# A Highway-Centric Labeling Approach for Answering Distance Queries on Large Sparse Graphs *

Ruoming Jin† Ning Ruan† Yang Xiang‡ Victor E. Lee†
†Kent State University ‡ The Ohio State University
† {jin,nruan,vlee}@cs.kent.edu, ‡ yxiang@bmi.osu.edu

## ABSTRACT

The distance query, which asks the length of the shortest path from a vertex $u$ to another vertex $v$, has applications ranging from link analysis, semantic web and other ontology processing, to social network operations. Here, we propose a novel labeling scheme, referred to as *Highway-Centric Labeling*, for answering distance queries in a large sparse graph. It empowers the distance labeling with a highway structure and leverages a novel bipartite set cover framework/algorithm. Highway-centric labeling provides better labeling size than the state-of-the-art 2-hop labeling, theoretically and empirically. It also offers both exact distance and approximate distance with bounded accuracy. A detailed experimental evaluation on both synthetic and real datasets demonstrates that highway-centric labeling can outperform the state-of-the-art distance computation approaches in terms of both index size and query time.

## Categories and Subject Descriptors

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

## General Terms

Performance

## Keywords

Highway-centric labeling, Distance query, Bipartite set cover

## 1. INTRODUCTION

Computing the shortest path distance between any two vertices in a graph is a fundamental computer science problem and has been widely studied for several decades. The emergence of massive graph data, from social networks, the semantic web, biological networks, etc., and the need for this basic graph operator have recently attracted much interest in the database community [47, 26]. For instance, in a graph of the Web, the smallest number of links connecting two URLs can indicate page similarity [14]; in a semantic web ontology, the shortest path distance from one entity to another is also the key ingredient for ranking their relationship [5]; in a trust network, the number of hops from one person to another can indicate the level of trust [43]; in general social networks, the shortest path distance measures the closeness between users [46].

In the last decade, significant research progress has been made in both theoretical and practical distance computation (especially on road networks). The theoretical approach assigns each node $u$ a label (for instance, a set of vertices and the distances from $u$ to each of them) and then answers the distance query using the assigned labels [20]. The seminal work [45] by Thorup and Zwick shows a $(2k-1)$-multiplicative distance labeling scheme (the approximate distance is no more than $2k-1$ times the exact distance), for each integer $k \geq 1$, with labels of $O(n^{1/k}\log^2 n)$ bits. However, Potamias *et al.* [33] argue that for practical purposes, even $k=2$ is unacceptable (due to small-world phenomenon). The pioneering 2-hop method by Cohen *et al.* [12] provides exact distance labeling on directed graphs. Specifically, each vertex $u$ records a list of intermediate vertices $OUT(u)$ it can reach along with their (shortest) distances, and a list of intermediate vertices $IN(u)$ which can reach it along with their distances. To answer the shortest distance query from $u$ to $v$, the 2-hop method simply checks all the common intermediate vertices between $OUT(u)$ and $IN(v)$ and choose the vertex $p$, such that $dist(u,p)+dist(p,v)$ is minimized for all $p \in OUT(u) \cap IN(v)$. However, the computational cost to construct an optimal 2-hop labeling is very expensive [41, 28].

A variety of methods have been proposed to compute the (exact) shortest path distance in road networks [29, 30, 42, 39, 7, 27, 22, 38, 40, 32, 21, 44, 8]. Remarkably, Abraham *et al.* [2] recently discovered that the underlying mechanism for several of the fastest distance computation algorithms [27, 39, 7, 21, 8] can be explained by an intricate graph measure, referred to as *highway dimension*. Intuitively, a graph with low highway dimension (which road networks generally have) means for any $r$, there is a small and sparse set of vertices $S_r$, such that for any shortest path longer than $r$ hops must contains a vertex in $S_r$. Moreover, it turns out that the method with the best time bounds is actually a labeling algorithm [2], which resembles the 2-hop labeling. A fast and practical heuristic is developed in [1] to construct the distance labeling on large road networks. Interestingly, we also note that another state-of-the-art method, *Path-Oracle* [40] by Sankaranarayanan *et al.*, utilizes *Path-Coherent Pair (PCP)* for distance computation. Specifically, a PCP records two sets of vertices $A$ and $B$, along with a vertex or an edge that lies along all the shortest paths between $A$ and $B$. Indeed, this can be viewed as a different representation of the 2-hop labeling where each vertex in $A$ and $B$ explicitly records the shared the vertex or edge. In [40], both sets $A$ and $B$ are explicitly recorded using a spatial data structure.

Many real world graphs are quite sparse (similar to the road networks). However, they generally do not have the spatial and planar-like properties a road network has [26]. Though the 2-hop

*The work is partially supported by NSF CAREER award IIS-0953950, NSF Grant #1019343 to the CRA for the CIFellows Project, and Ohio Supercomputer Center Grant #PGS0218-1.


*SIGMOD'12,* May 20–24, 2012, Scottsdale, Arizona, USA.

or distance labeling have been effective on road networks [40, 1], it is unclear how these methods which aggressively utilize the spatial and planar-like properties can be applied to the general sparse graphs. *Can we construct distance labeling efficiently on general large sparse graphs?* This is an important question which clearly calls for an answer. Compared with online search algorithms [22, 26], the distance labeling approach can a provide much faster query result. The landmark [33] and Sketch [14], which can both be viewed as simplified and practical distance labeling algorithms, have been shown to quickly estimate the shortest path distance. Though the estimation can be rather accurate, they do not guarantee the exact results obtained from 2-hop labeling. Note that there have been a few studies using heuristics approaches [41, 10] to reduce the construction time for the 2-hop distance computation on general directed graphs. However, they do not produce the approximation bound of the labeling size produced by Cohen *et al.*'s approach [12].

Given this, the following questions are essential for applying distance-labeling for distance computation on general sparse graphs. 1) Can we derive better distance labeling for the exact distance query than the 2-hop approach, in terms of both indexing size, query time, and construction time? 2) Can we speed up 2-hop labeling without sacrificing its optimal labeling size? 3) Can we develop an approximate distance labeling scheme which can estimate distance with user-desired accuracy, i.e., $d(u,v) \leq \tilde{d}(u,v) \leq (1+\epsilon)d(u,v)$? Here $d(u,v)$ and $\tilde{d}(u,v)$ are the exact and estimated distance from vertex $u$ to vertex $v$, respectively, and $\epsilon$ is a user-defined accuracy threshold. 4) Can such labeling approach handle large sparse graphs?

