{\text{IN}}(v)$ have non-empty intersection (see Section 3). By contrast, our pruned path labeling follows 3-hop cover [7], which is a generalization of 2-hop cover and stores intervals of paths as labels (see Section 4.1). For both frameworks, while it is conjectured that small labels exist for real-world networks [5], previous indexing algorithms are too slow and cannot be applied to large graphs of the day.

Our key contributions are the new indexing algorithms of pruned landmark labeling and pruned path labeling. They are based on a recent shortest-path querying method [1]. To compute labels, in contrast to previous methods that solve indirect optimization problems with approximate algorithms, our algorithms conduct pruned graph searches and directly add label entries to labels of visited vertices. Note that our application of the shortest-path querying method [1] involves several non-trivial challenges. For example, while the shortest-path querying method is tailored to networks such as social networks and web graphs, we design the proposed methods for real-world directed acyclic graphs with different structures. Moreover, the indexing algorithm of pruned path labeling is an essentially new algorithm, which computes label entries from all the vertices in a path by just one pruned search.

Our experimental results in Section 6 show that *(1)* they have good scalability and can be applied to graphs with tens of millions of vertices and edges, *(2)* their query time is the fastest among the methods and two orders of magnitude faster than online-search-based methods, and *(3)* their index size is an order of magnitude smaller than transitive-closure-based methods.

Furthermore, we also theoretically discuss why our methods are efficient on real-world graphs in Section 5. In addition to the fact that both proposed methods can exploit tree-like structures of small treewidth, we present a novel analysis based on the graph minor theory proving that our pruned path labeling method can exploit a minor-closed property, which leads to efficiency in various kinds of structures.

Independence from arXiv:1305.0502 [6]. We have recently noticed that an approach similar to our pruned landmark labeling method is independently proposed by Jin and Wang [6].

2. PRELIMINARIES

Notations: Let $G = (V, E)$ be a directed graph. We denote the number of vertices $|V|$ and the number of edges $|E|$ by n and m, respectively. For two vertices $s, t \in V$, we define $\text{reach}(s, t)$ as **true** if there is a path from s to t and **false** otherwise. A reachability query (s, t) asks whether $\text{reach}(s, t)$ is **true** or not. We denote the children of v by $\text{children}(v)$ and the parents of v by $\text{parents}(v)$. We denote the indegree and outdegree of v by $d_{\text{IN}}(v)$ and $d_{\text{OUT}}(v)$.

Strongly Connected Components: We can safely assume that the input graph is always a directed acyclic graph (DAG). Note that all vertices in a strongly connected component (SCC) of G is equivalent in terms of reachability since they are reachable each other. Thus, G can be converted into a DAG by SCCs, preserving the information of reachability among vertices.

Problem Definition: The problem is to precompute some index for a given DAG $G = (V, E)$, and to answer reachability queries quickly by using the index, and the DAG if necessary. We assume that all reachability queries are given after indexing. That is, we cannot create an index specialized for a particular set of queries.

3. PRUNED LANDMARK LABELING

Given a DAG $G = (V, E)$, we create two types of labels $L_{\text{OUT}}(v), L_{\text{IN}}(v) \subseteq V$ for each vertex v. Upon a query (s, t), we return $\textsc{Query}(s, t, L_{\text{OUT}}, L_{\text{IN}})$ defined as follows. $\textsc{Query}(s, t, L_{\text{OUT}}, L_{\text{IN}})$ is **true** if $L_{\text{OUT}}(s)$ and $L_{\text{IN}}(t)$ have a non-empty intersection. Otherwise, $\textsc{Query}(s, t, L_{\text{OUT}}, L_{\text{IN}})$ is **false**. We will construct L_{OUT} and L_{IN} so that, for every $s, t \in V$, $\textsc{Query}(s, t, L_{\text{OUT}}, L_{\text{IN}}) = \text{reach}(s, t)$ holds. Naive calculation of $\textsc{Query}(s, t, L_{\text{OUT}}, L_{\text{IN}})$ needs $O(l_s \cdot l_t)$ time for $l_s = |L_{\text{OUT}}(s)|$ and $l_t = |L_{\text{IN}}(t)|$. However, if $L_{\text{OUT}}(s)$ and $L_{\text{IN}}(t)$ are sorted, it can be done in $O(l_s + l_t)$ time by scanning both labels from their heads to tails simultaneously.

