Simple, Fast, and Scalable Reachability Oracle

Ruoming Jin
Department of Computer Science
Kent State University
jin@cs.kent.edu

Guan Wang
Department of Computer Science
Kent State University
gwang@cs.kent.edu

ABSTRACT

A reachability oracle (or hop labeling) assigns each vertex v two sets of vertices: $L_{out}(v)$ and $L_{in}(v)$, such that u reaches v iff $L_{out}(u) \cap L_{in}(v) \neq \emptyset$. Despite their simplicity and elegance, reachability oracles have failed to achieve efficiency in more than ten years since their introduction: the main problem is high construction cost, which stems from a set-cover framework and the need to materialize transitive closure. In this paper, we present two simple and efficient labeling algorithms, *Hierarchical-Labeling* and *Distribution-Labeling*, which can work on massive real-world graphs: their construction time is an order of magnitude faster than the set-cover based labeling approach, and transitive closure materialization is not needed. On large graphs, their index sizes and their query performance can now beat the state-of-the-art transitive closure compression and online search approaches.

1. INTRODUCTION

As one of the most fundamental graph operators, reachability has drawn much research interest in recent years [5, 30, 8, 28, 17, 9, 6, 16, 33, 31, 4, 29, 15, 7] and seems to continue fascinating researchers with new focuses [19, 32, 18] and new variants [13, 10, 25]. The basic reachability query answers whether a vertex u can reach another vertex v using a simple path ($?u \rightarrow v$) in a directed graph. It has a wide range of applications from software engineering, to distributed computing, to biomedical and social network analysis, to XML and the semantic web, among others.

The majority of the existing reachability computation approaches belong to either transitive closure materialization (compression) [2, 21, 30, 17, 29] or online search [5, 28, 31]. The transitive closure compression approaches tend to be faster but generally have difficulty scaling to massive graphs due to the precomputation and/or memory cost. Online search is (often one or two orders of magnitude) slower but can work on large graphs [31, 19]. The latest research [19] introduces a unified SCARAB method based on "reachability backbone" (similar to the highway in the transportation network) to deal with their limitations: it can both help scale the transitive closure approaches and speed up online search. However, the query performance of transitive closure approaches tends to be slowed down and they may still not work if the size of the reachability backbone remains too large [19].

The reachability oracle, more commonly known as hop labeling, [11, 27] is an interesting third category of approaches which lie between transitive closure materialization and online search. Each vertex v is labeled with two sets: $L_{out}(v)$, which contains hops (vertices) v can reach; and $L_{in}(v)$, which contains hops that can reach v. Given $L_{out}(u)$ and $L_{in}(v)$, but nothing else, we can compute if u reaches v by determining whether there is at least a common hop, $L_{out}(u) \cap L_{in}(v) \neq \emptyset$. The idea is simple, elegant, and seems very promising: hop labeling can be considered as a factorization of the binary matrix of transitive closure; thus it should be able to deliver more compact indices than the transitive closure and also offer fast query performance.

Unfortunately, after more than ten years since its first proposal [11] and a list of worthy attempts [23, 8, 9, 16, 4], hop labeling or reachability oracle, still eludes us and still fails to meet its expectations. Despite its appealing theoretical nature, recent studies [19, 29, 10, 31] all seem to confirm its inability to handle real-world large graphs: hop labeling is expensive to construct, taking much longer time than other approaches, and can barely work on large graphs, due to prohibitive memory cost of the construction algorithm. Many studies [19, 29, 10, 31] also show up to an order of magnitude slower query performance compared with the fastest transitive closure compression approaches (though we discover the underlying reason is mainly due to the implementation of hop labeling L_{out} and L_{in}; employing a sorted vector/array instead of a set can significantly eliminate the query performance gap).

The high construction cost of the reachability oracle is inherent to the existing labeling algorithms and directly results in the scalability bottleneck. In order to minimize the labeling size, many algorithms [11, 23, 8, 16, 4] rely on a greedy set-cover procedure, which involves two costly operators: 1) repetitively finding densest subgraphs from a large number of bipartite graphs; and 2) materialization of the entire transitive closure. The latter is needed since each reachability pair needs to be explicitly covered by a selected hop. Even with concise transitive closure representation, such as using geometric format [8], or reducing the covered pairs using 3-hop [16, 4], the overall construction complexity is still close to or more than $O(n^3)$, which is still too expensive for large graphs. Alternative labeling algorithms [27, 9] try to use graph separators, but only special graph classes, such as planar graphs, consisting of small graph separators, can adopt such techniques well [27]. For general graphs, the scalability of such approach [9] is limited by the lack of good scalable partition algorithms for discovering graph separators on large graphs.

Can the reachability oracle be practical? Is it a purely theoretical concept which can only work on small toy graphs, or it is a powerful tool which can shape reality and can work on real-world large graphs with millions of vertices and edges? Arguably, this is one of the most important unsolved puzzles in reachability computation. This work resolves these questions by presenting two simple and efficient labeling algorithms, *Hierarchical-Labeling* and *Distribution-Labeling*, which can work on massive real-world graphs. Their construction costs are as fast as the state-of-the-art transitive closure compression approaches, there is no expensive transitive closure materialization, dense subgraph detection, or greedy set-cover procedure, there is no need for graph separators, and on

large graphs, their index sizes and their query performance beat the state-of-the-art transitive closure compression and online search approaches [17, 29, 19, 29, 10, 31]. Using these two algorithms, the power of hop labeling is finally unleashed and a fast, compact and scalable reachability oracle becomes a reality.

The rest of the paper is organized as follows. In Section 2, we review the prior work on reachability. In Section 3, we give an overview of the basic ideas of constructing a fast, compact and scalable reachability oracle. In Section 4, we present the *Hierarchical-Labeling* algorithm, which is based on a hierarchical decomposition of a DAG (direct acyclic graph). In Section 5, we introduce the *Distribution-Labeling* algorithm, which utilizes a total vertex order. In Section 6, we report the detailed experimental study on these two new labeling algorithms compared with the state-of-the-art reachability computation approaches. We offer concluding remarks in Section 7.

2. RELATED WORK

To compute the reachability, the directed graph is typically transformed into a DAG (directed acyclic graph) by coalescing strongly connected components into vertices, avoiding the trivial case where vertices reach each other in a strongly connected component. The size of the DAG is often much smaller than that of the original graph and is more convenient for reachability indexing. Let $G = (V, E)$ be the DAG for a reachability query, with number of vertices $n = |V|$ and number of edges $m = |E|$.

2.1 Transitive Closure and Online Search

There are two extremes in computing reachability. At one end, the entire transitive closure (TC) of G is precomputed and fully materialized (often in a binary matrix). Since the reachability between any pair is recorded, reachability can be answered in constant time, though the $O(n^2)$ storage is prohibitive for large graphs. At the other end, DFS/BFS can be employed. Though it does not need an additional index, its query answering time is too slow for large graphs. As we mentioned before, the majority of the reachability computation approaches aim to either compress the transitive closure [2, 14, 21, 30, 17, 7, 29, 15] or to speed up the online search [5, 28, 31].

Transitive Closure Compression: This family of approaches aims to compress the transitive closure – each vertex u records a compact representation of $TC(u)$, i.e., all the vertices it reaches. The reachability from vertex u to v is computed by checking vertex v against $TC(u)$. Representative approaches include chain compression [14, 6], interval or tree compression [2, 21], dual-labeling [30], path-tree [17], and bit-vector compression [29]. Using interval-compress as an example, any contiguous vertex segment in the original $TC(u)$ is represented by an interval. For instance, if $TC(u)$ is $\{1, 2, 3, 4, 8, 9, 10\}$, it can be represented as two intervals: $[1, 4]$ and $[8, 10]$.

Existing studies [29, 31, 19] have shown these approaches are the fastest in terms of query answering since checking against transitive closure $TC(u)$ is typically quite simple (linear scan or binary search suffices); in particular, the interval and path-tree approaches seem to be the best in terms of query answering performance. However, the transitive closure materialization, despite compression, is still costly. The index size is often the reason these approaches are not scalable on large graphs [31, 19].

Fast Online Search: Instead of materializing the transitive closure, this set of approaches [5, 28, 31] aims to speed up the online search. To achieve this, auxiliary labeling information per vertex is precomputed and utilized for pruning the search space. Using the state-of-the-art GRAIL [31] as an example, each vertex is assigned multiple interval labels where each interval is computed by a random depth-first traversal. The interval can help determine whether a vertex in the search space can be immediately pruned because it never reaches the destination vertex v.

The pre-computation of the auxiliary labeling information in these approaches is generally quite light; the index size is also small. Thus, these approaches can be applicable to very large graphs. However, the query performance is not appealing; even the state-of-the-art GRAIL can be easily one or two orders of magnitude slower than the fast interval and path-tree approaches [31, 19]. For very large graphs, these approaches may be too slow for answering reachability query.

2.2 Reachability Oracle

The reachability oracle [11, 27], also refer to as hop labeling, was pioneered by Cohen *et al*. [11]. Though it also encodes transitive closure, it does not explicitly compress the transitive closure of each individual vertex independently (unlike the transitive closure compression approaches). Here, each vertex v is labeled with two sets: $L_{out}(v)$, which contains hops (vertices) v can reach; and $L_{in}(v)$, which contain hops that can reach v. Given $L_{out}(u)$ and $L_{in}(v)$, but nothing else, we can compute if u reaches v by determining whether there is a common hop, $L_{out}(u) \cap L_{in}(v)$. In fact, a reachability oracle can be considered as a factorization of the binary matrix of transitive closure [16]; and thus more compact indices are expected from such a scheme.

The seminal 2-hop labeling [11] aims to minimize the reachability oracle size, which is the total label size $\sum(|L_{out}(u)| + |L_{in}(u)|)$. It employs an approximate (greedy) algorithm based on set-covering which can produce a reachability oracle with size no larger than the optimal one by a logarithmic factor. The optimal 2-hop index size is conjectured to be $\tilde{O}(nm^{1/2})$.

The major problem of the 2-hop indexing approach is its high construction cost: *The greedy set-covering algorithm needs to iteratively find a vertex v associated with two subsets of vertices X and Y which utilizes v as the intermediate hop, i.e., $v \in L_{out}(x), x \in X$ and $v \in L_{in}(y), y \in Y$. To select vertex v and its associated X and Y, the greedy procedure utilizes* **price***, which measures the cost-benefit tradeoff between recording the vertex in $L_{out}(x)$ and $L_{in}(y)$ (cost) and the number of reachability pairs being newly covered (benefit) by such labeling: $\frac{|X|+|Y|}{|X \times Y \setminus C|}$, where C are those reachable pairs already covered by previously selected hops.* This selection step can be transformed into the problem of finding a densest subgraph in n bipartite graphs. The approximate algorithm to solve this subproblem is in the linear order with respect to the number of edges in the bipartite graph. Such an iterative approach can be as costly as $O(n^3|TC|)$, where $|TC|$ is the total size of transitive closure.