In this work, we propose a novel labeling scheme, referred to as *Highway Centric Labeling* (HCL), for answering distance queries in a large sparse graph which provides positive answers to all these questions. Our scheme provides better labeling size than 2-hop does, both theoretically and empirically. Intuitively, it utilizes a *tree structured highway* to serve as the intermediaries to link the start vertex and end vertex, a generalization of 2-hop which only utilizes a single vertex as the intermediary. Though highway structure is widely used in shortest path computation in road networks, it is primarily used for speeding up the online search [29, 30, 39, 21, 32]. Based on our best knowledge, this is the first attempt to leverage highway structure in distance labeling. Also, the heart of the highway-centric labeling consists of a fast greedy algorithm for a general *bipartite set cover* problem, which by itself is very interesting and useful as it significantly generalizes the classic set cover problem. Furthermore, as a side product, we are able to speed up 2-hop without sacrificing its labeling size (empirically offering even better results). This scheme also offers both exact and approximate distance with bounded accuracy. Finally, the experimental studies show our approach can scale to large sparse graphs.

## 2. RELATED WORK

The shortest path distance (SPD) query is related to several key questions, such as single-source shortest path (SSSP) computation, in various branches of computer science. In the following, we overview the prior and related work on the SPD query. Throughout our discussion, we use $n$ and $m$ to denote the number of nodes and edges in the graph $G$, respectively.

**Shortest Path Computation:** One of the most well-known methods for shortest path computation is Dijkstra's algorithm [16]. It computes the single source shortest paths (SSSP) in a weighted graph and can be implemented with $O(m + n\log n)$ time. If the graph is unweighted, a Breadth-First Search (BFS) procedure can compute SSSP in $O(m+n)$. To compute the all-pairs shortest paths (APSP), we can directly invoke $n$ Dijkstra or BFS procedures with $O(nm+n^2\log n)$ and $O(nm)$, respectively. For sparse graphs, this can be very efficient. When the graph is dense, we may apply the Floyd-Warshall algorithm [13], which utilizes dynamic programming to compute APSP in $O(n^3)$.

To compute the point-to-point shortest path from source $u$ to sink $v$, the bidirectional algorithm running between two Dijkstra's algorithms, one from the source and the other from the sink, can be applied to improve search efficiency [36]. Goldberg *et al.* combine bidirectional search with the $A^*$ algorithm to improve the search performance [22]. The key ideas being explored in those works is how to derive better bounds to determine if an intermediate node should be visited to reach the destination.

In early database research, several techniques including graph decomposition and partial distance materialization are considered to help prune the search space [3, 24]. Recently, Wei utilizes the tree-width decomposition to help reduce the search space on unweighted undirected graphs [47]. Since the directed tree-width decomposition tends to be more elaborate and difficult, it is not clear whether it can be equally effective on directed or weighted graphs. In general, the online search for the shortest path distance (SPD) query is still very expensive (especially on large networks) due to the need to scan underlying graphs.

**Theoretical Distance Labeling:** Several studied have been performed to assign each node a label and then answer the SPD query using the assigned labels. Gavoille *et al.* show that general graphs support an exact distance labeling scheme with labels of $O(n)$ bits [19], and several special graph families, including trees or graphs with bounded tree-width, all have distance labeling schemes with $O(log^2 n)$ bit labels [4, 35, 19]. Several results are obtained for approximate distance labeling schemes. For arbitrary graphs, the best scheme to date is due to Thorup and Zwick [45]. They proposed a $(2k-1)$-multiplicative distance labeling scheme (the approximate distance is no more than $2k-1$ times the exact distance), for each integer $k \geq 1$, with labels of $O(n^{1/k}\log^2 n)$ bits. Basically, when $k=1$, it can support the exact labeling with storage cost being on the order of the entire distance matrix. However, when $k \geq 2$, the approximate distance can be 3-times higher than the true distance. Since in real world applications, most of the distances can be very small due to the small-world phenomenon, Potamias *et al.* [33] argue that this approximation bound is impractical.

Interestingly, Sarma *et al.* develop the *Sketch* method based on Thorup and Zwick's theoretical distance labeling and they show that in real Web graphs, distance labeling can provide fairly accurate estimation. More recently, Gubichev *et al.* further generalize the Sketch method to discover the shortest path (not only the distance) in large graphs [26].

**2-HOP Labeling:** The 2-hop labeling method proposed by Cohen *et al.* [12] is an exact distance labeling scheme. Each vertex $u$ records a list of intermediate vertices $OUT(u)$ it can reach along with their (shortest) distances, and a list of intermediate vertices $IN(u)$ which can reach it along with their distances. The index size is determined by the total number of intermediate vertices each vertex records. To answer the point-to-point shortest distance query from $u$ to $v$, we simply need to check all the common intermediate vertices between $OUT(u)$ and $IN(v)$ and choose the vertex $p$, such that $dist(u,p) + dist(p,v)$ is minimized for all $p \in OUT(u) \cap IN(v)$. They propose an approximate (greedy) algorithm based on set-covering which can produce a 2-hop cover with size no larger than the minimum possible 2-hop indexing by a logarithmic factor. The minimum 2-hop index size is conjectured to be $\tilde{O}(nm^{1/2})$ for directed graphs.

The major problem of the 2-hop indexing approach is its high

construction cost. Several heuristics approaches [41, 10] have been proposed to reduce construction time for the 2-hop distance computation. However, they do not produce the approximation bound of the labeling size produced by Cohen *et al.*'s approach [12].

**Landmark Encoding:** Several works uses *landmarks* to approximate the shortest path distance (SPD) [34, 31, 33]. The basic idea is to precompute the shortest distance between all nodes in the graph to these landmarks and then apply the triangle inequality to help estimate the shortest path distance. The recent work by Potamias *et al.* [33] investigates the selection of the optimal set of landmarks from the graphs. In particular, they target answering the point-to-point shortest path distance (SPD) query. Specifically, they introduce the LandMark-Cover problem which tries to find a minimum number of points such that for any pair of vertices $u$ and $v$, there exists at least one landmark in a shortest path from $u$ to $v$. This guarantees that each estimation is the exact distance. They show this problem is NP-hard and point out that a set-cover framework can solve this problem. Note that this problem has a very close relationship to the aforementioned 2-hop labeling scheme. In 2-hop, only a subset of relevant landmarks is assigned to a given node. Here, they assign the complete set of landmarks to each node. However, unlike 2-hop, this method employs a set of heuristics based on the node properties, such as degree or centrality, to select the landmarks, and landmarking does not aim to provide exact approximation. Instead, they use the small number of landmarks to estimate the SPD.

**Shortest Path Distance Computation on Road Networks:** Computing shortest path distance on road networks has been widely studied. Here we only provide a short review (More detailed review on this topic can be found in [15]). Several early studies [29, 30, 42] utilize the decomposition of a topological map to speed up shortest path search. For instance, in *HEPV* [29] and *HiTi* [30], they first decompose the entire map into fragments (subgraphs), then they extract the *border nodes* (nodes in fragments directly adjacent to nodes in other fragments), and finally the relationship of these border nodes form a new graph to express the high level structure of the graphs. Recently, a variety of techniques [15], such as Arc-flag (directing the search towards the goal) [8], highway hierarchies (building shortcuts to reduce search space) [27, 39], transit node routing (using a small set of vertices to relay the shortest path computation) [7], and utilizing spatial data structures to aggressively compress the distance matrix [38, 40], have been developed.