For simplicity, we first describe a naive algorithm to construct L_{OUT} and L_{IN} without pruning, and then proceed to an algorithm with pruning.

3.1 Naive Algorithm

Let $V = \{v_1, \ldots, v_n\}$ be the vertex set. We incrementally construct labels by processing $v_1, \ldots, v_n$ in this order. Though the choice of this ordering has a large impact on the performance, we just let it arbitrary at this moment.

To make exposition easier, we define L^k_{OUT} and L^k_{IN} as L_{OUT} and L_{IN}, respectively, right after processing v_k. In the beginning, we start with $L^0_{\text{OUT}}(v) = L^0_{\text{IN}}(v) = \emptyset$ for every $v \in V$. Suppose that we have constructed L^{k-1}_{OUT} and L^{k-1}_{IN}. Then, we construct L^k_{OUT} and L^k_{IN} by processing v_k. First we describe how to construct L^k_{IN}. We conduct a BFS from v_k, and add v_k to the labels of vertices that are visited during the BFS. Specifically, we set $L^k_{\text{IN}}(v)$ to $L^{k-1}_{\text{IN}}(v) \cup \{v_k\}$ if v is visited during the BFS, otherwise set $L^k_{\text{IN}}(v)$ to $L^{k-1}_{\text{IN}}(v)$. Similarly, to construct L^k_{OUT}, we conduct a *reversed* BFS from v_k, for which we traverse edges backwards. That is, we set $L^k_{\text{OUT}}(v)$ to $L^{k-1}_{\text{OUT}}(v) \cup \{v_k\}$ if v is visited during the reversed BFS, otherwise set $L^k_{\text{OUT}}(v)$ to $L^{k-1}_{\text{OUT}}(v)$.

We use L^n_{OUT} and L^n_{IN} to answer reachability queries. Obviously, they satisfy that $\textsc{Query}(s, t, L^n_{\text{OUT}}, L^n_{\text{IN}}) = \text{reach}(s, t)$ since every vertex has all information about which vertices it can reach and it can be reached from. We also note that $\textsc{Query}(s, t, L^k_{\text{OUT}}, L^k_{\text{IN}}) =$ **true** if and only if there is a path from s to t passing through one of $v_1, \ldots, v_k$. The proof is similar to [1], and we omit.

3.2 Pruned Landmark Labeling

The naive algorithm costs too much time and space since we conduct BFSs $2n$ times, which results in $O(nm)$ time. In pruned landmark labeling, we stop BFSs by pruning vertices whose reachability can be answered correctly from the labels constructed so far.

Suppose that we are visiting a vertex v during the BFS from a vertex v_k, and that v can be shown to be reachable from v_k by existing labels, that is, $\textsc{Query}(v_k, v, L^{k-1}_{\text{OUT}}, L^{k-1}_{\text{IN}})$ is **true**. Then, we prune the vertex v and do not search descendants of v. Similarly, when we visit v during the reversed BFS from v_k and $\textsc{Query}(v, v_k, L^{k-1}_{\text{OUT}}, L^{k-1}_{\text{IN}})$ is **true**, then we prune the vertex v.

Though we have pruned vertices, L^k_{OUT} and L^k_{IN} still satisfy the property that $\textsc{Query}(s, t, L^k_{\text{OUT}}, L^k_{\text{IN}}) =$ **true** if and only if there is a path from s to t passing through one of $v_1, \ldots, v_k$. The correctness of the pruning is proved in [1].

The performance of pruned landmark labeling for shortest path queries depends heavily on the vertex ordering, as Akiba *et al.* have shown by experimental comparisons [1]. Thus, to prune more vertices, we want to find a vertex ordering $v_1, \ldots, v_n$ such that most reachable pairs (s, t) satisfy that s can reach t via an early vertex in the ordering. We adopt the strategy InOut where we sort vertices by $(d_{\text{IN}}(v) + 1) \times (d_{\text{OUT}}(v) + 1)$ in decreasing order, since it performed well in preliminary experiments.

4. PRUNED PATH LABELING

In this section, we first propose the pruned path labeling method, which is based on pruned landmark labeling in Section 3. Then we discuss heuristics to select such paths in Section 4.2.