A number of approaches have sought to reduce construction cost through speeding up the set cover procedure [23], using concise transitive closure representation [8], or reducing the covered pairs using 3-hop [16, 4]. However, they still need to repetitively find densest subgraphs from a large number of bipartite graphs and to materialize the transitive closure to explicitly confirm each reachable pair is covered by the hop labeling. Alternative labeling algorithms [27, 9] try to use graph separators, but only special graph classes, such as planar graphs, consisting of small graph separators, can adopt such technique well [27]. For general graphs, the scalability of such approach [9] is limited by the lack of good scalable partition algorithms for discovering graph separators on large graphs.

2.3 Reachability Backbone and SCARAB

In the latest study [19], the authors introduce a general framework, referred to as SCARAB (SCAling ReachABility), for scaling

the existing reachability indices (including both transitive closure compression and hop labeling approaches) and for speeding up the online search approaches. The central idea is to leverage a "reachability backbone" (like highways in a road network), which carries the major "reachability flow" information.

Formally, the reachability backbone $G^\star = (V^\star, E^\star)$ of graph G is defined as a subgraph of the transitive closure of G ($E^\star \subseteq TC(G)$), such that *for any reachable (u, v) pair, there must exist local neighbors $u^\star \in V^\star$, $v^\star \in V^\star$ with respect to locality threshold ϵ, i.e., $d(u, u^\star) \leq \epsilon$ and $d(v^\star, v) \leq \epsilon$, and $u^\star \rightarrow v^\star$.* Here $d(u, u^\star)$ is the shortest path distance from u to $u^\star$ where the weight of each edge is unit. To compute the reachability from u to v, u collects a list of local outgoing backbone vertices (entries) using forward BFS, and v collects a list of local incoming backbone vertices (exits) using backward BFS. Then an existing reachability approach can be utilized to determine if there is a local entry reaching a local exit on the reachability backbone $G^\star$.

Two algorithms are developed to approximate the minimal backbone, one based on set-cover and the other based on BFS. The latter, referred to as *FastCover*, is particularly efficient and effective, with time complexity $O(\sum_{v \in V} |N_\epsilon(v)|log|N_\epsilon(v)|+|E_\epsilon(v)|)$, where $N_\epsilon(v)$ ($E_\epsilon(v)$) is the set of vertices (edges) v can reach in ϵ steps. Experiments shows that even with ϵ, the size of the reachability backbone is significantly smaller than the original graph (about $1/10$ the number of vertices of the original graph). As we will discuss later, our first *Hierarchical-Labeling* algorithm is directly inspired by the reachability backbone and effectively utilizes it for reachability oracle construction.

Though the scaling approach is quite effective for helping deal with large graphs, it is still constrained by the power of the original index approaches. For many large graphs, the reachability backbone can still be too large for them to process as shown in the experiment study in [19]. Also, using the reachability backbone slows down the query performance of the transitive closure compression and hop labeling approaches (typically two or three times slower than the original approaches) on the graphs where they can still run. In addition, theoretically, the reachability backbone could be applied recursively; this may further slow down query performance. In [19], this option is not studied.

We also note that in [10], a new variant of reachability queries, k-hop reachability, is introduced and studied. It asks whether vertex u can reach v within k steps. This problem can be considered a generalization of the basic reachability, where $k = \infty$. A k-reach indexing approach is developed and the study shows that approach can handle basic reachability quite effectively (with comparable query performance to the fastest transitive closure compression approaches on small graphs). The k-reach indexing approach is based on vertex cover (a set of vertices covers all the edges in the graph). To compute the reachability from u to v, each vertex only needs to access their immediate neighbors in the vertex cover; and the pairwise reachability between any two vertices in the set cover is precomputed and fully materialized (for basic reachability computation). It is easy to see that *this vertex cover based approach is a reachability backbone with $\epsilon = 1$ as defined in [19]*. But this study directly materializes the transitive closure between any pair of vertices in the vertex cover, where in [19], the existing reachability indices are used. Thus, for very large graphs where the vertex cover is often large, the pair-wise reachability materialization is not feasible. (This observation is also confirmed through our experimental study in Section 6).

3. APPROACH OVERVIEW

In a reachability oracle of graph G, each vertex v is labeled with two sets: $L_{out}(v)$, which contains hops (vertices) v can reach; and $L_{in}(v)$, which contain hops that can reach v. A labeling is *complete* if and only if for any vertex pair where $u \rightarrow v$, $L_{out}(u) \cap L_{in}(v) \neq \emptyset$. The goal is to minimize the total label size, i.e., $\sum(|L_{out}(u)| + |L_{in}(u)|)$. A smaller reachability oracle not only help to fit the index in main memory, but also speeds up the query processing (with $O(|L_{out}(u)| + |L_{in}(v)|)$ time complexity).

As we mentioned before, though the existing set-cover based approaches [11, 23, 8, 16, 4] can achieve approximate optimal labeling size within a logarithmic factor, its computational and memory cost is prohibitively expensive for large graphs. The labeling process not only needs to materialize the transitive closure, but it also uses an iterative set-cover procedure which repetitively invokes dense subgraph detection. The reason for such complicated algorithm is that the following two criteria need to be met: 1) a labeling must be complete, and 2) we wish the labeling to be minimal. The existing approach [11, 16] essentially transforms the labeling problem into a set cover problem with the cost of constructing the ground set (which is the entire transitive closure) and dynamic generation and selection of good candidate sets (through dense subgraph detection).

To achieve efficient labeling which can work on massive graphs, the following issues have to appropriately handled:

1. (Completeness without Transitive Closure): Can we guarantee labeling completeness without materialization of the transitive closure? Even compact [8] or reduced [16] materialization can be expensive for large graphs. Thus, the key is whether a labeling process can avoid the need to explicitly check whether a reachable pair (against some form of transitive closure) is covered by the existing labeling.

2. (Compactness without Optimization): Without the set-cover, it seems difficult to produce bounded approximate optimal labeling. But this does not mean that a compact reachability oracle cannot be produced. Clearly, each vertex should not record every valid hop in the labeling. In the set-cover framework, a price is computed to determine whether a vertex should be added to certain vertex labels. What other criteria can help determine the importance of hops (vertices) so that each vertex can be more selective in what it records?

In this paper, we investigate how the hierarchical structure of a DAG can help produce a complete and compact reachability oracle. The basic idea is as follows: assuming a DAG can be represented in a hierarchical (multi-level) structure, such that the lower-level reachability needs to go through upper-level (but vice versa), then we can somehow recursively broadcast the upper-level labels to lower-level labels. In other words, the labels of lower-level vertices (L_{in} and L_{out}) can directly utilize the already computed labels in the upper-level. Thus, on one side, by using the hierarchical structure, the completeness of labeling can be automatically guaranteed. On the other side, it provides an importance score (the level) of every hop; and each vertex only records those hops whose levels are higher than or equal to its own level. We note that there have been several studies [24, 22, 12, 20, 3, 1] using the hierarchical structure for shortest path distance computation on road networks; however, how to construct and utilize the hierarchical structure for reachability computation has not been fully addressed. To the best of our knowledge, this is the first study to construct a fast and scalable reachability oracle based on hierarchical DAG decomposition.

Now, to turn such an idea into a fast labeling algorithm for reachability oracle, the following two research questions need to be answered: 1) What hierarchical structure representation of a DAG can be used? 2) How should L_{out} and L_{in} be computed efficiently using a given hierarchical structure? In this paper, we introduce two

fast labeling algorithms based on different hierarchical structures of a DAG:

Hierarchical-Labeling (Section 4): In this approach, the hierarchical structure is produced by a recursive reachability backbone approach, i.e., finding a reachability backbone $G^\star$ from the original graph G and then applying the backbone extraction algorithm on $G^\star$. Recall that the reachability backbone is introduced by the latest SCARAB framework [19] which aims to scale the existing reachability computation approaches. Here we apply it recursively to provide a hierarchical DAG decomposition. Given this, a fast labeling algorithm is designed to quickly compute L_{in} and L_{out} one vertex by one vertex in a level-wise fashion (from higher level to lower level).

Distribution-Labeling (Section 5): In this approach, the sophisticated reachability backbone hierarchy is replaced with the simplest hierarchy – a total order, i.e.., each vertex is assigned a unique level in the hierarchy structure. Given this, instead of computing L_{in} and L_{out} one vertex at a time, the labeling algorithm will *distribute* the hop one by one (from higher order to lower order) to L_{in} and L_{out} of other vertices. The worst cast computation complexity of this labeling algorithm is $O(n(n+m))$ (of the same order as transitive closure computation), though in practice it is much faster than the transitive closure computation.

In the experimental study (Section 6), through an extensive study on both real and synthetic graphs, we found that both labeling approaches not only are fast (up to an order of magnitude faster than the best set-cover based approach [11, 16]) and work on massive graphs, but most surprisingly, their label sizes are actually smaller than the set-cover based approaches.

4. HIERARCHICAL LABELING

Before we proceed to discuss the *Hierarchical Labeling* approach, let us formally introduce the one-side reachability backbone (first defined in [19] for scaling the existing reachability computation), which serves as the basis for hierarchical DAG decomposition and the labeling algorithm.

DEFINITION 1. *(**One-Side Reachability Backbone** [19]) Given DAG G, and local threshold ϵ, the one-side reachability backbone $G^\star = (V^\star, E^\star)$ is defined as follows: 1) $V^\star \subseteq V$, such that for any vertex pair (u, v) in G with $d(u, v) = \epsilon$, there is a vertex $v^\star$ with $d(u, v^\star) \leq \epsilon$ and $d(v^\star, v) \leq \epsilon$; 2) $E^\star$ includes the edges which link vertex pair $(u^\star, v^\star)$ in $V^\star$ with $d(u^\star, v^\star) \leq \epsilon + 1$.*

Note that $E^\star$ can be simplified as a transitive reduction [19] (the minimal edge set preserving the reachability). Since computing transitive reduction is as expensive as transitive closure, rules like the following can be applied: $(u^\star, v^\star) \in E^\star$ can be removed if there is another intermediate vertex $x \in V^\star$ (not $u^\star$ and $v^\star$) with $d(u^\star, x) \leq \epsilon$ and $d(x, v^\star) \leq \epsilon$.

EXAMPLE 4.1. *As a simple example, let $V^\star$ be a vertex cover of G, i.e., at least one end of an edge in E is in $V^\star$; and let $E^\star$ contain all edges $(u^\star, v^\star) \in V^\star \times V^\star$, such that $d(u^\star, v^\star) \leq 2$. Then, $G^\star = (V^\star, E^\star)$ is one-side reachability backbone with $\epsilon = 1$. In Figure 1(b), G_1 is the reachability backbone of graph G_0 (Figure 1(a)) for $\epsilon = 2$.*

The important property of the one-side reachability backbone is that *for any non-local pair (u, v): $u \to v$ and $d(u, v) > \epsilon$, there always exists $u^\star \in V^\star$ and $v^\star \in V^\star$, such that $d(u, u^\star) \leq \epsilon$, $d(v^\star, v) \leq \epsilon$, and $u^\star \to v^\star$.* This property will serve as the key tool for recursively computing L_{out} and L_{in}. In [19], the authors develop the *FastCover* algorithm employing ϵ-step BFS for each vertex for discovering the one-side reachability backbone. They also show that when $\epsilon = 2$, the backbone can already be significantly reduced. To simplify our discussion, in this paper, we will focus on using the reachability backbone with $\epsilon = 2$ though the approach can be applied to other locality threshold values.