As we mentioned before, Abraham *et al.* [2] recently discovered that several of the fastest distance computation algorithms [27, 39, 7, 21, 8] need the underlying graphs to have small *highway dimension*. Furthermore, they demonstrate the method with the best time bounds is actually a labeling algorithm [2], which resembles the 2-hop labeling. In [1], they develop a fast and practical algorithm to heuristically construct the distance labeling on large road networks. Interestingly, another state-of-the-art method, *Path-Oracle* [40] by Sankaranarayanan *et al.*, also turns out to be derived from a similar idea as the 2-hop labeling. To sum, exact distance labeling has been shown to be fundamental and practical for distance computation on road network. However, it is not clear how these techniques which reply on the road network property can be broadened to a larger class of graphs.

# 3. LABELING OVERVIEW

The highway-centric labeling scheme simulates the usage of the highway system in a transportation network. In order to find the shortest travel distance from city $A$ to city $B$, we simply find the shortest distance from city $A$ to the correct entrance of an appropriate highway, and then get off at the correct exit which leads to the remaining shortest distance to city $B$. Indeed, highway structure has been studied for finding the shortest path in a large graph [39, 29, 30, 7]. However, they primarily utilize the highway structure to speed up the online search. A key thrust of our labeling scheme is to effectively utilize the highway structure for distance labeling (instead of searching). It also generalizes the distance labeling as it typically utilizes only intermediate vertices as the meeting point for distance computation [45, 12].

**Utilizing Highway in Distance Labeling:** Our distance labeling scheme utilizes the highway in the following fashion. Each vertex in the graph records an *outgoing list*, i.e., its shortest distances to a list of entry points in the highway, and an *incoming list*, i.e., the shortest distances from another list of exit points to this vertex. Here, we focus on directed graphs in this work. For undirected graphs, only one list is needed. Basically, the outgoing and incoming lists associated with each vertex are the "label" our scheme generates and uses. To query the shortest distance from the starting vertex to destination vertex, we simply match the outgoing list of the starting vertex with the incoming list of the destination vertex. If one vertex (an entry) of the outgoing list can reach another vertex (an exit) of the incoming list in the highway system, we refer to this as a *match* and assume their distance can be computed in constant time.

Thus, the estimated distance from $u$ to $v$ through such a match is simply the sum of three distances: *1) the distance from the starting vertex to the entry vertex in its outgoing list; 2) the distance from the entry vertex to the exit vertex in the highway; 3) the distance to the destination vertex from the exit vertex of its incoming list.* By choosing the smallest distance of all the matched pairs from the outgoing list of the starting vertex to the incoming list of the destination vertex, our labeling scheme can guarantee that the smallest distance is equivalent to the true shortest distance between them.

Figure 1 illustrates the difference between highway-centric labeling and 2-hop labeling. In highway-centric labeling, the highway connection can be effectively utilized as each vertex only needs to record distances to highway entries and distances from highway exits. Since 2-hop cannot utilize the highway connection, each vertex has to record more entries and exits than the new scheme to recover all the pairwise shortest path distances. Indeed, the 2-hop can be viewed as a special case of highway-centric labeling, where the highway degenerates to a set of disconnected points. In other words, the highway connection in the new scheme improves the capacity of these entry and exit vertices as they can combine together to encode shortest path distances.

**Utilizing Tree as Highway Structure:** For a highway structure to provide the necessary speed, we need a *distance oracle* to answer distance queries from any entrance to any exit quickly, e.g., in constant time. A simple solution would be to precompute all pairwise shortest distances. However, to encode all the exact shortest path distances, in the worst case the highway would include (nearly) every vertex in the graph. Then the size of the highway distance index would approach that of the graph's distance matrix ($O(n^2)$), leaving little benefit to having a highway. Basically, the distance oracle of the highway structure should have a small fingerprint.

In this work, we consider the highway structure to be a sparse subgraph of the entire graph and the distance oracle answers the distance from one vertex to another vertex by using only the highway structure. In other words, this distance is the "highway distance" and may not always be the shortest path distance in the original graph. As we will see in Section 4, this treatment enables a unified labeling scheme for *approximate* distance answering. In this work, we choose a tree to be the highway structure, and thus the distance oracle is simply *tree-distance*. Specifically, in a directed graph, our tree is a *rooted directed tree*, i.e., the root vertex

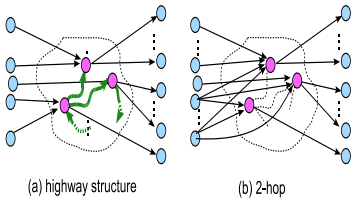



**Figure 1: highway structure vs 2-hop**

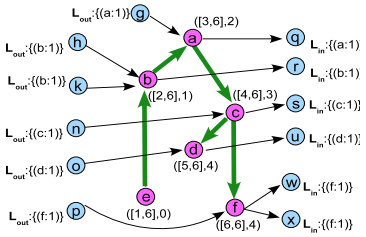
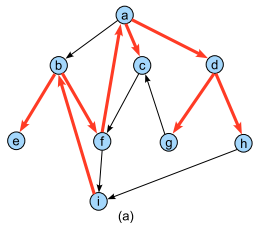



**Figure 2: An example of using tree as highway structure**




| $L_{out}$ | $L_{in}$ |
|---|---|
| a: {(f:2),(a:0),(i:3)} | a: {(a:0)} |
| b: {(b:0)} | b: {(a:1),(b:0)} |
| c: {(f:0)} | c: {(c:0)} |
| d: {(d:0),(c:2),(f:3),(i:2)} | d: {(d:0)} |
| e: {} | e: {(a:2),(e:0)} |
| f: {(f:0)} | f: {(f:0)} |
| g: {(f:2),(c:1)} | g: {(g:0)} |
| h: {(i:1)} | h: {(h:0)} |
| i: {(i:0)} | i: {(f:1),(i:0)} |

(b)

**Figure 3: (a) A directed graph with a spanning tree. Tree edges are highlighted in bold and red lines. (b) $L_{out}$ and $L_{in}$ labels for each vertex.**

(or virtual root) in the tree can reach all the other vertices in the tree and each vertex except the root has exactly one incoming edge [11].

Figure 2 shows a simple example of using a tree (in green) as a highway structure in answering distance queries. To tell the distance between two vertices, say $h$ and $w$, we compare $h$'s outgoing list, denoted as $L_{out}$ and $w$'s incoming list, denoted as $L_{in}$. A pair $(b : 1)$ in $L_{out}$ of vertex $h$ means $b$ is an entry of $h$ to the tree highway, and the shortest distance from $h$ to $b$ is 1. Similarly, a pair $(f : 1)$ in $L_{in}$ of vertex $w$ means $w$ can be reached from $f$ in distance 1. Since the tree distance from vertex $b$ to vertex $f$ is 3, we conclude the distance from $h$ to $w$ is 5.