4.1 Overview

The idea of pruned path labeling is iteratively selecting paths and conducting BFSs from these paths. The main difference from pruned landmark labeling is that we use paths instead of vertices to start BFSs with. Then, we store which vertices can reach these paths or can be reached from these paths. If a query (s, t) is given, we find a path we have selected with two vertices u, v such that there is a path of the form $s - u - v - t$. In this sense, our method can be seen as a 3-hop cover [7]. The detail is given in the following.

For a given DAG $G = (V, E)$, we take l paths $P_1, P_2, \ldots, P_l$ such that $\bigcup_{k=1}^{l} P_k = V$. Let $\{v_{k,1}, v_{k,2}, \ldots, v_{k,p_k}\}$ denote the sequence of vertices that forms the path P_k, where $p_k = |P_k|$. We construct two types of labels $L_{\text{OUT}}(v), L_{\text{IN}}(v) \subseteq \mathbb{N} \times \mathbb{N}$. It is supposed that, if $(i, j) \in L_{\text{OUT}}(v)$ for some vertex $v \in V$, then v can reach $v_{i,j}$. Similarly, it is supposed that, if $(i, j) \in L_{\text{IN}}(v)$ for a vertex $v \in V$, then v can be reached from $v_{i,j}$.

We note that, for any vertex $v \in V$ and i, we only have to store at most one pair (i, j) in $L_{\text{OUT}}(v)$ to answer reachability. To see this, suppose that v can reach $v_{i,j}$ for some i and j. Then, v can reach every $v_{i,j'}$ for $j \leq j' \leq p_i$ since $v_{i,j}$ can reach $v_{i,j'}$ through the path P_i. Thus, we can choose an integer $j_{\min}$ such that v can reach $v_{i,j}$ if and only if $j_{\min} \leq j \leq p_i$. Therefore, we only have to store the pair $(i, j_{\min})$ in $L_{\text{OUT}}(v)$ to answer the reachability of vertices in P_i from v. Conversely, for each $v \in V$ and i, we only have to store one pair (i, j) in $L_{\text{IN}}(v)$.

Upon a query (s, t), we return $\textsc{Query}(s, t, L_{\text{OUT}}, L_{\text{IN}})$ defined as follows.

$$\textsc{Query}(s, t, L_{\text{OUT}}, L_{\text{IN}}) = \begin{cases} \textbf{true} & \text{if } \exists i, j, j' \in \mathbb{N} \text{ s.t. } j \leq j', \\ & \quad (i, j) \in L_{\text{OUT}}(s), \\ & \quad (i, j') \in L_{\text{IN}}(t), \\ \textbf{false} & \text{otherwise.} \end{cases}$$

In words, $\textsc{Query}(s, t, L_{\text{OUT}}, L_{\text{IN}})$ is **true** if and only if there are a path P_i and two integers j, j' with $j \leq j'$ such that s can reach $v_{i,j}$ and t can be reached from $v_{i,j'}$. We emphasize again that $v_{i,j}$ can reach $v_{i,j'}$ through P_i. We can compute $\textsc{Query}(s, t, L_{\text{OUT}}, L_{\text{IN}})$ in $O(|L_{\text{OUT}}(s)| + |L_{\text{IN}}(t)|)$ time by a merge-sort-like algorithm if $L_{\text{OUT}}(s)$ and $L_{\text{IN}}(t)$ are sorted by path index.

4.1.1 Label Construction

Now we describe how to construct labels L_{OUT} and L_{IN}. Again, we start with a naive algorithm. We basically conduct BFSs from paths $P_1, P_2, \ldots, P_l$ in this order. Since labels L_{OUT} and L_{IN} grow gradually during the algorithm, we define L^i_{OUT} and L^i_{IN} as L_{OUT} and L_{IN} obtained right after processing the i-th path P_i. In particular, we define $L^0_{\text{OUT}}(v)$ and $L^0_{\text{IN}}(v)$ as $\emptyset$ for all $v \in V$, and the pair $L^l_{\text{OUT}}(v)$ and $L^l_{\text{IN}}(v)$ is the label finally output by the algorithm. Suppose that we have already constructed L^{k-1}_{OUT} and L^{k-1}_{IN}. Then, we construct L^k_{OUT} and L^k_{IN} as follows.