Below, Subsection 4.1 presents the hierarchical decomposition of a DAG and the labeling algorithm using this DAG; Subsection 4.2 discusses the correctness of the labeling approach and its time complexity.

4.1 Hierarchical DAG Decomposition and Labeling Algorithm

Let us start with the hierarchical DAG decomposition which is based on the reachability backbone.

DEFINITION 2. *(**Hierarchical DAG Decomposition**) Given DAG $G = (V, E)$, a vertex hierarchy is defined as $V_0 = V \supset V_1 \supset V_2 \supset \cdots \supset V_h$, with corresponding edge sets $E_0, E_1, E_2 \cdots E_h$, such that $G_i = (V_i, E_i)$ is the (one-side) reachability backbone of $G_{i-1} = (V_{i-1}, E_{i-1})$, where $0 < i \leq h$. The final graph $G_h = (V_h, E_h)$ is referred to as the core graph.*

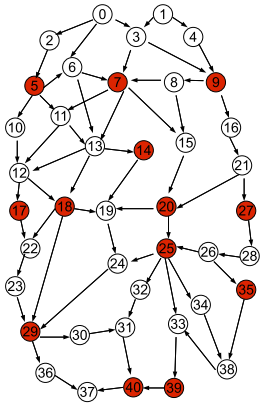
Intuitively, the vertex hierarchy shows the relative importance of vertices in terms of reachability computation. The lower level reachability computation can be resolved using the higher level vertices, but not the other way around. In other words, the reachability (backbone) property is preserved through the vertex hierarchy.

LEMMA 1. *Assuming $u \in V_i, v \in V_i$, u reaches v in G ($u \xrightarrow{G} v$) iff u reaches v in G_i ($u \xrightarrow{G_i} v$). Furthermore, for any non-local vertex pairs $(u_i, v_i) \in V_i$, $d(u_i, v_i|G_i) > \epsilon$ (the distance in G_i), there always exists $u_{i+1} \in V_{i+1}$ and $v_{i+1} \in V_{i+1}$, such that $d(u_i, u_{i+1}|G_i) \leq \epsilon$, $d(v_{i+1}, v_i|G_i) \leq \epsilon$, and $u_{i+1} \xrightarrow{G_{i+1}} v_{i+1}$.*

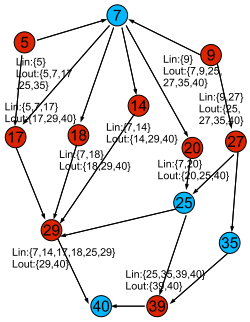
Proof Sketch: The first claim: assuming $u \in V_i, v \in V_i$, u reaches v in G ($u \xrightarrow{G} v$) iff u reaches v in G_i ($u \xrightarrow{G_i} v$), can be proved by induction. The base case where $i = 1$ is clearly true based on the reachability backbone definition (the reachability backbone will preserve the reachability between vertices in the backbone as they appear in the original graph). Assuming this is true for all $i < k$, then it also holds to be true for $i = k$. This is because for any $u \in V_i, v \in V_i$, we must have $u \in V_{i-1}$ and $v \in V_{i-1}$. Based on the reachability backbone definition, we have $u \xrightarrow{G_{i-1}} v$ iff $u \xrightarrow{G_{i-1}} v$. Then based on the induction, we have G ($u \xrightarrow{G_0 = G} v$) iff u reaches v in G_i ($u \xrightarrow{G_i} v$). The second claim directly follows the reachability definition. □

EXAMPLE 4.2. *Figure 1 shows a vertex hierarchy for DAG G_0 (a), where $V_1 = \{5, 7, 9, \cdots, 40\}$ (b) and $V_2 = \{7, 25, 35, 40\}$ (c). G_1 is the (one-side) reachability backbone of G_0 and G_2 is the corresponding (one-side) reachability backbone of G_1.*

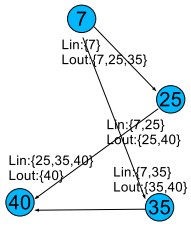
To utilize the hierarchical decomposition for labeling, let us further introduce a few notations related to the vertex hierarchy. Each vertex v is assigned to a unique *level*: $level(v) = i$ iff $v \in V_i \setminus V_{i+1}$, where $0 \leq i \leq h$ and $V_{h+1} = \emptyset$. (Later, we will show that each vertex is labeled at its corresponding level using G_i and labels of vertices from higher levels). Assuming v is at level i, i.e., $level(v) = i$, let $N^k_{out}(v|G_i)$ ($N^k_{in}(v|G_i)$) be the v's *k-degree outgoing (incoming) neighborhood*, which includes all the vertices v can reach (reaching v) within k steps in G_i. Finally, for any vertex v at level $i < h$, its corresponding outgoing (incoming) backbone vertex set $\boldsymbol{\mathcal{B}}^\epsilon_{out}(v)$ ($\boldsymbol{\mathcal{B}}^\epsilon_{in}(v)$) is defined as:



(a) Original Graph G_0



(b) First Level Backbone G_1



(c) Second Level Backbone G_2

Labeling on Original Graph	
Lin List	Lout List
0 : 0	0 : 0, 2, 3, 5, 6, 7, 9, 17, 25, 27, 35, 40
1 : 1	1 : 1 ,3, 4, 7, 9, 25, 27, 35, 40
2 : 0, 2	2 : 5, 7, 17, 25, 35
3 : 0, 1, 3	3 : 3, 7, 9, 20, 25, 27, 35, 40
4 : 1, 4	4 : 4, 7, 9, 25, 27, 35, 40
6 : 0, 5, 6	6 : 6, 7, 14, 18, 25, 29, 35, 40
8 : 8, 9	8 : 7, 8, 15, 20, 25, 35, 40
...	...
38 : 7, 25, 34, 35, 38	38 : 33, 38, 39, 40

(d) Hop Labeling for V_0

Figure 1: Running Examples of Hierarchical-Labeling

$$\mathcal{B}^{\epsilon}_{out}(v) = \{u \in V_{i+1} | d(v,u|G_i) \leq \epsilon \text{ and there is no other vertex } x \in V_{i+1}, \text{ such that } d(v,x|G_i) \leq \epsilon \wedge d(x,u|G_i) \leq \epsilon (v \to x \to u)\} \quad (1)$$

$$\mathcal{B}^{\epsilon}_{in}(v) = \{u \in V_{i+1} | d(u,v|G)_i) \leq \epsilon \text{ and there is no other vertex } y \in V_{i+1}, \text{ such that } d(u,y|G_i) \leq \epsilon \wedge d(y,v|G_i) \leq \epsilon (u \to y \to v)\} \quad (2)$$

Now, let us see how the labeling algorithm works given the hierarchical decomposition. Contrary to the decomposition process which proceeds from the lower level to higher level (like peeling), the labeling performs from the higher level to the lower level. Specifically, it first labels the core graph G_h and then iteratively labels the vertex at level $h-1$ to level 0.

Labeling Core Graph G_h: Theoretically, the diameter of the core graph G_h is no more than ϵ (the pairwise distance between any vertex pair in G_h is no more than ϵ), and thus no more reachability backbone is needed ($V_{h+1} = \emptyset$). In this case, for a vertex $v \in V_h$ ($level(v) = h$), the basic labeling can be as simple as follows:

$$L_{out}(v) = N^{\lceil \epsilon/2 \rceil}_{out}(v|G_h); \quad L_{in}(v) = N^{\lceil \epsilon/2 \rceil}_{in}(v|G_h) \quad (3)$$

The labeling is clearly complete for G_h as any reachable pair is within distance ϵ. Alternatively, since the core graph is typically rather small, we can also employ the existing 2-hop labeling algorithm [11, 23] to perform the labeling for core graphs. Given this, practically, the decomposition can be stopped when the vertex set V_h is small enough (typically less than $10K$) instead of making its diameter less than or equal to ϵ.

Labeling Vertices with Lower Level i ($0 \leq i < h$): After the core graph is labeled, the remaining vertices will be labeled in a level-wise fashion from higher level $h-1$ to lower level (until level 0). For each vertex v at level $0 \leq i < h$, assuming all vertices in the higher level ($> i$) have been labeled (L_{out} and L_{in}), then the following simple rule can be utilized for labeling v:

$$L_{out}(v) = N^{\lceil \epsilon/2 \rceil}_{out}(v|G_i) \cup \Big(\bigcup_{u \in \mathcal{B}^{\epsilon}_{out}(v|G_i)} L_{out}(u) \Big) \quad (4)$$

$$L_{in}(v) = N^{\lceil \epsilon/2 \rceil}_{in}(v|G_i) \cup \Big(\bigcup_{u \in \mathcal{B}^{\epsilon}_{in}(v|G_i)} L_{out}(u) \Big) \quad (5)$$

Basically, the label of $L_{out}(v)$ ($L_{in}(v)$) at level i consists of two parts: the outgoing (incoming) $\lceil \epsilon/2 \rceil$-degree neighbors of v in G_i, and the labels from its corresponding outgoing (incoming) backbone vertex set $\mathcal{B}^{\epsilon}_{out}(v|G_i)$ ($\mathcal{B}^{\epsilon}_{in}(v|G_i)$). In particular, if $\epsilon = 2$ (the typical locality threshold), then each vertex v basically records its direct outgoing (incoming) neighbors in G_i and the labels from its backbone vertex set.

Algorithm 1 Hierachical-Labeling($G = (V, E)$)

1: Perform Hierarchical Decomposition of G based on Definition 2;
2: Labeling core graph G_h;
3: $i \leftarrow h-1$;
4: **while** $i \geq 0$ {Labeling V_i from higher level to lower} **do**
5: **for each** $v \in V_i \setminus V_{i+1}$ {labeling each vertex specific for V_i} **do**
6: $L_{out}(v) \leftarrow N^{\lceil \epsilon/2 \rceil}_{out}(v|G_i) \cup (\bigcup_{u \in \mathcal{B}^{\epsilon}_{out}(v|G_i)} L_{out}(u))$
7: $L_{in}(v) \leftarrow N^{\lceil \epsilon/2 \rceil}_{in}(v|G_i) \cup (\bigcup_{u \in \mathcal{B}^{\epsilon}_{in}(v|G_i)} L_{out}(u))$
8: **end for**
9: $i \leftarrow i-1$;
10: **end while**

Overall Algorithm: Algorithm 1 sketches the complete Hierarchical-Labeling approach. Basically, we first perform the recursive hierarchical DAG decomposition (Line 1). Then, the vertices at the core graph G_h will be labeled either by Formula 3 or using the existing 2-hop labeling approach (Line 2). Finally, the while-loop performs the labeling from higher level $h-1$ to lower level 0 iteratively (Lines 4-10), where each vertex v in the level i (Lines $5-9$) will be labelled based on Formulas 4 and 5.