## 3.1 Research Challenges and Our Approach

The major research problem we study in this work is as follows: *Given a directed graph $G$, how can we construct a rooted directed tree $T$ (a subgraph of $G$), and utilize it to label each vertex with outgoing list $L_{out}$ and incoming list $L_{in}$, so that the distance query can be efficiently answered with minimal labeling cost?*

A central challenge here is that because the highway links the entry and exit vertices, the choices of entry and exit vertices in $L_{out}$ and $L_{in}$ are not independent decisions. Thus, we cannot apply the classical set cover framework to find the optimal labeling cost, which is the core of all the existing works on 2-hop reachability and distance labeling [12], as well as the recent 3-hop reachability labeling [28]. Interestingly, we found our problem can be transformed into a new variant of the set cover problem, referred to as the *Set-Cover-with-Pairs* (SCP) problem. This SCP problem was recently proposed [37] and no good approximation algorithm is available to our best knowledge. It is still an open research question whether there exists a logarithmic bound approximation algorithm for SCP, as for the classical set cover problem [18].

Furthermore, we consider using a rooted directed spanning tree of the graph to serve as its highway structure. However, each tree may have different "routing power", i.e., each tree covers different sets (and different portions) of shortest paths. How to select an optimal spanning tree which can maximally cover shortest paths to minimize the labeling cost is another important research problem.

Overall, our approach includes the following key components:

**1.** Given a spanning tree as the highway structure, we transform the labeling problem to a special case of the SCP problem, which we refer to as the *bipartite set cover* (BSC) problem.

**2.**We propose a new approximation algorithm for the bipartite set cover problem and SCP problem. The approximation bound of our algorithm is very close to a logarithmic bound. Thus, our algorithm is of independent interest for the theoretical SCP problem.

**3.** We describe a linear distance labeling scheme for directed trees which provides a constant-time distance oracle. We efficiently construct the spanning tree that minimizes the labeling cost.

**4.** Utilizing a virtual tree, we demonstrate an efficient procedure for speeding up query processing.

# 4. LABELING FRAMEWORK

In this section, we study the following problem: *Given a rooted directed spanning tree $T$ of $G$, how can we utilize it to answering the distance query with minimal labeling cost?*

To facilitate our discussion, we begin by defining some notation and terminology. Let $G = (V, E)$ be a directed graph, where $V = \{1, 2, \cdots, n\}$ is the vertex set, and $E \subseteq V \times V$ is the edge set ($m = |E|$). We use $(v, w)$ to denote the edge from vertex $v$ to vertex $w$, and we use $(v_0, v_1, \cdots, v_p)$ to denote a *path* from vertex $v_0$ to vertex $v_p$, where $(v_i, v_{i+1})$ is an edge ($0 \leq i \leq p-1$). We say vertex $u$ can reach $v$ (denoted as $u \rightarrow v$) if there is a path starting from $u$ and ending at $v$. Let $TC(G)$ be the transitive closure of $G$, i.e., $TC(G) = \{(u, v)|u \rightarrow v\}$. The distance from vertex $u$ to $v$ in graph $G$ is denoted as $d(u, v)$, which is the shortest distance among all the paths from $u$ to $v$. In the unweighted graph, the shortest distance from $u$ to $v$ is simply the smallest path length of all the paths from $u$ to $v$. Further, let the rooted directed spanning tree $T = (V_T, E_T)$, where $V_T = V$ and $E_T \subseteq E$. We denote the root of $T$ as $r_T$. The distance from vertex $u$ to $v$ in tree $T$ is denoted as $d_T(u, v)$. Note that we may have a spanning forest instead and in this case, we can add a virtual root to a tree. For simplification, we focus our discussion on unweighted graphs in this paper, though our approach can be easily extended to weighted graphs (by replacing the path length in the unweighted graph with the general distance in the weighted graph).

## 4.1 The Labeling Problem

Our scheme assigns each vertex $u$ in $G$ two labels, $L_{out}(u)$ and $L_{in}(u)$. $L_{out}(u)$ records the outgoing list of entry vertices in the highway structure, tree $T$, and their respective distances, i.e., $L_{out}(u) = \{(x_1 : d_{x_1}), \ldots, (x_p : d_{x_p})\}$, where $d_{x_i} = d(u, x_i)$, the distance from $u$ to entry vertex $x_i$, $1 \leq i \leq p$. $L_{in}(u)$ records the incoming list of exit vertices from tree $T$, and their respective distances, i.e., $L_{in}(u) = \{(y_1 : d_{y_1}), \ldots, (y_q : d_{y_q})\}$, where $d_{y_j} = d(y_j, v)$, the distance from exit vertex $y_j$ to $v$, $1 \leq j \leq q$. Figure 3 shows a directed graph with a spanning tree (a) and the $L_{out}$ and $L_{in}$ label for each vertex in the graph (b). Given starting vertex $u$ and destination vertex $v$, let $(x_i : d_{x_i}) \in L_{out}(u)$ and $(y_j : d_{y_j}) \in L_{in}(v)$. We define the distance $d(u, v|x_i, y_j) =$

$$d_{x_i} + d_T(x_i, y_j) + d_{y_j} = d(u, x_i) + d_T(x_i, y_j) + d(y_j, v)$$

This is the distance of a path from $u$ to $x_i$ via a shortest path in $G$, from $x_i$ to $y_j$ via $T$ (which may be $\infty$, i.e., not reachable), and from $y_j$ to $v$ via a shortest path in $G$. Further, we define the *labeling distance* from vertex $u$ to $v$:

$$d(u, v|L_{out}(u), L_{in}(v)) = \min_{\substack{(x_i:d_{x_i}) \in L_{out}(u) \\ (y_j:d_{y_j}) \in L_{in}(v)}} d(u, v|x_i, y_j)$$

In addition, the size of the labeling is defined to be

$$Cost(HCL) = \sum_{u \in V(G)} (|L_{out}(u)| + |L_{in}(u)|)$$

Formally, we define the following highway-centric labeling ($HCL$ distance labeling) problems:

DEFINITION 1. **(Exact $HCL$ Distance Labeling Problem)** *Given a rooted directed spanning tree $T$ of $G$, the exact $HCL$ distance labeling problem is to assign each vertex an outgoing list $L_{out}(u)$ and an incoming list $L_{in}(v)$, such that for any vertices $u$ and $v$:*

$$\begin{aligned} (u,v) \in TC(G) &\implies d(u,v|L_{out}(u), L_{in}(v)) = d(u,v), \\ (u,v) \notin TC(G) &\implies d(u,v|L_{out}(u), L_{in}(v)) = \infty, \end{aligned}$$

*where the labeling size $cost(HCL)$ is minimal.*

DEFINITION 2. **(Approximate $HCL$ Distance Labeling Problem)** *Let $\epsilon$ be user-specified relative error for the point-to-point shortest distance query. The approximate $HCL$ distance problem is to assign each vertex an outgoing list $L_{out}(u)$ and an incoming list $L_{in}(v)$, such that for any vertex pair $u$ and $v$:*

$$\begin{aligned} (u,v) \in TC(G) &\implies d(u,v) \le d(u,v|L_{out}(u), L_{in}(v)) \\ &\quad \le (1+\epsilon) d(u,v), \\ (u,v) \notin TC(G) &\implies d(u,v|L_{out}(u), L_{in}(v)) = \infty, \end{aligned}$$

*where the labeling size $cost(HCL)$ is minimal.*

Since 2-hop can be treated as a special case of $HCL$, where all vertices in $G$ directly link to a virtual root, we can easily observe that given a rooted directed spanning tree $T$ of $G$, both the optimal exact $HCL$ distance labeling problem and optimal approximate $HCL$ distance labeling problem are as hard as the 2-hop labeling (the generalized 2-hop labeling is proved to be NP-hard [12]).