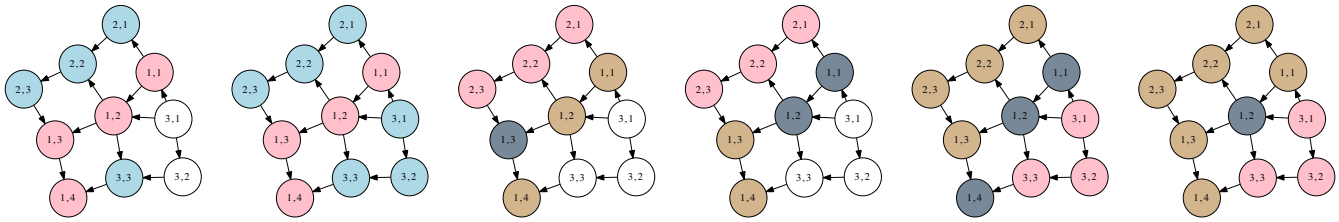
First, we conduct BFSs from vertices in P_k in descending order, that is, from v_{k,p_k} to $v_{k,1}$. In the BFS from the vertex $v_{k,j}$, we update L^k_{IN} to $L^{k-1}_{\text{IN}}(v) \cup \{(k, j)\}$ if v is visited. Otherwise, we set L^k_{IN} to $L^{k-1}_{\text{IN}}(v)$. When performing a BFS from $v_{k,j}$, we do not have to visit vertices that are already visited in previous BFSs since they already have a pair (k, j') for some $j' \geq j$.

After BFSs to construct L^k_{IN} are finished, we conduct reversed BFSs by traversing edges backwards from vertices in P_k in ascending order, that is, from $v_{k,1}$ to v_{k,p_k}. In the reversed BFS from the vertex $v_{k,j}$, we update L^k_{OUT} to $L^{k-1}_{\text{OUT}}(v) \cup \{(k, j)\}$ if v is visited, otherwise we set L^k_{OUT} to $L^{k-1}_{\text{OUT}}(v)$. As in the previous case, we do not have to visit vertices that are previously visited by reversed BFSs.

Now we improve the naive algorithm by introducing pruning. The idea is the same as pruned landmark labeling. Suppose that we are processing a vertex v in the BFS from a vertex $v_{k,j}$ for some k and j. Then, we issue a query $\textsc{Query}(v_{k,j}, v, L^{k-1}_{\text{OUT}}, L^{k-1}_{\text{IN}})$. If the answer is **true**, we prune v, that is, we stop the BFS at v. When we are processing a vertex v in the reversed BFS from a vertex $v_{k,j}$ for some

Algorithm 1 Conduct pruned BFSs from P_k

```
1: procedure PrunedBFS(G, P_k, L^{k-1}_OUT, L^{k-1}_IN)
2:     p ← the number of vertices in P_k
3:     L^k_OUT[v], L^k_IN[v] ← L^{k-1}_OUT[v], L^{k-1}_IN[v] for all v ∈ V
4:     Q ← an empty queue
5:     U ← ∅
6:     for i ← p ... 1 do
7:         s ← P_k[i]
8:         Enqueue s onto Q
9:         while Q is not empty do
10:            Dequeue v from Q
11:            U ← U ∪ {v}
12:            if Query(s, v, L^{k-1}_OUT, L^{k-1}_IN) is false then
13:                L^k_IN[v] ← L^k_IN[v] ∪ {(k, i)}
14:                for all u ∈ children(v) do
15:                    if u ∉ U then
16:                        Enqueue u onto Q
17:    U ← ∅
18:    for i ← 1 ... p do
19:        s ← P_k[i]
20:        Enqueue s onto Q
21:        while Q is not empty do
22:            Dequeue v from Q
23:            U ← U ∪ {v}
24:            if Query(s, v, L^{k-1}_OUT, L^{k-1}_IN) is false then
25:                L^k_OUT[v] ← L^k_OUT[v] ∪ {(k, i)}
26:                for all u ∈ parents(v) do
27:                    if u ∉ U then
28:                        Enqueue u onto Q
29:    return (L^k_OUT, L^k_IN)
```



(a) a BFS from P_1 (b) a reversed BFS from P_1 (c) a BFS from P_2 (d) a reversed BFS from P_2 (e) a BFS from P_3 (f) a reversed BFS from P_3

Figure 1: An example of pruned path labeling. Vertex color indicates its status: Red is a start point of BFSs, blue is a vertex being searched, gray is a pruned vertex, and brown is a vertex already used as a start point.

k and j, we issue a query $\textsc{Query}(v, v_{k,j}, L_{\text{OUT}}^{k-1}, L_{\text{IN}}^{k-1})$ instead. If the answer is **true**, we prune v. A pseudocode for constructing L_{OUT} and L_{IN} is shown in Algorithm 1.