EXAMPLE 4.3. *Figure 1 illustrates the Hierarchical-Labeling process, where Figure 1(c) shows the labeling of core graphs. Note that for simplicity, each vertex by default records itself in both L_{in} and L_{out}, and $\epsilon = 2$. Figure 1(b) shows the labeling for vertices in V_1; and Table 1(c) illustrates the labeling of a few vertices in V_0. Taking vertex 14 for example: $L_{in}(14)$ records its direct incoming neighbors in G_1 $\{7, 14\}$ (and itself), and other labels from the labels of its corresponding incoming backbone vertex set $\mathcal{B}^{\epsilon}_{in}(14|G_1) = \{7\}$. Thus, $L_{in}(14) = \{7, 14\}$. Now $L_{out}(14)$ records its direct outgoing neighbors $\{14, 29\}$ and L_{out} of vertex 40 ($\mathcal{B}^{\epsilon}_{out}(14|G_1) = \{40\}$).*

4.2 Algorithm Correctness and Complexity

In the following, we first prove the correctness of the Hierarchical-Labeling algorithm, that is, that it produces a complete labeling: for

any vertex pair (u, v), $u \to v$ iff $L_{out}(u) \cap L_{in}(v) \neq \emptyset$. We then discuss its time complexity.

THEOREM 1. *The Hierarchical-Labeling approach (Algorithm 1) produces a complete labeling for each vertex v in graph G, such that for any vertex pair (u, v): $u \to v$ iff $L_{out}(u) \cap L_{in}(v) \neq \emptyset$.*

Proof Sketch: We prove the correctness through induction: assuming Algorithm 1 produces the correct labeling for V_{i+1}, then it produces the correct labeling for V_i. Basically if for any vertex pair $u^\star$ and $v^\star$ in V_{i+1}, $u^\star \to v^\star$ iff $L_{out}(u^\star) \cap L_{in}(v^\star) \neq \emptyset$, then we would like to show that for any vertex pair u and v in V_i, this also holds. To prove this, we consider four different cases for any u and v in V_{i+1}: 1) $u \in V_i \setminus V_{i+1}$ and $v \in V_i \setminus V_{i+1}$; 2) $u \in V_i \setminus V_{i+1}$ and $v \in V_{i+1}$; 3) $u \in V_{i+1}$ and $v \in V_i \setminus V_{i+1}$; and 4) $u \in V_{i+1}$ and $v \in V_{i+1}$. Since case 4 trivially holds based on the reduction and cases 2 and 3 are symmetric, we will focus on proving cases 1 and 2.

Case 1 ($u \in V_i \setminus V_{i+1}$ and $v \in V_i \setminus V_{i+1}$): We observe: 1) $u \to v$ with $d(u, v) \leq \epsilon$ (local pair) iff there is $x \in V_i$, such that $d(u, x) \leq \lceil \frac{\epsilon}{2} \rceil$ and $d(x, v) \leq \lceil \frac{\epsilon}{2} \rceil$, i.e., $N_{out}^{\lceil \epsilon/2 \rceil}(v|G_i) \cap N_{in}^{\lceil \epsilon/2 \rceil}(v|G_i) \neq \emptyset$;
and 2) $u \to v$ with $d(u, v) > \epsilon$ (non-local pair)
iff there are backbone vertices $u^\star, v^\star \in V_{i+1}$, such that $d(u, u^\star) \leq \epsilon$, $d(v^\star, v) \leq \epsilon$ and $u^\star \to v^\star$. That is, $L_{out}(u^\star) \cap L_{in}(v^\star) \neq \emptyset$
iff there are $x \in \boldsymbol{\mathcal{B}}_{out}^{\epsilon}(u|G_i)$ and $y \in \boldsymbol{\mathcal{B}}_{in}^{\epsilon}(v|G_i)$, such that $x \to y$, i.e., $L_{out}(x) \cap L_{in}(y) \neq \emptyset$ (if there is $x \in V_{i+1}$, such that $d(u, x) \leq \epsilon$ and $d(x, u^\star) \leq \epsilon$, then we can always use x to replace $u^\star$ for the above claim; ($u^\star \to v^\star$ then $x \to v^\star$))
iff $(\bigcup_{u \in \boldsymbol{\mathcal{B}}_{out}^{\epsilon}(v|G_i)} L_{out}(u)) \cap (\bigcup_{u \in \boldsymbol{\mathcal{B}}_{in}^{\epsilon}(v|G_i)} L_{out}(u)) \neq \emptyset$.

Case 2 ($u \in V_i \setminus V_{i+1}$ and $v \in V_{i+1}$): We observe 1) $u \to v$ with $d(u, v) \leq \epsilon$ (local pair) iff either $v \in \boldsymbol{\mathcal{B}}_{out}^{\epsilon}(u|G_i)$ ($v \in L_{out}(u)$ and $v \in L_{in}(v)$), or there is $x \in \boldsymbol{\mathcal{B}}_{out}^{\epsilon}(v|G_i)$, such that $x \to v$, i.e. $L_{out}(x) \cap L_{in}(v) \neq \emptyset$
iff $(\bigcup_{u \in \boldsymbol{\mathcal{B}}_{out}^{\epsilon}(v|G_i)} L_{out}(u)) \cap (\bigcup_{u \in \boldsymbol{\mathcal{B}}_{in}^{\epsilon}(v|G_i)} L_{out}(u)) \neq \emptyset$; and 2) $u \to v$ with $d(u, v) > \epsilon$ (non-local pair) iff there exists x such that $x \in \boldsymbol{\mathcal{B}}_{out}^{\epsilon}(v|G_i)$ and $x \to v$, i.e. $L_{out}(x) \cap L_{in}(v) \neq \emptyset$
iff $(\bigcup_{u \in \boldsymbol{\mathcal{B}}_{out}^{\epsilon}(v|G_i)} L_{out}(u)) \cap (\bigcup_{u \in \boldsymbol{\mathcal{B}}_{in}^{\epsilon}(v|G_i)} L_{out}(u)) \neq \emptyset$.

Thus, in all cases, we have the correct labeling for any vertex pair u and v in V_{i+1}. Now, the core labeling is correct either based on the basic case where the graph diameter is no more than ϵ or based on the existing 2-hop labeling approaches [11, 23]. Together with the above induction rule, we have for any vertex pair in $V = V_0$, the label is complete and we thus prove the claim. □

Complexity Analysis: The computational complexity of Algorithm 1 comes from three components: 1) the hierarchical DAG decomposition, 2) the core graph labeling, and 3) the remaining vertex labeling for levels from $h - 1$ to 0. For the first component, as we mentioned earlier, we can employ the *FastCover* algorithm [19] iteratively to extract the reachability backbone vertices V_i and their corresponding graph G_i. The *FastCover* algorithm is very efficient and to extract G_{i+1} from G_i, it just needs to traverse the ϵ neighbors of each vertex in G_{i+1}. Its complexity is $O(\sum_{v \in V} |N_{out}^{\epsilon}(v|G_i)| log |N_{out}^{\epsilon}(v|G_i)| + |E_{out}^{\epsilon}(v|G_i)|)$, where $E_{out}^{\epsilon}(v|G_i)$ is the set of edges v can reach in ϵ steps. Also, we note that in practice, the vertex set V_i shrinks very quickly and after a few iterations (5 or 6 typically for $\epsilon = 2$), the number of backbone vertices is on the order of thousands (Section 6). We can also limit the total number of iterations, such as bounding h to be 10 and/or stop the decomposition when the V_i is smaller than some limit such as $10K$. For the second component, if the diameter is smaller than ϵ and Formula 3 is employed, it also has a linear cost: $O(\sum_{v \in V}(|N_{out}^{\epsilon}(v|G_h)| + |E_{out}^{\epsilon}(v|G_h)| + |N_{in}^{\epsilon}(v|G_h)| + |E_{in}^{\epsilon}(v|G_h)|))$. If we employ the existing 2-hop labeling approach [11, 23], the cost can be $O(|V_h|^4)$. However, since $|V_h|$ is rather small, the cost can be acceptable and in practice (Section 6), it is also quite efficient. Finally, the cost to assign labels for all the remaining vertices is linear to their neighborhood cardinality and the labeling size of each vertex. It can be written as $O(\sum_{v \in V_i \setminus V_{i+1}}(|N_{out}^{\epsilon}(v|G_h)| + |E_{out}^{\epsilon}(v|G_h)| + |N_{in}^{\epsilon}(v|G_h)| + |E_{in}^{\epsilon}(v|G_h)|) + ML$, where M is the maximal number of vertices in the backbone vertex set and L is the maximal number of vertices in any L_{in} or L_{out}.

We note that for large graphs, the last component typically dominates the total computational cost as we need to perform list merge (set-union) operations to generate L_{out} and L_{in} for each vertex. However, compared with the existing hop labeling approach, Hierarchical-Labeling is significantly cheaper as there is no need for materializing transitive closure and the set-cover algorithm. The experimental study (Section 6) finds that the labeling size produced by the Hierarchical-Labeling approach is comparable to that produced by the expensive set-cover based optimization.

5. DISTRIBUTION LABELING

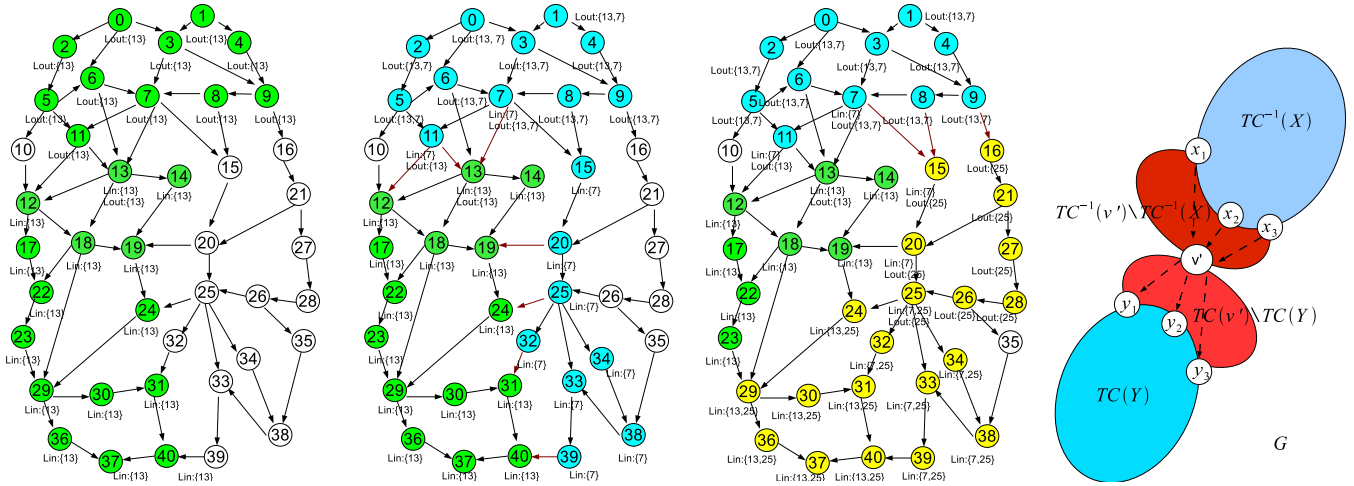
The Hierarchical-Labeling approach provides a fast alternative to produce a *complete* reachability oracle. Its labeling is dependent on a reachability-based hierarchical decomposition and follows a process similar to the classical transitive closure computation [26], where the transitive closure of all incoming neighbors are merged to produce the new transitive closure. However, the potential issue is that when merging L_{out} and L_{in} of higher level vertices for the lower level vertices, this approach does not (and cannot) check whether any hop is redundant, i.e., their removal can still produce a complete labeling. Given the current framework, it is hard to evaluate the importance of each individual hop as they being cascaded into lower level vertices. Recall that for a vertex v, when computing its $L_{out}(v)$ and $L_{in}(v)$, its corresponding backbone vertex sets ($\boldsymbol{\mathcal{B}}_{out}^{\epsilon}(v)$ and $\boldsymbol{\mathcal{B}}_{in}^{\epsilon}(v)$) only eliminate those redundant backbones if they can be linked through a local vertex (Formulas 1 and 2). Thus even if $u \in \boldsymbol{\mathcal{B}}_{out}^{\epsilon}(v)$, it may still be redundant as there is another vertex $u' \in \boldsymbol{\mathcal{B}}_{out}^{\epsilon}(v)$ such that $u' \to u$ (but $d(u', u)$ is large). However, this issue is related to the difficulty of computing transitive reduction as mentioned earlier.