Furthermore, since $HCL$ distance labeling can utilize not only the intermediate vertices, but also the directed tree structure whereas 2-hop utilizes only the former, thus we observe:

THEOREM 1. *Given any spanning tree $T$ of $G$, the minimum distance labeling cost of $HCL$ is no larger than the minimum distance labeling cost of 2-hop, i.e., $cost(HCL) \le cost(2-hop)$.*

## 4.2 Transforming to Bipartite Set Cover (BSC)

The HCL distance labeling problem has a nice connection to the Bipartite Set Cover (BSC) problem. To understand this intrinsic connection, we briefly introduce the BSC problem as follows.

DEFINITION 3. **(Bipartite Set Cover (BSC) Problem)** *Let $U$ be the ground set of elements. Given a bipartite graph $\mathcal{G} = (A \cup B, \mathcal{E})$, where each edge $e$ in $\mathcal{E}$ is associated with a subset of elements in $U$ ($\mathcal{C}(e) \subseteq U$), we seek a subgraph $\mathcal{G}[\mathcal{X} \cup \mathcal{Y}]$ induced by $\mathcal{X} \cup \mathcal{Y}$ ($\mathcal{X} \subseteq A$ and $\mathcal{Y} \subseteq B$) to cover the ground set $U$, i.e., $U = \bigcup_{e \in \mathcal{E}(\mathcal{G}[\mathcal{X} \cup \mathcal{Y}])} \mathcal{C}(e)$ ($\mathcal{E}(\mathcal{G}[\mathcal{X} \cup \mathcal{Y}])$ is the edge set in the induced subgraph $\mathcal{G}[\mathcal{X} \cup \mathcal{Y}]$), with the minimal cost which is usually defined as $|\mathcal{X}| + |\mathcal{Y}|$.*

Figure 4(a) shows an example of the BSC problem. The ground set contains $a$,$b$,$c$,$d$,$e$,$f$. An optimal cover is a bipartite subgraph induced by vertices 2,3,5,7. By reducing the classical set-cover problem to BSC, we have

THEOREM 2. *The bipartite set covering (BSC) problem is an NP-hard optimization problem.*

Now we transform our distance labeling problem to the BSC problem as follows. Let the ground set $U = TC(G)$, i.e., it contains any vertex pair $(u,v)$ in $G$, such as $u \to v$. To cover a pair $(u,v)$ means we can restore their corresponding exact (or approximate) distance in the labeling scheme. We construct a bipartite graph $\mathcal{G} = (A \cup B, \mathcal{E})$ for the covering purpose: *a vertex in $A$ corresponds to a tuple $(u, (x_i : d_{x_i}))$, where $u$ is a starting vertex, $x_i$ is a candidate entry on the tree-highway structure, and $d_{x_i} = d(u, x_i)$ is the distance from $u$ to $x_i$; a vertex in $B$ corresponds to a tuple $((y_i : d_{y_j}), v)$, where $v$ is a destination vertex, $y_j$ is a candidate exit from the tree-highway, and $d_{y_j} = d(y_j, v)$ is the distance from $y_j$ to $v$. Given this, for any four vertices $(u, x_i, y_j, v)$, if*

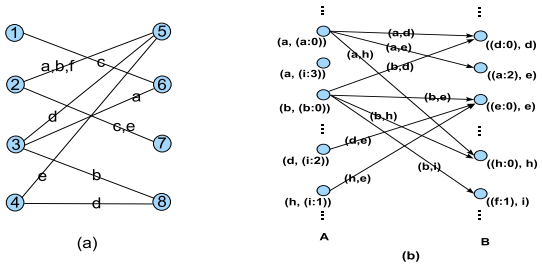



**Figure 4: (a) An example of BSC problem. (b) The bipartite graph (partially displayed) for the example in Figure 3.**

$$d(u,x_i) + d_T(x_i,y_j) + d(y_j,v) = d(u,v) \quad (1)$$

*we will add vertex $(u, (x_i : d_{x_i}))$ to $A$, add vertex $((y_i : d_{y_j}), v)$ to $B$, and add an edge from the first vertex to the latter one. For the approximate distance labeling, the condition will be generalized as*

$$d(u,x_i) + d_T(x_i,y_j) + d(y_j,v) \le (1+\epsilon) d(u,v) \quad (2)$$

*Note that those vertices may not be distinct, i.e., $u$ may be $x_i$, $x_i$ may be $y_j$, etc. In addition, we associate each edge with the singleton set $\{(u,v)\}$. Further, we assign a unit cost to each vertex in the bipartite graph, i.e., $cost(e) = 1$. It is not hard to see that if we can find vertex subsets $\mathcal{X} \cup \mathcal{Y}$ ($\mathcal{X} \subseteq A$ and $\mathcal{Y} \subseteq B$) to cover the ground set, we basically are able to restore all the distances for any reachable pair of vertices in graph $G$. Specifically, we have the following relationship:*

$$\begin{aligned} (u, (x_i : d_{x_i})) \in \mathcal{X} &\iff (x_i : d_{x_i}) \in L_{out}(u); \\ ((y_i : d_{y_j}), v) \in \mathcal{Y} &\iff (y_i : d_{y_j}) \in L_{in}(v). \end{aligned}$$

Thus, we have demonstrated that our labeling problem is an instance of the bipartite set cover (BSC) problem. Figure 4(b) illustrates the constructed bipartite graph.

## 4.3 Bipartite Set Cover Algorithms

In the following, we study efficient approximation algorithms for solving the bipartite set cover (BSC) problem. This is the key for generating the highway-centric labeling.

A straightforward approach is based on the observation that the bipartite set cover (BSC) can be considered a special case of the recently proposed set-cover-with-pairs problem [37].