Figure 1 shows an example of pruned path labeling. Three paths $P_1 = \{v_{1,1}, v_{1,2}, v_{1,3}, v_{1,4}\}$, $P_2 = \{v_{2,1}, v_{2,2}, v_{2,3}\}$, and $P_3 = \{v_{3,1}, v_{3,2}, v_{3,3}\}$ are selected. Then, BFSs are conducted from $v_{1,4}$, $v_{1,3}$, $v_{1,2}$, and $v_{1,1}$ in this order (Figure 1a) to obtain L_{IN}^1. We add a pair $(1, 2)$ to $L_{\text{IN}}^1(v_{2,2})$, $L_{\text{IN}}^1(v_{2,3})$, and $L_{\text{IN}}^1(v_{3,3})$. Also, we add $(1, 1)$ to $L_{\text{IN}}^1(v_{2,1})$. No pruning occurs during BFSs from P_1. Then similarly, reversed BFSs are conducted from $v_{1,1}$, $v_{1,2}$, $v_{1,3}$, and $v_{1,4}$ in this order (Figure 1b) to obtain L_{OUT}^1. For example, $L_{\text{OUT}}(v_{2,3})$ obtains a pair $(1, 3)$. Next, we conduct BFSs from vertices in P_2 in an appropriate order. When $v_{1,3}$ is visited during the BFS from $v_{2,3}$, we issue the query $\textsc{Query}(v_{2,3}, v_{1,3}, L_{\text{OUT}}^1, L_{\text{IN}}^1)$. The query is **true** since $(1, 3) \in L_{\text{OUT}}^1(v_{2,3})$ and $(1, 3) \in L_{\text{IN}}^1(v_{1,3})$. Therefore, $v_{1,3}$ is pruned and we no longer continue the search from $v_{1,3}$ (Figure 1c). We continue this process until we finish performing BFSs from all the paths (Figure 1d,1e,1f).

A potential drawback of adopting paths instead of vertices is that it may increase the index size. This is because that each element in a label is a pair of integers (a path index and an index of a vertex in the path) instead of one integer (a vertex number) as opposed to pruned landmark labeling. Therefore, we do not have any benefit if we cannot find long paths. Also, it is practically difficult to cover all the vertices by long paths. To address these issues, we combine the two methods. That is, for some constant $a \geq 0$, we perform pruned path labeling from a paths and then perform pruned landmark labeling from remaining vertices. Furthermore, we stop taking paths if the length of the path is shorter than b. From preliminary experiment, we decided to choose $a = 50$ and $b = 10$ in experiments in Section 6.

4.2 Path Selection

As we already mentioned in Section 3, vertex ordering strategies largely influence the performance of pruned landmark labeling. Correspondingly, effectiveness of pruning should depend on how to select paths in pruned path labeling. We empirically compared a few path selection strategies and found that the strategy DPInOut performed the best among them.

In DPInOut, we first assign a value to each vertex. The value assigned to a vertex v is $(d_{\text{IN}}(v)+1) \times (d_{\text{OUT}}(v)+1)$ if v is not selected as a part of a path before, and 0 otherwise. Then, we select the path in which the sum of the value of vertices is maximized, by dynamic programming on the DAG. After selecting 50 paths, we order remaining vertices by InOut. The idea behind DPInOut is to select paths that contain important vertices as many as possible.

5. THEORETICAL ANALYSIS

In this section, we give theoretical evidence that our methods perform well on real-world networks. Due to the space limitation, we only show two main theorems and omit the proofs. See [14] for the definitions of terms.

Akiba *et al.* [1] showed that their method works efficiently on bounded-treewidth graphs. Since pruned landmark labeling for reachability is similar to their method, we can use the same proof and we obtain the following.