In light of these issues, we ponder the following: Can we perform labeling without the recursive hierarchical decomposition? Can we explicitly confirm the "power" or "importance" of an individual hop as it is being added into L_{out} and L_{in}? In this work, we provide positive answers to these questions and along the way, we discover a simple, fast, and elegant labeling algorithm, referred to as *Distribution-Labeling*: 1) the recursive hierarchical decomposition is replaced with a simple total order of vertices (the order criterion can be as simple as a basic function of vertex degree); 2) each hop is explicitly verified to be added into L_{out} and L_{in} only when it can cover some additional reachable pairs, i.e., it is non-redundant. Surprisingly, the labeling size produced by this approach is even smaller than the set-cover approach on all the available benchmarking graphs used in the recent reachability studies (Section 6).

In Subsection 5.1, we first introduce a simple yet fundamental observation of *hop-covering* (given a hop, what vertex pairs can it cover), which is the basis for the *Distribution-Labeling* algorithm; and Subsection 5.2, we present the labeling algorithm and discuss its properties.

5.1 Hop Coverage and Labeling Basis

We first formally define the "covering power" of a hop and then study the relationship of two vertices in terms of their "covering power".



(a) Labeling for $Cov(13)$ (b) Labeling for $Cov(\{13,7\})$ (c) Labeling for $Cov(\{13,7,25\})$ (d) Basic Labeling

Figure 2: Running Example of Distribution-Labeling

DEFINITION 3. *(**Hop Coverage**) For vertex v, its* **coverage** *$Cov(v)$ is defined as $TC^{-1}(v)\times TC(v)=\{(u,w): u\to v$ and $v\to w\}$. Note that $TC^{-1}(v)$ is the reverse transitive closure of v which includes all the vertices reaching v. If for any pair in $(u,w)\in Cov(v)$, $L_{out}(u)\cap L_{in}(w)\neq\emptyset$, then we say $Cov(v)$ is covered by the labeling. We also say $Cov(v)$ can be covered by v if each vertex u reaching v ($u\in TC^{-1}(v)$) has $v\in L_{out}(u)$ and each vertex w being reached by v has $v\in L_{in}(w)$ ($w\in TC(v)$).*

Given this, the labeling L_{out} and L_{in} is complete if it covers $Cov(V)=\cup_{v\in V}Cov(v)$, i.e., for any $(u,w)\in Cov(V)$,
To achieve a complete labeling, let us start with $Cov(v,v')=Cov(v)\cup Cov(v')$. We study how to use only v and v' to cover $Cov(v,v')$. Specifically, we consider the following question: *assuming v has been recorded by $L_{out}(u)$ for every $u\in TC^{-1}(v)$ and by $L_{in}(w)$ for every $w\in TC(v)$, then in order to cover the reachability pairs in $Cov(v,v')$ and only v' can serve as the hop, what vertices should record v' in their L_{out} and L_{in}?*

To answer this question, we consider three cases: 1) v and v' are incomparable, i.e., $v\nrightarrow v'$ and $v\nleftarrow v'$; 2)$v'\to v$; and 3) $v\to v'$. For the first case, the labeling is straightforward: each $u\in TC^{-1}(v')$ needs to record $v'\in L_{out}(u)$ and each $w\in TC(v')$ needs to record $v'\in L_{in}(u)$. Note that in the worst case, this is needed in order to recover pairs as $TC^{-1}(v')\times\{v'\}$ and $\{v'\}\times TC(v')$. For Cases 2 and 3, Lemma 2 provides the answer.

LEMMA 2. *Let $L_{in}(u)=\{v\}$ for every $u\in TC^{-1}(v)$ and $L_{out}(w)=\{v\}$ for every $w\in TC(v)$. If $v'\to v$, then with $L_{out}(u)=\{v,v'\}$ for $u\in TC^{-1}(v')$ and $L_{in}(w)=\{v'\}$ for $w\in TC(v')\backslash TC(v)$ (other labels remain the same), $Cov(\{v,v'\})$ is covered (using only hops v and v'). If $v\to v'$, then with $L_{out}(u)=\{v'\}$ for $u\in TC^{-1}(v')\setminus TC^{-1}(v)$ and $L_{in}(w)=\{v,v'\}$ for $w\in TC(v')$ (other labels remain the same), $Cov(\{v,v'\})$ is covered (using only hops v and v').*

Proof Sketch: We will focus on proving the case where $v'\to v$ as the case $v'\to v$ is symmetric. We first note that if $v'\to v$, then $TC^{-1}(v)\subseteq TC^{-1}(v')$ and $TC(v')\supseteq TC(v)$. Since $Cov(v)=TC{-}1(v)\times TC(v)$ is already covered by v, the uncovered pairs in $Cov(\{v,v'\})$ can be written as

$$Cov(\{v,v'\})\setminus Cov(v)=TC^{-1}(v')\times(TC(v')\setminus TC(v))$$

Given this, adding v' to $L_{out}(u)$ where $u\in TC^{-1}(v')$ and to $L_{in}(w)$ where $w\in TC(v')\setminus TC(v)$ can thus cover all the pairs in $Cov(\{v,v'\})$. □

EXAMPLE 5.1. *Figure 2(a) shows the labeling for $Cov(13)$ and Figure 2(b) shows that for $Cov(13,7)$ where $7\to 13$. In particular, $TC^{-1}(13)=TC^{-1}(7)\cup\{11\}$ and $TC(13)\subset TC(7)$. For all $u\in TC^{-1}(7)$, we have $L_{out}(u)=\{7,13\}$ and for all $w\in L_{in}(7)\setminus L_{in}(13)$, we have $L_{in}(w)=\{7\}$.*

Given Lemma 2, we consider the following general scenario: for a subset of hops $V_s\subset V$, assume L_{out} and L_{in} are correctly labeled using only hops in V_s to cover $Cov(V_s)$. Now how can we cover $Cov(V_s\cup\{v'\})$ by adding the only additional hop v to L_{in} and L_{out}? The following theorem provides the answer (Lemma 2 can be considered a special case):

THEOREM 2. *(**Basic Labeling**) Given a subset of hops $V_s\subset V$, let $L_{out}(u)\subseteq V_s$ and $L_{in}(u)\subseteq V_s$ be complete for covering $Cov(V_s)$, i.e., for any $(u,v)\in Cov(V_s)$, $L_{out}(u)\cap L_{in}(v)\neq\emptyset$. To cover $Cov(V_s\cup\{v'\})$ using additional hop v', the following labeling is complete:*

$$L_{out}(u)\leftarrow L_{out}(u)\cup\{v'\}, u\in TC^{-1}(v')\setminus TC^{-1}(X) \quad (6)$$
$$L_{in}(w)\leftarrow L_{in}(w)\cup\{v'\}, w\in TC(v')\setminus TC(Y) \quad (7)$$

where $X=TC^{-1}(v')\cap V_s$ including all the vertices in V_s reaching v' and $Y=TC(v')\cap V_s$ including all the vertices in V_s that can be reached by v'; $TC^{-1}(X)=\bigcup_{v\in X}TC^{-1}(v)$ and $TC(Y)=\bigcup_{v\in Y}TC(v)$.

The theorem and its proof can be illustrated in Figure 2(d).

Proof Sketch: We first observe the following relationships between the (reverse) transitive closure of v' and X,Y.

$$TC^{-1}(v')\supseteq TC^{-1}(X);\qquad TC(v')\subseteq TC(v), v\in X;$$
$$TC(v')\supseteq TC(Y);\qquad TC^{-1}(v')\subseteq TC^{-1}(v), v\in Y;$$

Thus, following the similar proof of Lemma 2, we can see that

$$Cov(V_s\cup\{v'\})=Cov(V_s)\cup TC^{-1}(v')\times TC(v')$$
$$=Cov(V_s)\cup(TC^{-1}(v')\setminus TC^{-1}(X))\cup TC^{-1}(X))$$
$$\times((TC(v')\setminus TC(Y))\cup TC(Y))$$

$$= Cov(V_s) \cup (TC^{-1}(v') \setminus TC^{-1}(X)) \times (TC(v') \setminus TC(Y)) \cup (TC^{-1}(v') \setminus TC^{-1}(X)) \times \bigcup_{v \in Y} TC(v') \cup TC^{-1}(X) \times (TC(v') \setminus TC(Y)) \cup TC^{-1}(X) \times TC(Y)$$

$$= Cov(V_s) \cup (TC^{-1}(v') \setminus TC^{-1}(X)) \times (TC(v') \setminus TC(Y)),$$
since $(TC^{-1}(v') \setminus TC^{-1}(X)) \times TC(Y) \subseteq Cov(V_s)$;
$TC^{-1}(X) \times (TC(v') \setminus TC(Y)) \subseteq Cov(V_s)$;
$TC^{-1}(X) \times TC(Y) \subseteq Cov(V_s)$

Thus, by adding v' to $L_{out}(u), u \in TC^{-1}(v') \setminus TC^{-1}(X)$ and to $L_{in}(w), w \in TC(v') \setminus TC(Y)$, the labeling will be complete to cover $Cov(V_s \cup \{v'\})$. □

EXAMPLE 5.2. *Figure 2(c) shows an example of* $Cov(\{13, 7\} \cup \{25\})$*, where* $X = \{13, 7\}$ *(both can reach* 25 *and* $Y = \emptyset$*. Thus* 25 *is added to* $L_{out}(u), u \in TC^{-1}(25) \setminus (TC(13) \cup TC(7))$ *and to* $L_{in}(w), w \in TC(25)$.

5.2 Distribution-Labeling Algorithm

In the following, base on Lemma 2 and Theorem 2, we introduce the *Distribution-Labeling* algorithm, which will iteratively distribute each vertex v to L_{out} and L_{in} of other vertices to cover $Cov(V_s \cup \{v\})$ (V_s includes processed vertices). Intuitively, it first selects a vertex v_1 and provides complete labeling for $Cov(v_1)$; then it selects the next vertex v_2, provides complete labeling for $Cov(\{v_1, v_2\})$ based on Lemma 2. It continues this process, at each iteration i selecting a new vertex v_i and producing the complete labeling for $Cov(V_s \cup \{v_i\})$ based on Theorem 2 where V_s includes all the $i-1$ vertices which have been processed. The complete labeling will be produced when $V_s = V$.