DEFINITION 4. **(Set-Cover-with-Pairs Problem [37])** *Let $U$ be the ground set and let $Q = \{1, \ldots, M\}$ be a set of objects, where each object $i \in Q$ has a non-negative cost $w_i$. For every $\{i,j\} \subseteq Q$, let $\mathcal{C}(i,j)$ be the collection of elements in $U$ covered by the pair $\{i,j\}$. The objective of the set cover with pairs (SCP) problem is to find a subset $Q' \subseteq Q$ such that $\mathcal{C}(Q') = \bigcup_{\{i,j\} \subseteq Q'} \mathcal{C}(i,j) = U$ with a minimum covering cost $\sum_{i \in Q'} w_i$. We refer to the special case in which each object has a unit weight ($w_i = 1$) as the **cardinality SCP** problem.*

Using a graph formulation, we can see that the input of the SCP problem is a complete graph where each edge associates with a subset of the ground set $U$, each node has a weight, and the objective is to find a subset of nodes whose corresponding induced subgraph will cover $U$. In BSC, our input graph is a bipartite graph

$\mathcal{G} = (A \cup B, \mathcal{E})$. In addition, to configure our bipartite graph for distance labeling, each edge is associated with an element in $U$ (a single vertex pair in the original graph), and each node is associated with a unit cost. Given this, we can directly apply the existing algorithms for SCP to the BSC problem. However, the best available algorithm for SCP proposed in [37] yields an $O(\sqrt{N \log N})$ approximation ratio, where $N$ is the cardinality of the ground set $N = |U|$, for the cardinality SCP problem. Note that in our generated bipartite set cover for distance labeling, the ground set is $U = TC(G)$ and $N = |TC(G)|$, which in the worst case is $\frac{n(n-1)}{2}$. Thus, the approximation bound is $O(n\sqrt{\log n})$, which is almost useless in practise.

### 4.3.1 $\frac{2}{1-\alpha}(\ln N + 1)$-Approximate Algorithm

Here, we present a fast greedy algorithm (Algorithm 1) that yields $\frac{2}{1-\alpha}(\ln N + 1)$ approximation ratio for our BSC problem ($\alpha$ will be defined later). Note that each node has a unit cost in this problem. Our greedy algorithm works as follows: *each time we select a node in the bipartite graph $\mathcal{G} = (A \cup B, \mathcal{E})$ according to our defined criterion, until all the elements in the ground set, i.e., all pair shortest distance of $G$ in our HCL problem, have been covered.*

To describe our node-selection criterion, we introduce the following definitions: Let $U$ be the ground set and $R$ be the set recording the already covered elements: $R \subseteq U$. Let $S$ be the set of already selected nodes in the bipartite graph and let $\overline{S}$ denote the nodes not in $S$, i.e., $\overline{S} = (A \cup B) \backslash S$. We use $\mathsf{S}$ to denote the $S$ at the end of our algorithm, i.e., $\mathsf{S}$ is our solution.

Assume at some iteration, node $v$ is to be selected. Let $N_v$ ($N_v \subseteq U$) be the set of all elements on the edges adjacent to $v$. Let $S_v$ be those elements in $N_v$ which are immediately covered by $S \cup \{v\}$. It is easy to see that $S_v \subseteq N_v \setminus R$ holds. Let $Z_v = N_v \setminus (R \cup S_v)$, i.e., $Z_v$ is the set of uncovered elements in $N_v$. Intuitively, $|S_v|$ is exactly how many new elements will be covered by selecting $v$ at current iteration, and $|Z_v|$ is how many new elements may *potentially* be covered in future iterations as a consequence of selecting $v$. In other words, $|S_v| + |Z_v| = |N_v \backslash R|$ is the upper bound of the number of elements which can be covered by selecting node $v$.

In our algorithm, at each iteration, we select node $v$ from $\overline{S}$ with minimum $\frac{1}{|S_v|+|Z_v|}$. Intuitively, we can see that our algorithm with its criteria is "optimistic" or "far-sighted" as it always selects the node with "maximal potential" ($\max(|S_v| + |Z_v|)$) . As our analysis will show, this optimistic approach can produce much better approximation results (near logarithmic factor) than the existing SCP algorithm [37]. In the experimental results, we further show that this $\alpha$ is typically very small ($\alpha < 0.2$). Therefore, our algorithm has near logarithmic factor approximation ratio.

We now specify $\alpha$. For each selected node $v \in \mathsf{S}$, let $\hat{Z}_v$ be the set of elements in $Z_v$ that are covered as a consequence of selecting $v$ and $v'$. More precisely, let $\varepsilon$ be an element covered by $(v, v')$, where $v$ is selected before $v'$ is selected. If there is another node $u$ selected before $v'$ and $(u, v')$ is also associated with $\varepsilon$, this tie can be broken arbitrarily, i.e., we only consider one pair, either $(v, v')$ or $(u, v')$ but not both, covers $\varepsilon$. Without loss of generality, let $(v, v')$ cover $\varepsilon$, and thus, we have 1) $\varepsilon \in Z_v$; 2) $\varepsilon \in S_{v'}$; 3) $\varepsilon \in \hat{Z}_v$. Let $\Delta_{Z_v} = Z_v \backslash \hat{Z}_v$, and thus $|\Delta_{Z_v}| = |Z_v \backslash \hat{Z}_v|$ is the penalty from "overestimation".

THEOREM 3. *The ratio between the solution discovered by the greedyCover algorithm (Algorithm 1) and the optimal solution is bounded by $\frac{2}{1-\alpha}(\ln N + 1)$ where $\alpha = \frac{1}{|\mathsf{S}|} \sum_{v \in \mathsf{S}} \frac{|\Delta_{Z_v}|}{|S_v|+|Z_v|}$.*

Let $OPT$ be the cardinality of the optimal vertex set in $A \cup B$ for covering all elements in the ground set $U$. Then we need to prove $\frac{|\mathsf{S}|}{OPT} \leq \frac{2}{1-\alpha}(\ln N + 1)$. We note that when Algorithm 1 selects vertex $v$, its corresponding $S_v$, $Z_v$, and $\hat{Z}_v$ are considered to be finalized and will not change as the algorithm continues. In the following proofs, we only consider these finalized values.

**Algorithm 1** greedyCover($U$,$\mathcal{G} = (A \cup B, \mathcal{E})$)

**Parameter:** $U$ is the ground set and $\mathcal{G}$ is the bipartite graph for set cover.

```
R ← ∅;
S ← ∅;
while R ≠ U do
    Select a node v from (A ∪ B)\S such that 1/(|S_v|+|Z_v|) is minimum;
    S ← S ∪ {v}
    Add into R new elements covered by edges induced by v and S_v.
end while
return S
```

To prove this theorem, we first prove several lemmas.

LEMMA 1. *When Algorithm 1 selects $v$, $\frac{1}{|S_v|+|Z_v|} \leq \frac{OPT}{|\overline{R}|}$.*

**Proof Sketch:** This is because the vertex $v$ has the largest $|S_v| + |Z_v|$, and thus, considering all the vertices in the optimal solution, we shall have $(|S_v| + |Z_v|)OPT \geq (|S_v| + |Z_v|)(OPT - S) \geq |\overline{R}|$. □.

Next, we prove the following inequality:

LEMMA 2. $\sum_{v \in \mathsf{S}} \frac{|S_v|}{|S_v|+|Z_v|} \leq OPT \cdot (\ln |U| + 1)$.