Theorem 1. *Let G be a digraph whose underlying graph has bounded treewidth. Then, there is a strategy of selecting vertices for which pruned landmark labeling on G outputs a label of size $O(\log n)$ for each vertex. Furthermore, we can find the strategy in $O(n + m)$ time. (Constants depending on treewidth are hidden in the $O(\cdot)$ notations.)*

The theorem implies that index size is $O(n \log n)$ and query time is $O(\log n)$. We note that a complex network is known to have a *core* and *fringes* attached to it [9]. Since fringes are supposed to have small treewidth, this theorem implies that pruned landmark labeling performs well on complex networks.

Furthermore, we show that pruned path labeling can efficiently process graphs satisfying a minor-closed property.

Theorem 2. *Let P be a minor-closed property and G be a digraph whose underlying graph satisfies P. Then, there is a strategy of choosing paths for which pruned path labeling on G outputs a label of size $O(\log n)$ for each vertex. (Constants depending on P are hidden in the $O(\cdot)$ notations.)*

Again, the theorem implies that index size is $O(n \log n)$ and query time is $O(\log n)$. We note that examples of minor-closed properties include having bounded treewidth, planarity and bounded genus. Thus, pruned path labeling is not only practically but also theoretically stronger than pruned landmark labeling.

6. EXPERIMENTS

We show experimental results in this section. We compared our two proposed methods with state-of-the-art existing methods on both real and synthetic graphs. These methods are evaluated in terms of query time, index size, and indexing time. As query time, we report the average time over one million random queries.

Table 1: Real-world datasets

Dataset	$\|V\|_{SCC}$	$\|E\|_{SCC}$
ff/successors	1,858,504	2,009,541
citeseerx	6,540,399	15,011,259
cit-patents	3,774,768	16,518,948
go-uniprot	6,967,956	34,770,235
uniprot22m	1,595,444	1,595,442
uniprot100m	16,087,295	16,087,293
uniprot150m	25,037,600	25,037,598

Table 2: Average query time (μs)

Dataset	PLL	PPL	GRAIL	IL	PWAH
ff/successors	0.085	0.133	0.279	0.154	0.202
citeseerx	0.124	0.164	27.946	0.103	0.214
cit-patents	0.253	0.296	11.591	0.292	15.451
go-uniprot	0.156	0.194	0.520	0.233	0.521
uniprot22m	0.083	0.122	0.403	0.173	0.243
uniprot100m	0.133	0.197	0.743	0.292	0.361
uniprot150m	0.153	0.223	0.776	0.248	0.351

Table 3: Index size (MB)

Dataset	PLL	PPL	GRAIL	IL	PWAH
ff/successors	122.3	91.6	29.7	40.0	34.1
citeseerx	122.0	126.7	104.6	441.3	156.0
cit-patents	664.6	691.2	60.4	22444.5	5593.1
go-uniprot	263.1	273.5	111.5	792.7	255.9
uniprot22m	19.4	19.4	25.5	19.6	19.5
uniprot100m	206.8	206.8	257.4	223.0	218.8
uniprot150m	334.0	334.0	400.6	373.8	366.2

Table 4: Indexing time (sec)

Dataset	PLL	PPL	GRAIL	IL	PWAH
ff/successors	10.46	8.19	1.08	7.84	5.02
citeseerx	23.13	45.42	7.65	6.70	16.03
cit-patents	192.05	239.95	8.24	397.04	847.83
go-uniprot	26.60	29.74	5.78	18.33	31.10
uniprot22m	2.82	3.02	0.96	0.96	1.23
uniprot100m	30.80	32.99	12.39	10.64	14.39
uniprot150m	49.48	53.56	20.52	17.21	24.16

6.1 Experimental Setup

We conducted all the experiments on Linux server with Intel Xeon X5675 3.07GHz and 288GB memory. We only used one core on all the experiments. Pruned landmark labeling (PLL) and pruned path labeling (PPL) are compared with three state-of-the-art existing methods, GRAIL [19], interval list (IL) [10] and PWAH [18]. GRAIL is a graph traversal method exploiting labels created by random DFSs, and one of the most memory efficient methods for reachability queries. IL and PWAH are methods that construct compressed transitive closure and they were shown to be the fastest methods for answering reachability queries on large graphs. The implementations of GRAIL and PWAH are by their authors, and the implementation of IL is by the authors of PWAH. We set a parameter k for GRAIL to 2. All algorithms are implemented in C++ using standard template library (STL).