Given this, two issues needs to be resolved for this labeling process: 1) What should be the order in selecting vertices, and 2) How can we quickly compute X (processed vertices which can reach the current vertex v_i) and Y (processed vertices v_i can reach), and identify $u \in TC^{-1}(v_i) \setminus TC^{-1}(X)$ and $w \in TC(v_i) \setminus TC(Y)$.

Vertex Order: The vertex order can be considered an extreme hierarchical decomposition, where each level contains only one vertex. Furthermore, the higher level the vertex, then the more important it is, the earlier it will be selected for covering, and the more vertices that are likely to record it in their L_{out} and L_{in} lists. There are many approaches for determining the vertex order. For instance, if following the set-cover framework, the vertex can be dynamically selected to be the *cheapest* in covering new pairs, i.e., $\frac{|TC^{-1}(v_i) \setminus TC^{-1}(X)| + |TC(v_i) \setminus TC(Y)|}{|Cov(V_s \cup \{v_i\}) \setminus Cov(V_s)|}$. However, this is computationally expensive. We may also use $|Cov(v_i)|$ which measures the covering power of vertex v, but this still needs to compute transitive closure. In this study, we found the following rank function, $(|N_{out}(v)|+1) \times (|N_{in}(v)|+1)$, which measures the vertex pairs with distance no more than 2 being covered by v, is a good candidate and can provides compact labeling. Indeed, we note a similar criterion actually used in [19] for selecting reachability backbone as well.

Labeling L_{out} and L_{in}: Given vertex v_i, we need to find (1) $u \in TC^{-1}(v_i) \setminus TC^{-1}(X)$, i.e., the vertices reaching v_i but not reaching by v such that $v \to v_i$ and it has a higher order (already being processed); and (2) $w \in TC(v_i) \setminus TC(Y)$, i.e., the vertices which can be reached by v_i but cannot be reached by v such that $v_i \to v$ and it has a higher order. The straightforward way for solving (1) is to perform a reversed traversal and visit (expand) the vertices based on the reversed topological order; then once the visited vertex has a higher order then v_i, all its descendents (including itself) will be colored (flagged) to be be excluded from adding v_i to L_{out}; thus v_i will be added to L_{out} for all uncolored vertices during the reverse traveral process. A similar ordered traversal process can be used for solving (2). However, the (reverse) ordered traversal needs a priority queue which results in $O(|V| \log |V| + |E|)$ complexity at each iteration. In this work, we utilize a more efficient approach can effectively prune the traversal space and avoid the priority queue, which is illustrated in Algorithm 2.

Algorithm 2 Distribution-Labeling(G=(V,E))

```
1:  Rank vertices in G in certain order;
2:  for each v_i ∈ V {from higher order to lower} do
3:      Perform Reverse BFS starting from v_i, and for each vertex
        u being visited:
4:      if L_out(u) ∩ L_in(v_i) ≠ ∅ then
5:          Do not add v_i to L_out(u) nor expand u;
6:      else
7:          Add v_i into L_out(u) and expand u in the reverse BFS;
8:      end if
9:      Perform BFS starting from v_i, and for each vertex w being
        visited:
10:     if L_in(w) ∩ L_out(v_i) ≠ ∅ then
11:         Do not add v_i to L_in(u) nor expand w;
12:     else
13:         Add v_i into L_in(w) and expand w in the BFS;
14:     end if
15: end for
```

In Algorithm 2, the iteration labeling process is sketched in the foreach loop (Lines 2 to 15). The main procedure in computing $u \in TC^{-1}(v_i) \setminus TC^{-1}(X)$ for labeling L_{out} is outlined in Lines $3-8$. The main idea is that when visiting a vertex u, once $L_{out}(u) \cap L_{in}(v_i)$ is no longer empty, we can simply exclude u and its descendents from consideration, i.e., $u \in TC^{-1}(X)$ (Lines $4-6$). Intuitively, this is because there exists a vertex v, such that $u \to v \to v_i$ and has order higher than v_i. Similarly, the procedure that computes $w \in TC(v_i) \setminus TC(Y)$ for labeling L_{in} is outlined in Lines $9-14$. Here, the condition $L_{in}(w) \cap L_{out}(v_i) \neq \emptyset$ is utilized to prune w and its descendents to determine L_{in} labeling. Figure 2 illustrates the labeling process based on Algorithm 2 for the first three vertices 13, 7, and 25.

5.3 Completeness, Compactness, and Complexity

In the following, we discuss the labeling completeness (correctness), compactness (non-redundancy), and time complexity.

THEOREM 3. *(***Completenss***) The* Distribution-Labeling *algorithm (Algorithm 2) produces a complete* L_{out} *and* L_{in} *labeling, i.e., for any vertex pair* (u, v), $u \to v$ *iff* $L_{out}(u) \cap L_{in}(v) \neq \emptyset$.

Proof Sketch: $u \in TC^{-1}(v_i) \setminus TC^{-1}(X)$ and 2) $w \in TC(v_i) \setminus TC(Y)$. They are symmetric and we will focus on 1). Note that for $u \in TC^{-1}(v_i) \setminus TC^{-1}(X)$, we need to exclude vertex u' such that $u' \to v \to v_i$, where v is already processed (has higher order than v_i). Assuming the labeling is complete for $Cov(V_s)$, where $V_s = \{v_1, \cdots, v_{i-1}\}$, then $L_{out}(u') \cap L_{in}(v_i) \neq \emptyset$ (Line 4). If u' should be excluded, then its descendents from the BFS traversal will also be true and should also be excluded. Furthermore, the reverse BFS can visit all vertices where this condition does not hold, i.e., $L_{out}(u) \cap L_{out}(v_i) = \emptyset$, and thus $u \in TC^{-1}(v_i) \setminus TC^{-1}(X)$. □

Theorem 3 shows that the Distribution-Labeling algorithm is correct; but how compact is the labeling? The following theorem shows an interesting *non-redundant* property of the produced labeling, i.e., no hop can be removed from L_{in} or L_{out} while preserving completeness. We note that this property has not been investigated before in the existing studies on reachability oracle and hop labeling [11, 23, 8, 9, 16, 4].

THEOREM 4. *(**Non-Redundancy**) The* Distribution-Labeling *algorithm (Algorithm 2) produces a non-redundant L_{out} and L_{in} labeling, i.e., if any hop h is removed from a L_{out} or L_{in} label set, then the labeling becomes incomplete.*

Proof Sketch: We will show that 1) for any $u \in TC^{-1}(v_i) \setminus TC^{-1}(X)$, v_i cannot be removed from L_{out}; and 2) for any $w \in TC(v_i) \setminus TC(Y)$, v_i cannot be removed from L_{in}. Note that when v_i is being added to $L_{out}(u)$ and $L_{in}(w)$, it is non-redundant as the new labeling at least covers $(TC^{-1}(v_i) \setminus TC^{-1}(X)) \times \{v_i\}$ and $\{v_i\} \times TC(v_i) \setminus TC(Y)$.

However, will any later processed vertex v_j, such that $i < j$, make v_i redundant? The answer is no because in this case (still focusing on the above covered pairs by v_i), $u \rightarrow v_j \rightarrow v_i$ (or $w \leftarrow v_j \leftarrow v_i$), but the order of v_i is higher than v_j and v_j will not be added v_i into its L_{out} or L_{in}. In other words, for any vertex pair in $(TC^{-1}(v_i) \setminus TC^{-1}(X)) \times \{v_i\}$ or $\{v_i\} \times TC(v_i) \setminus TC(Y)$, v_i is the only hop linking these pairs, i.e., $L_{out}(u) \cap L_{in}(v_i) = \{v_i\}$ and $L_{out}(v_i) \cap L_{out}(u) = \{v_i\}$. Thus, v_i is non-redundant for all the vertices recording it as label, i.e., $L_{out}(u), u \in TC^{-1}(v_i) \setminus TC^{-1}(X)$ and $L_{in}(w), w \in TC(v_i) \setminus TC(Y)$. □

As we discussed earlier, Hierarchical-Labeling does not have this property; we can see this through counter-examples. For instance, in Figure 1(b), 17 is redundant for $L_{out}(5)$. However, to remove these cases, the transitive reduction would have to be performed, which is expensive. Furthermore, whether the labels produced by the existing set-cover based approach [11] are redundant or not remains an open question though we conjecture they might be redundant.

Time Complexity: The worst case computational complexity of Algorithm 2 can be written as $O(|V|(|V|+|E|)L)$, where L is the maximal labeling size. However, the conditions in Line 4 and 10 can significantly prune the search space, and L is typically rather small, the *Distribution-Labeling* can perform labeling very efficiently. In the experimental study (Section 6), we will show Algorithm 2 is on average more than an order of magnitude faster than the existing hop labeling and has comparable or faster labeling time than the state-of-the-art reachability indexing approaches on large graphs. Its labeling size is also small and surprisingly, even smaller than the greedy set-cover based labeling approaches in most of the cases. This may be an evidence that the labeling of the existing set-cover based approach [11] is redundant.

6. EXPERIMENTAL EVALUATION

In this section, we empirically evaluate the *Hierarchical-Labeling* and *Distribution-Labeling* labeling algorithms against the state-of-the-art reachability computation approaches on a range of real graphs which have been widely used for studying reachability [31, 29, 15, 19, 10]. In particular, we are interested in the following questions (in terms of the query efficiency, construction cost, and index (labeling) size) : 1) how do the reachability oracle approaches perform compared with the transitive closure compression and online search approaches? 2) how do these two approaches perform compared with the existing 2-hop approaches assuming the later one can complete the labeling? 3) How do these two methods (Hierarchical-Labeling and Distribution-Labeling) compare with one another?

Small Real Graph			Large Real Graph		
Dataset	$\|V\|$	$\|E\|$	Dataset	$\|V\|$	$\|E\|$
agrocyc	12684	13408	citeseer	693,947	312,282
amaze	3710	3600	go_uniprot	6,967,956	34,770,235
anthra	12499	13104	mapped_100K	2,658,702	2,660,628
ecoo	12620	13350	mapped_1M	9,387,448	9,440,404
hpycyc	4771	5859	uniprotenc_22m	1,595,443	1,595,442
human	38811	39576	uniprotenc_100m	16,087,294	16,087,293
kegg	3617	3908	uniprotenc_150m	25,037,599	25,037,598
mtbrv	9602	10245	citeseerx	6,540,399	15,011,259
nasa	5605	7735	cit-Patents	3,774,768	16,518,947
reactome	901	846			
vchocyc	9491	10143			
xmark	6080	7028			

Table 1: Real datasets

6.1 Experimental Setup

To answer these questions, we evaluate the *Hierarchical-Labeling (HL)* and *Distribution-Labeling (DL)* labeling algorithms against the state-of-the-art reachability computation approaches:

1) *PathTree* [17], an improved version of Agrawal's tree-interval method [2];

2) *Nuutila's Interval* [21], a transitive closure compression method, recently demonstrated to be one of the fastest reachability computation methods [29];

3) *PAWH-8* [29], the latest bit-vector compression method for transitive closure compression) [29] and *PWAH-8* is its best variant [29].