**Proof Sketch:** First, we note that for some vertex $v \in \mathsf{S}$, its corresponding $S_v$ can be empty (no element is immediately covered by $v$ and $S$), and thus $|S_v|$ can be zero correspondingly. Given this, assume we rank each element in $U$ according to its order of covering in Algorithm 1, breaking ties arbitrarily for elements covered in the same iteration. Let $\varepsilon$ be the element with rank $k$, and covered by vertex $v(\varepsilon)$. Let $k'$ be the first element being covered by vertex $v(\varepsilon)$, i.e., $k' = |R| + 1$. Then we have $\frac{1}{|S_{v(\varepsilon)}|+|Z_{v(\varepsilon)}|} \leq \frac{OPT}{|U|-(k'-1)} \leq \frac{OPT}{|U|-k+1}$ (Lemma 1). Thus, $\sum_{\varepsilon \in U} \frac{1}{|S_{v(\varepsilon)}|+|Z_{v(\varepsilon)}|} \leq \frac{OPT}{|U|} + \frac{OPT}{|U|-1} + \cdots + \frac{OPT}{1} \leq OPT \cdot (\ln |U| + 1)$. Since $|S_v|$ is the number of elements being covered when selecting node $v$, we have

$$\sum_{v \in \mathsf{S}} \frac{|S_v|}{|S_v| + |Z_v|} = \sum_{\varepsilon \in U} \frac{1}{|S_{v(\varepsilon)}| + |Z_{v(\varepsilon)}|} \leq OPT \cdot (\ln |U| + 1). \ \square$$

Lemma 2 is from the perceptive of $S_v$ where $\bigcup S_v = U$. In the following we show a very interesting result: The above analysis also holds from the perspective of $\hat{Z}_v$.

LEMMA 3. $\sum_{v \in \mathsf{S}} \frac{|\hat{Z}_v|}{|S_v|+|Z_v|} \leq OPT \cdot (\ln |U| + 1)$.

**Proof Sketch:** We will show $\sum_{v \in \mathsf{S}} \frac{|\hat{Z}_v|}{|S_v|+|Z_v|} \leq \sum_{v \in \mathsf{S}} \frac{|S_v|}{|S_v|+|Z_v|}$. Note that if this is true, based on the proof of Lemma 2, the lemma holds. The key observation is that at any iteration, $\bigcup_{v \in S} S_v \subseteq \bigcup_{v \in S} \hat{Z}_v$ (and $\bigcup_{v \in \mathsf{S}} \hat{Z}_v = \bigcup_{v \in \mathsf{S}} S_v = U$). This is because for every element (say $\varepsilon \in U$) to be covered by $v$ in $S_v$, there must be a node $v'$ already selected in $S$, which can link to $v$ to cover it. In other words, $\varepsilon \in \hat{Z}_{v'}$ (also described in the definition of $\hat{Z}_{v'}$). Then, we conclude that $\frac{1}{|S_{v'}|+|Z_{v'}|} \leq \frac{1}{|S_v|+|Z_v|}$. This is due to two reasons: (1) $v'$ is selected before $v$. Therefore, at the time $v$ is selected, $\frac{1}{|S_{v'}|+|Z_{v'}|} \leq \frac{1}{|S_v|+|Z_v|}$ according to Algorithm 1. (2) For any vertex $u \in \overline{S}$ at an iteration of Algorithm 1, $\frac{1}{|S_u|+|Z_u|}$ will either remain the same or increase for the next iteration, because $|S_u| + |Z_u| = |S_u \cup Z_u| = |N_u \backslash R|$ will not increase.

Assume $\frac{1}{|S_v|+|Z_v|}$ is the price of covering an element in $S_v$, and $\frac{1}{|S_{v'}|+|Z_{v'}|}$ is the price of covering an element in $\hat{Z}_{v'}$. Thus, $\sum_{v' \in \mathsf{S}} \frac{|\hat{Z}_{v'}|}{|S_{v'}|+|Z_{v'}|}$ corresponds to the total cost of covering each

element in $U$ using set $\{\hat{Z}_{v'} : v' \in \mathsf{S}\}$. Therefore, we have $\sum_{v' \in \mathsf{S}} \frac{|\hat{Z}_{v'}|}{|S_{v'}|+|Z_{v'}|} \leq \sum_{v \in \mathsf{S}} \frac{|S_v|}{|S_v|+|Z_v|}$. □

Now we prove Theorem 3.

**Proof Sketch:** From Lemma 2 and Lemma 3, we get

$$|\mathsf{S}| = \sum_{v \in \mathsf{S}} \frac{|S_v|+|Z_v|}{|S_v|+|Z_v|} = \sum_{v \in \mathsf{S}} \frac{|S_v|+|\hat{Z}_v|}{|S_v|+|Z_v|} + \sum_{v \in \mathsf{S}} \frac{|\Delta_{Z_v}|}{|S_v|+|Z_v|}$$

$$\leq 2OPT \cdot (\ln |U| + 1) + \sum_{v \in \mathsf{S}} \frac{|\Delta_{Z_v}|}{|S_v|+|Z_v|}$$

This can be rewritten as:

$$|\mathsf{S}| \leq OPT \cdot \frac{2(\ln |U| + 1)}{1 - \frac{1}{|\mathsf{S}|} \sum_{v \in \mathsf{S}} \frac{|\Delta_{Z_v}|}{|S_v|+|Z_v|}} \leq OPT \cdot \frac{2(\ln N + 1)}{1 - \alpha}$$

where $\alpha = \frac{1}{|\mathsf{S}|} \sum_{v \in \mathsf{S}} \frac{|\Delta_{Z_v}|}{|S_v|+|Z_v|}$. □

**Speeding up the greedy algorithm:** In the greedy algorithm, we have to choose node $v$ with minimum $\frac{1}{|S_v|+|Z_v|}$. A straightforward implementation would compare all the remaining unselected nodes in the bipartite graph. However, we observe that this value $\frac{1}{|S_v|+|Z_v|}$ can only increase or remain the same after each iteration. Based on this observation, we can use a queue to maintain $\frac{1}{|S_v|+|Z_v|}$ for each $v$. Initially it is ranked in ascending order. Each time we select a node from the head of the queue to update its value. If this value is still smaller than the next value in the queue, we retrieve this node without updating the value for any other nodes in the queue. Otherwise we insert it back to the queue in the order of its new value. Note that a similar technique has been used in the standard set cover applications [41, 28]. It can result in up to $O(|A \cup B|)$ times speedup for our greedy cover algorithm.

**Alternative 2-hop Solution:** The original 2-hop approach [12] applies the classical set cover framework to deal with the special case of BSC problem. However, it has prohibitive computational complexity $O(n^3|TC(G)|)$ [28]. Now, we can apply our general BSC solver to deal with the bipartite graph ($\mathcal{G}_0$) generated by 2-hop.

**Complexity:** The time complexity of the greedy algorithm is as follows. To complete the covering, the main loop needs $O(\sum_{u \in V}(|L_{out}(u)|+|L_{in}(u)|))$ iterations in the worst case. Each iteration, using the speedup queue technique, assuming we only need to visit $\kappa << n$ nodes in the queue, takes $O(\kappa(\log |TC(G)|+|TC(G)|))$ time to extract and update. In the next section, we will introduce a heuristic to further reduce the cost to approximately $O(\kappa n)$ in each iteration.