We used real-world networks with more than a million vertices that have been used in the literature [18, 19]. The numbers of vertices and edges (after contracting SCCs) are shown in Table 1.

ff/successors: This is a graph used for source code analysis of Firefox [18].

citeseerx, cit-patents: These are citation networks from CiteSeerX[1] and US patents[2] [19].

go-uniprot: This is the joint graph of Gene Ontology terms and annotation files from UniProt[3] [19].

uniprot22m, uniprot100m, and uniprot150m: These are RDF graphs from UniProt database [19]. We note that underlying graphs of these graphs are very close to trees.

We also conducted experiments on even larger synthetic graphs to show the scalability of our methods. These graphs are created as follows. We first randomly determine the topological order of 10 million vertices. Then we randomly connect two non-adjacent vertices $|E|$ times, where $|E|$ is chosen as a parameter. Note that the direction of each edge is uniquely determined by the topological order.

[1] `http://citeseer.ist.psu.edu/`
[2] `http://snap.stanford.edu/data/`
[3] `http://www.uniprot.org/`

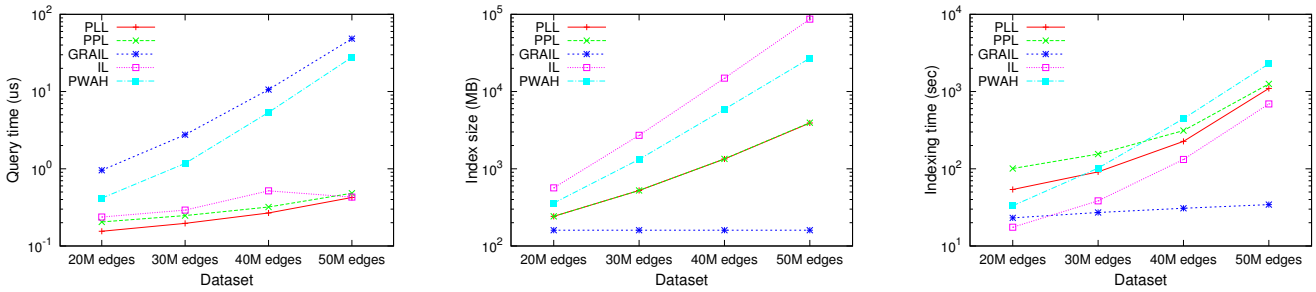
6.2 Performance on Real-World Networks

First, we compared PLL and PPL with existing methods on real-world networks. Tables 2, 3, and 4 show the summary of our experiments.

Table 2 shows the query time on real-world networks. PLL and PPL outperform all the other methods in general. IL also performs quite well especially on citeseerx, but in many cases, PLL is about twice faster than IL. This is possibly because of compactness of labels and simplicity of the query processing procedure of PLL. PPL is slightly slower than PLL since answering queries by PPL is a little more complicated than PLL. PWAH and GRAIL are comparable on very sparse graphs, but they get very slow on the other graphs.

Table 3 suggests that the index size of PLL and PPL are reasonable, though there is no doubt that GRAIL is the most memory-efficient method. On uniprot22m, uniprot100m, and uniprot150m, PLL and PPL perform the best, but the difference on these datasets is not very significant. This may be due to the sparseness of these graphs, which makes it easier to compress the transitive closure by using IL or PWAH. IL and PWAH perform better than PLL and PPL on ff/successors. On the other hand, PLL and PPL outperform IL and PWAH on citeseerx and cit-patents. The index size of PLL and PPL is about 3% of IL and 12% of PWAH on cit-patents. We can say that PLL and PPL are robust in the sense that it only takes moderate space, less than 1GB, on all graphs in the experiments. As for the difference between PLL and PPL, PLL is slightly more space-efficient than PPL in most cases since we need two integers to represent each element in a label PPL whereas we only need one integers in PLL. However, the result on ff/successors shows that PLL has a potential to represent reachability in a more efficient way than PPL in some cases.

Then we look at Table 4, which shows indexing time on real-world networks. GRAIL constantly shows great performance in indexing time since the number of elements in labels is linear in the number of vertices. Still, indexing time of PLL and PPL is acceptable, while they are relatively slow. They are even faster than IL and PWAH on cit-patents. This suggests that PLL and PPL work well on large and mildly dense graphs. IL performs quite well except on cit-patents, and PWAH needs approximately 1.5 to 2.5 times longer time than IL.