4) *K-Reach* [10], a latest vertex-cover based approach for general reachability computation, i.e., determine whether two vertices are within distance k. Here k is set to be the total number of vertices in the graph for the basic reachability.

5) *GRAIL* [31], a scalable reachability indexing approach using random DFS labeling (the number of intervals is set at 5, as suggested by authors).

6) *2HOP* [11], Cohen *et al.*'s 2-hop labeling approach;

Here, Path-Tree (1), Interval (2), and PAWH-8(3) are the state-of-the-art transitive closure compression approaches; K-Reach (4) is the latest general reachability approach and has been shown to be very capable in dealing with basic reachability [10] (it can also be considered as transitive closure compression as it materializes the transitive closure for the vertex-cover, a subset of vertices); GRAIL (5) is the state-of-the-art online search approach; and 2HOP (6) is the existing set-cover based hop labeling approach. In addition, we also include the latest *SCARAB* method [19] for scaling PathTree and speeding up GRAIL, referred to as *PATH-TREE** and *GRAIL**, respectively. The locality parameter ϵ is set at 2 for SCARAB.

All the methods (including source code) except 2HOP are either downloaded from authors' websites or provided by the authors directly. We have implemented 2HOP, Hierarchical-Labeling (HL), Distribution-Labeling (DL), and 2HOP has been improved with several fast heuristics [23, 16] to speed up its construction time. All these algorithms are implemented in C++ based on the Standard Template Library (STL).

In the experiments, we focus on reporting the three key measures for reachability computation: query time, construction time, and index size. For the query time, similar to the latest SCARAB work [19], both *equal* and *random* reachability query workload are used. The equal query workload has about 50% positive (reachable pairs) and about 50% negative (unreachable pairs) queries. Positive queries are generated by sampling the transitive closure. Also the query time is the running time of a total of $100,000$ reachability queries.

All experiments are performed on a Linux 2.6.32 machine with Intel Xeon 2.67GHz CPU and 32GB RAM.

6.2 Experimental Results

Dataset	GRAIL	GRAIL*	PATH-TREE	PATH-TREE*	K-REACH	PWAH-8	INTERVAL	2HOP	HL	DL
agrocyc	189.84	55.01	1.11	6.20	1.54	7.86	2.59	3.78	4.30	2.11
amaze	343.47	18.43	1.16	12.64	1.44	3.55	3.08	3.03	2.95	2.22
anthra	124.24	43.86	1.28	6.51	1.48	7.74	2.58	3.79	3.93	2.11
ecoo	122.11	55.45	1.10	6.29	1.50	7.69	2.71	3.86	4.41	2.14
hpycyc	87.76	15.30	1.06	12.00	1.45	8.62	1.47	3.83	4.03	2.29
human	185.38	68.54	1.16	6.84	1.78	4.45	3.23	3.61	2.50	2.30
kegg	272.12	26.90	1.19	13.10	1.54	4.75	2.59	3.18	3.44	2.38
mtbrv	115.41	49.22	1.06	6.44	1.47	7.20	2.59	3.95	5.14	2.10
nasa	135.78	32.41	1.37	14.44	2.16	18.41	4.87	4.45	4.07	3.72
reactome	111.75	15.74	1.12	9.66	1.81	12.10	3.01	3.10	2.87	2.18
vchocyc	107.65	44.89	1.04	6.50	1.45	7.86	2.58	3.75	3.81	2.09
xmark	134.72	91.72	1.42	14.55	1.93	35.77	4.89	5.94	6.52	3.79

Table 2: Query Time (ms) of $100K$ Equal Queries on Small Real Datasets

Dataset	GRAIL	GRAIL*	PATH-TREE	PATH-TREE*	K-REACH	PWAH-8	INTERVAL	2HOP	HL	DL
agrocyc	29.04	3.64	1.40	4.47	1.17	2.58	2.20	4.22	4.45	3.40
amaze	501.99	12.75	1.83	9.72	2.31	3.67	4.41	3.96	4.12	3.77
anthra	29.57	3.49	1.39	4.36	1.22	1.30	2.17	4.15	4.37	3.37
ecoo	30.37	3.67	1.32	4.45	1.45	2.52	2.19	4.15	4.46	3.41
hpycyc	28.82	3.17	1.31	4.21	2.00	3.07	2.33	3.81	3.95	3.62
human	33.61	3.97	1.48	5.77	1.22	1.46	1.41	2.67	4.81	3.85
kegg	616.65	3.60	1.88	4.41	2.58	4.52	4.63	4.11	4.35	3.98
mtbrv	28.90	8.29	1.33	8.08	1.11	1.37	2.17	3.90	4.07	3.34
nasa	28.20	5.03	1.66	4.99	2.74	7.99	5.33	4.82	5.26	5.40
reactome	32.92	3.52	2.97	4.37	3.00	7.86	3.40	3.52	3.37	3.70
vchocyc	29.62	8.85	1.36	6.54	1.42	2.61	2.24	4.04	3.79	3.38
xmark	63.73	17.16	1.71	10.88	2.27	11.16	4.47	5.48	5.36	5.44

Table 3: Query Time (ms) of $100K$ Random Queries on Small Real Datasets

Dataset	GRAIL	GRAIL*	PATH-TREE	PATH-TREE*	K-REACH	PWAH-8	INTERVAL	2HOP	HL	DL
agrocyc	22.62	27.20	128.10	68.91	284.50	5.01	3.72	245.60	120.75	12.62
amaze	7.35	10.38	357.40	27.84	330.37	4.47	3.21	2672.16	43.16	4.14
anthra	14.11	24.97	88.20	64.16	246.33	4.14	2.90	241.05	89.44	12.43
ecoo	12.76	26.01	94.54	66.88	282.14	4.98	3.67	254.64	92.16	12.49
hpycyc	4.75	12.96	39.02	23.25	223.63	2.74	1.84	199.20	41.48	5.24
human	71.24	72.29	298.24	143.08	296.47	5.30	4.12	417.55	155.19	37.38
kegg	4.08	26.38	435.99	44.85	411.80	5.56	2.24	2877.98	48.33	2.35
mtbrv	9.38	17.82	71.72	46.98	249.31	2.19	2.99	208.31	115.31	9.80
nasa	10.18	7.72	49.37	26.58	1637.47	9.83	6.06	835.88	143.46	8.88
reactome	1.21	20.17	8.25	26.43	35.15	1.23	0.78	80.68	25.38	1.02
vchocyc	9.32	16.67	70.29	45.86	260.10	4.39	3.22	224.31	65.30	9.53
xmark	11.07	30.24	109.30	50.49	806.25	10.43	5.89	1557.01	53.23	8.70

Table 4: Construction Time (ms) of Small Real Datasets

In the following, we report the experimental results on small graphs first and then on large graphs. These graphs have been widely used for studying reachability computation [30, 8, 17, 16, 33, 31, 4, 29, 15, 10, 19]. In Table 1, the first three columns give the names, number of vertices and number of edges for the coalesced DAGs derived from each original graph. The last three columns give similar information for large real graphs.

Small Graphs: Table 2 reports the query times of the reachability oracle approaches (2HOP, Hierarchical-Labeling (HL), and Distribution-Labeling (DL)) against the state-of-the-art transitive closure compression approaches (PWAH-8, INTERVAL, PATH-TREE, K-REACH), and online search (GRAIL), as well as some of their SCARAB counterparts, including GRAIL* and PATH-TREE* using the *equal* query load. Table 3 reports the query time using the *random* query load.

We make the following important observations on the query time:
1) On small graphs, PATH-TREE outperforms other methods, though K-REACH is fairly close (as it is quite similar to the transitive closure materialization). Interestingly, the reachability oracle methods turn out to be quite comparable. In particular, the Distribution-Labeling (DL) is consistently about 2 times slower than PATH-TREE, and even faster than the other transitive closure compression approaches, INTERVAL and PWAH-8, on equal query load.
2) Compared to the existing set-cover based labeling approach 2HOP, Hierarchical-Labeling (HL) is quite comparable (slightly slower), but the query time of Distribution-Labeling (DL) is only $2/3$ of that of 2HOP.
3) The reachability oracle approaches are slightly slower on the random query load than on the equal query load. This is because to determine vertex u cannot reach vertex v, the query processing has to completely scan $L_{out}(u)$ and $L_{in}(v)$.

Table 4 shows the construction time of different reachability indices on small graphs. We observe K-REACH and 2HOP are the slowest. This is understandable as K-REACH needs to perform vertex-cover discovery and materialize the transitive closure for the vertex-cover; and 2HOP needs to perform the expensive greedy set-cover and completely materialize the transitive closure. INTERVAL and PAWH-8 turn out be the fastest and even faster than the online search GRAIL approach as the later still needs to perform random DFS a few times (in this study, we choose the number

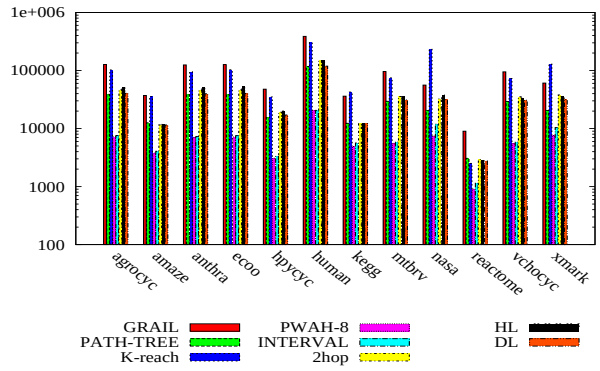


Figure 3: Index Size on Small Real Graphs (in terms of the number of integers used in the indices)

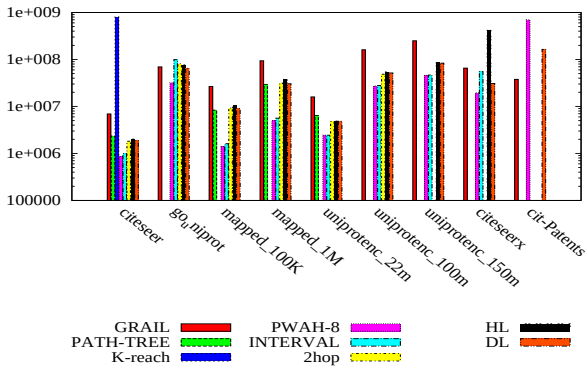


Figure 4: Index Size on Large Real Graphs (in terms of the number of integers used in the indices)

Dataset	GRAIL	GRAIL*	PATH-TREE	PATH-TREE*	K-REACH	PWAH-8	INTERVAL	2HOP	HL	DL
citeseer	63.40	18.60	4.93	18.40	9.29	20.63	12.28	4.51	7.71	3.70
go_uniprot	77.61	23.26	—	20.37	—	41.93	17.02	16.04	6.21	12.82
mapped_100K	253.82	32.98	6.72	16.05	—	90.61	6.04	5.06	5.15	6.26
mapped_1M	762.24	30.31	8.40	16.87	—	46.59	6.31	5.59	6.07	7.28
uniprotenc_22m	52.99	9.26	6.19	14.70	—	23.13	15.50	47598.40	4.42	5.07
uniprotenc_100m	82.69	15.43	—	22.92	—	29.82	19.83	—	5.67	5.40
uniprotenc_150m	79.15	18.11	—	30.18	—	31.06	20.35	—	6.52	5.80
citeseerx	2012.33	2358.31	—	—	—	76.33	8.76	—	210.19	5.99
cit-Patents	403.91	141.02	—	—	—	2538.92	—	—	—	35.01

Table 5: Query Time (ms) of $100K$ Equal Queries on Large Real Datasets

Dataset	GRAIL	GRAIL*	PATH-TREE	PATH-TREE*	K-REACH	PWAH-8	INTERVAL	2HOP	HL	DL
citeseer	40.16	6.14	4.37	8.40	5.21	12.39	9.64	6.97	4.70	4.71
go_uniprot	47.63	9.03	—	13.57	—	52.52	20.76	13.02	12.02	11.64
mapped_100K	52.40	4.99	5.88	7.37	—	4.94	4.99	6.47	6.68	8.81
mapped_1M	55.00	5.33	8.75	9.49	—	5.60	6.73	7.08	9.78	9.33
uniprotenc_22m	40.54	8.65	9.11	12.50	—	21.87	15.19	4.43	5.88	7.04
uniprotenc_100m	53.01	10.35	—	17.55	—	28.30	20.07	—	7.47	7.40
uniprotenc_150m	56.63	10.66	—	18.83	—	29.14	23.12	—	10.65	7.86
citeseerx	2585.63	94.86	—	—	—	39.82	13.39	—	23.70	8.78
cit-Patents	501.53	110.13	—	—	—	1766.25	—	—	—	20.49

Table 6: Query Time (ms) of $100K$ Random Queries on Large Real Datasets

Dataset	GRAIL	GRAIL*	PATH-TREE	PATH-TREE*	K-REACH	PWAH-8	INTERVAL	2HOP	HL	DL
citeseer	2,011	1,250	18,025	2,192	187,182	487	307	14,054	2,232	528
go_uniprot	32,358	23,454	—	5,058,641	—	34,373	20,664	252,540	279,132	16,706
mapped_100K	6,220	5,362	26,667	9,775	—	448	419	9,760	10,141	1,902
mapped_1M	28,303	22,147	103,265	42,475	—	2,399	3,777	52,190	45,490	6,894
uniprotenc_22m	5,034	2,830	9,801,660	43,734	—	1,408	1,064	102,679	5,209	1,004
uniprotenc_100m	66,285	43,050	—	8,206,130	—	16,330	11,624	—	67,270	13,854
uniprotenc_150m	101,556	69,032	—	18,900,437	—	27,202	18,863	—	119,570	21,015
citeseerx	17,564	21,206	—	—	—	17,006	7,015	—	182,068	9,909
cit-Patents	15,669	42,175	—	—	—	935,457	—	—	—	114,583

Table 7: Construction Time (ms) of Large Real Datasets

to be 5 as being used in [31]). Both Hierarchical-Labeling (HL) and Distribution-Labeling (DL) are much more efficient in labeling: The Hierarchical-Labeling is on average 5 times faster than 2HOP whereas the Distribution-Labeling is consistently 20 times faster (and in some case more than two order of magnitude faster) than 2HOP. In fact, it has even faster construction time than GRAIL and quite comparable to the INTERVAL and PWAH-8.

Figure 3 shows the index size of different reachability index methods along with some of their SCARAB counterparts on small graphs. Here, PWAH-8 and INTERVAL outperform the others on index size. It is interestingly to observe that the labeling size of Hierarchical-Labeling (HL) is quite comparable to 2HOP (and this is also consistent with the query time). More importantly and rather surprisingly, the labeling size of Distribution-Labeling (DL) is consistently smaller than that of 2HOP, the set-cover based optimization labeling targeting for minimizing the labeling size. This, we believe, can be attributed to the effectiveness of the total order based hierarchy and the non-redundant labeling process.

Large Graphs: Large graphs provide the real challenge for the reachability computation. We observe that only three methods, GRAIL, PWAH-8, and Distribution-Labeling are able to handle all these graphs (GRAIL* is the SCARAB variant for speeding up query performance). Distribution-Labeling and INTERVAL can work on 8, and PATH-TREE* can work on 7, out of 9 large graphs. K-REACH can only perform one graph, where PATH-TREE and 2HOP fail on 5 and 4 large graphs, respectively.

Tables 5 and 6 report the query time using the *equal* and *random* query load, respectively. We make the following observations: 1) On large graphs, the transitive closure compression approaches, even on the graphs they can work, become significant slower. This is expected as the compressed transitive closure $TC(v)$ becomes larger, its search (linear or binary) becomes more expensive. Now, the advantage of the reachability oracle becomes clear as they become the fastest in terms of query time (even faster than PATH-TREE and INTERVAL, and consistently more than 5 times faster than PWAH-8). 2) Compared with the original 2HOP labeling, both Hierarchical-Labeling and Distribution-Labeling have comparable query performance on the graphs which they all can run.

Tables 7 shows the construction time on large graphs for all methods. We observe that PAWH-8 and INTERVAL are very fast though as the graph becomes larger, they become slower or cannot finish. Distribution-Labeling turns out to be quite comparable (fastest on several graphs). Hierarchical-Labeling can work on 8 out of 9 graphs and it shows signifiant improvement on 2 out of 5 graphs which 2HOP can also process. Distribution-Labeling is on average of one order of magnitude performance faster than 2HOP on these five graphs.

Figure 4 shows the index size of different approaches. The results are quite consistent with the results on the small graphs on those graphs they can work. For most cases, PWAH-8 And INTERVAL have the smallest index size. 2HOP, Hierarchical-Labeling and Distribution-Labeling also perform well (better than GRAIL and K-Reach). The labeling sizes of 2HOP, Hierarchical-Labeling and Distribution-Labeling are quite comparable; Distribution-Labeling has smaller labeling size than Hierarchical Labeling and very close to (or better than) 2HOP on the graphs it can run.

7. CONCLUSION

In this paper, by introducing two simple, elegant, and effective labeling approaches, *Hierarchical Labeling* and *Distribution Labeling*, we are able to resolve an important open question in reachability computation: the reachability oracle can be a powerful tool (or even the most useful one) to handle real large graphs. Our experimental results demonstrate that they can perform on graphs with millions of vertices/edges (scalable), are quickest in answering reachability queries on large graphs (fast), and have comparable or better labeling size as the set-cover based optimization approaches (compact). In the future, we will investigate the labeling on dynamic graphs and how to apply them on more general reachability computation, such as k-reach problem.

8. REFERENCES

[1] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. A hub-based labeling algorithm for shortest paths in road networks. In *Proceedings of the 10th international conference on Experimental algorithms*, 2011.

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

[3] R. Bauer, D. Delling, P. Sanders, D. Schieferdecker, D. Schultes, and D. Wagner. Combining hierarchical and goal-directed speed-up techniques for dijkstra's algorithm. *J. Exp. Algorithmics*, 15, March 2010.

[4] J. Cai and C. K. Poon. Path-hop: efficiently indexing large graphs for reachability queries. In *CIKM '10*, 2010.

[5] L. Chen, A. Gupta, and M. E. Kurul. Stack-based algorithms for pattern matching on dags. In *VLDB '05*, pages 493–504, 2005.

[6] Y. Chen and Y. Chen. An efficient algorithm for answering graph reachability queries. In *ICDE*, 2008.

[7] Y. Chen and Y. Chen. Decomposing dags into spanning trees: A new way to compress transitive closures. In *ICDE'11*, 2011.

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

[9] 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*, 2008.

[10] James Cheng, Zechao Shang, Hong Cheng, Haixun Wang, and Jeffrey Xu Yu. K-reach: who is in your small world. *Proc. VLDB Endow.*, 5(11), July 2012.

[11] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and distance queries via 2-hop labels. *SIAM J. Comput.*, 32(5):1338–1355, 2003.

[12] Daniel Delling, Martin Holzer, Kirill Mller, Frank Schulz, and Dorothea Wagner. High-performance multi-level graphs. In *9th DIMACS Implementation Challenge*, pages 52–65, 2006.

[13] Wenfei Fan, Xin Wang, and Yinghui Wu. Performance guarantees for distributed reachability queries. *Proc. VLDB Endow.*, 5(11), July 2012.

[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, Y. Xiang, and H. Wang. Path-tree: An efficient reachability indexing scheme for large directed graphs. *TODS*, 36(1), 2011.

[16] R. Jin, Y. Xiang, N. Ruan, and D. Fuhry. 3-hop: a high-compression indexing scheme for reachability query. In *SIGMOD'09*, 2009.

[17] R. Jin, Y. Xiang, N. Ruan, and H. Wang. Efficiently answering reachability queries on very large directed graphs. In *SIGMOD'08*, 2008.

[18] Ruoming Jin, Lin Liu, Bolin Ding, and Haixun Wang. Distance-constraint reachability computation in uncertain graphs. *Proc. VLDB Endow.*, 4(9), June 2011.

[19] Ruoming Jin, Ning Ruan, Saikat Dey, and Jeffrey Yu Xu. Scarab: scaling reachability computation on large graphs. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, 2012.

[20] H. Kriegel, P. Kröger, M. Renz, and T. Schmidt. Hierarchical graph embedding for efficient query processing in very large traffic networks. In *SSDBM '08*, 2008.

[21] E. Nuutila. *Efficient Transitive Closure Computation in Large Digraphs*. PhD thesis, Finnish Academy of Technology, 1995.

[22] P. Sanders and D. Schultes. Highway hierarchies hasten exact shortest path queries. In *17th Eur. Symp. Algorithms (ESA)*, 2005.

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

[24] S. Shekhar, A. Fetterer, and B. Goyal. Materialization trade-offs in hierarchical shortest path algorithms. In *SSD '97*, 1997.

[25] Houtan Shirani-Mehr, Farnoush Banaei-Kashani, and Cyrus Shahabi. Efficient reachability query evaluation in large spatiotemporal contact datasets. *Proc. VLDB Endow.*, 5(9), May 2012.

[26] K. Simon. An improved algorithm for transitive closure on acyclic digraphs. *Theor. Comput. Sci.*, 58(1-3):325–346, 1988.

[27] Mikkel Thorup. Compact oracles for reachability and approximate distances in planar digraphs. *J. ACM*, 51(6):993–1024, November 2004.

[28] S. Trißl and U. Leser. Fast and practical indexing and querying of very large graphs. In *SIGMOD '07*, 2007.

[29] S. J. van Schaik and O. de Moor. A memory efficient reachability data structure through bit vector compression. In *SIGMOD '11*, pages 913–924, 2011.

[30] H. Wang, H. He, J. Yang, P. S. Yu, and J. X. Yu. Dual labeling: Answering graph reachability queries in constant time. In *ICDE '06*, page 75, 2006.

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

[32] Zhiwei Zhang, Jeffrey Xu Yu, Lu Qin, Qing Zhu, and Xiaofang Zhou. I/o cost minimization: reachability queries processing over massive graphs. In *Proceedings of the 15th International Conference on Extending Database Technology*, EDBT '12, 2012.

[33] L. Zhu, B. Choi, B. He, J. X. Yu, and W. K. Ng. A uniform framework for ad-hoc indexes to answer reachability queries on large graphs. In *DASFAA '09*, 2009.