(a) Query time (b) Index size (c) Indexing time

Figure 2: Performance comparison on synthetic graphs. GRAIL answers queries slowly, and IL and PWAH consume large index space, while our PLL and PPL achieve remarkable trade-offs.

6.3 Performance on Synthetic Graphs

Second, we compared PLL and PPL with existing methods on synthetic graphs. Query time, index size, and indexing time on synthetic graphs are shown in Figure 2. These synthetic graphs have ten million vertices and number of edges ranges from twenty million to fifty million. Note that these figures are drawn with logarithmic-scale y-axis.

Figure 2a shows that PLL and PPL achieve very fast query time. The query time of PLL, PPL and IL increase very slowly as the number of edges becomes larger, within a microsecond even on the graph with 50 million edges. On the other hand, the query time of GRAIL and PWAH grows fast and exceeds 10 microseconds on that graph.

In Figure 2b, the index size of IL and PWAH become larger drastically as the graph becomes dense. The index size of PLL and PPL grow relatively slowly, and that of GRAIL does not change by the number of edges.

Figure 2c shows that GRAIL outperforms other methods in indexing time, especially on relatively dense graphs. PLL and PPL are relatively slow on very sparse graphs. However, these two methods overtake IL and PWAH as the graph becomes dense.

As a whole, we can say that PLL and PPL outperform other methods on relatively dense graphs, achieving very fast query time and moderate index size. The index size of IL and PWAH becomes very large on dense graphs, and the query time of GRAIL and PWAH becomes very slow on these graphs. These experimental results show that PLL and PPL has a potential to handle real-world networks larger than those we used in the experiments.

7. ACKNOWLEDGMENTS

Yosuke Yano and Yuichi Yoshida are supported by JST, ERATO, Kawarabayashi Large Graph Project. Takuya Akiba and Yoichi Iwata are supported by Grant-in-Aid for JSPS Fellows (256563 and 256487). Yuichi Yoshida is supported by JSPS Grant-in-Aid for Research Activity Start-up (24800082) and MEXT Grant-in-Aid for Scientific Research on Innovative Areas (24106001).

8. REFERENCES

[1] T. Akiba, Y. Iwata, and Y. Yoshida. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *SIGMOD*, 2013. to appear.

[2] D. Chamberlin. XQuery: a query language for XML. In *SIGMOD*, pages 682–682, 2003.

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

[4] J. Cheng, J. X. Yu, X. Lin, H. Wang, and P. S. Yu. Fast computation of reachability labeling for large graphs. In *EDBT*, pages 961–979, 2006.

[5] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. *SIAM Journal on Computing*, 32(5):1338–1355, 2003.

[6] R. Jin and G. Wang. Simple, fast, and scalable reachability oracle. *arXiv:1305.0502*, May 2013.

[7] 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.

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

[9] M. E. J. Newman, S. H. Strogatz, and D. J. Watts. Random graphs with arbitrary degree distributions and their applications. *Physical Review E*, 64(2):026118, 2001.

[10] E. Nuutila. *Effcient transitive closure computation in large digraphs*. PhD thesis, Finnish Academy of Technology, 1995.

[11] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. *Transactions on Database Systems*, 34(3):16:1–16:45, Sept. 2009.

[12] T. Reps. Program analysis via graph reachability. In *ILPS*, pages 5–19, 1997.

[13] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, pages 49–61, 1995.

[14] N. Robertson and P. D. Seymour. Graph minors. III. planar tree-width. *Journal of Combinatorial Theory, Series B*, 36(1):49–64, 1984.

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

[16] K. Simon. An improved algorithm for transitive closure on acyclic digraphs. *Theoretical Computer Science*, 58(1):325–346, 1988.

[17] J. van Helden, A. Naim, R. Mancuso, M. Eldridge, L. Wernisch, D. Gilbert, and S. J. Wodak. Representing and analysing molecular and cellular function using the computer. *Biological Chemistry*, 381(9-10):921–935, 2000.

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

[19] H. Yildirim, V. Chaoji, and M. Zaki. GRAIL: a scalable index for reachability queries in very large graphs. *The VLDB Journal*, 21(4):509–534, 2012.

[20] Z. Zhang, J. Yu, L. Qin, Q. Zhu, and X. Zhou. I/O cost minimization: reachability queries processing over massive graphs. In *EDBT*, pages 468–479, 2012.