# 5. CONSTRUCTING TREE-HIGHWAY

In the previous section, we assume the spanning tree is given. Here, we consider how to find a spanning tree which can assist in optimal distance labeling. In addition, we discuss a simple labeling scheme for a tree for constant-time distance oracle. Finally, we summarize the overall high-centric labeling approach.

## 5.1 Tree Construction Methods

Now, we introduce two criteria for selecting the best tree to serve as the highway structure for high-centric labeling.

**Criterion 1 (Shortest-Path Tree with Maximal Cover)**: Here we wish to find a spanning tree which maximally directly covers the shortest paths in the original graph $G$. We say tree $T$ directly covers the shortest path from $u$ to $v$ iff the distance in $T$ is equivalent to the distance using the entire graph, i.e., $d_T(u,v) = d(u,v)$. Formally, we define the objective function as follows:

$$C_1(T) = |\{(u,v)|(u,v) \in V \times V \wedge d_T(u,v) = d(u,v)\}| \quad (3)$$

Intuitively, the more shortest path distances are directly covered by the spanning tree, the fewer shortest path distances are left for the labeling process. This potentially reduces the size of labeling.

Finding the optimal tree $T$ in $G$ for (3) seems still very difficult. We propose to limit our candidate trees to only the shortest-path trees, i.e., the breadth-first search trees (BFS-trees) from vertices in $G$, which have the following appealing property:

LEMMA 4. *The total number of shortest paths directly covered by a BFS-tree $T$ is $C_1(T) = \sum_{v \in V(G)} desc(v)$, where $desc(v)$ is the number of all descendants of vertex $v$ in the tree.*

That is, in a BFS-tree $T$, if $u$ is the ancestor of $v$, then the path in the tree from $u$ to $v$ is the shortest one: $d_T(u,v) = d(u,v)$. The detailed proof is omitted for brevity. For this lemma, we can utilize a post-order traversal to compute $C_1(T)$. Given this, we can simply enumerate all BFS trees, compute their respective $C_1$, and choose the tree with maximal $C_1$ as our spanning tree (Highway structure) for the distance labeling. This can be done in $O(n(n+m))$ time.

**Criterion 2 (Directed MST on Edge-Betweenness)**: A limitation of the first criterion is its focus on only "directly" covered vertex pairs based on the tree structure. However, the purpose of the tree is to facilitate pair-wise "shortest path traffic" for all of $G$. Even though a tree may not directly cover many vertex pairs, a large number of shortest paths may employ some segment of the tree. Now the key question is how to measure a tree's utility for carrying such shortest path traffic. Consider the utility of a single edge in the tree: given an edge $e = (x,y)$ in graph $G$,

$$b(e) = |\{(u,v)|u,v \in V \wedge d(u,v) = d(u,x) + 1 + d(y,v)\}|$$

records the number of pairs whose shortest path go through edge $e$. In complex network theory, $b(e)$ is referred to as the *edge betweenness* [9]. Given this, we define the tree's utility $C_2(T)$ as the sum of all its tree edge betweenness measures: $C_2(T) = \sum_{e \in T} b(e)$. Thus, the optimal tree $T$ is the one which maximizes $C_2(T)$.

This optimal spanning tree problem (for $C_2(T)$) can be formulated as a *maximum directed spanning tree* problem if we first compute the edge betweenness $b(e)$ for each edge $e$ in graph $G$ and assign $b(e)$ as the edge weight for $e$. Given this, we can then utilize any well-known algorithm [11, 17] to find the maximal directed spanning tree of $G$ and use it in our distance labeling, which can be done in $O(n \log n + m)$. In addition, the exact edge-betweenness can be computed in $O(mn)$ [9] and faster accurate approximation can be achieved through sampling [6].

## 5.2 Distance Labeling for Rooted Directed Tree

Here, we introduce a simple labeling method for the rooted directed tree, which can provide constant distance oracle. Specifically, in a rooted directed tree, each vertex needs record only three numbers and offers a constant query time to answer the tree-distance between any two vertices in the tree.

The first two numbers are the interval labeling to answer the reachability from one vertex to another in the tree [25] and the third number is the distance to the root of the tree (or simply the depth of the node). There are several interval-labeling schemes [25] for a rooted directed tree. A simple scheme is as follows. We perform a preorder traversal of the tree to determine a sequence number for each vertex. Each vertex $u$ in the tree is assigned an interval: $[pre(u), index(u)]$, where $pre(u)$ is $u$'s *preorder* number and $index(u)$ is the highest preorder number of $u$'s successors. It is easy to see that *vertex $u$ is a predecessor of vertex $v$ ($u$ can reach $v$ in the tree) iff* $[pre(v), index(v)] \subseteq [pre(u), index(u)]$ [25].

Further, let $depth(u)$ be the third number (the depth) of vertex $u$ in the tree. To answer the tree-distance query from $u$ to $v$, we first use the interval labeling to check in $O(1)$ time if $u$ can reach $v$. If yes, we return $depth(v) - depth(u)$ as the tree-distance, and $+\infty$ otherwise. Figure 2 illustrates the distance labeling scheme for the highlighted tree, where the three numbers associated with each

vertex in the tree is the actual labeling (the first two are intervals and the last one is the depth).

## 5.3 Overall Labeling Algorithm and Batch Processing

In summary, here are the key steps of the highway-centric labeling approach:
**(Step 1):** Compute spanning tree $T$ based on Criterion (1) or (2) and label it with 3 integers;
**(Step 2):** Compute the pair-wise shortest path distances of $G$ and use it to construct the bipartite graph $\mathcal{G} = (A \cup B, \mathcal{E})$;
**(Step 3):** Run greedy bipartite set cover(BSC) algorithm.

The time complexity of Step 1 and 3 are analyzed earlier. In the following, we first focus on analyzing the complexity of Step 2. Here, the we need $O(n(n+m))$ time to compute the distance matrix and $O(n^3)$ time complexity to construct the bipartite graph (we will reduce this complexity later). Also, due to space limitation, we will focus on the exact distance labeling (the analysis of approximate labeling is similar). To observe the complexity of the bipartite graph construction, we introduce the following lemma.

LEMMA 5. *Given directed graph $G$ and spanning tree $T$, if vertex $u$ reaches both $x_i$ and $x'_i$, where $x_i$ is the parent of $x'_i$ in tree $T$, and $d(u,x_i)+1 = d(u,x'_i)$, then we can always replace $(x'_i : d_{x'_i})$ with $(x_i : d_{x_i})$ in $L_{out}(u)$ without affecting the correctness and optimality of the labeling results.*

Due to its simplicity, its proof is omitted. This lemma suggests we can construct the simplified bipartite graph $\mathcal{G} = (A \cup B, \mathcal{E})$ for graph $G$ by processing each vertex $u$ as follows. For any vertex $v \in TC(u)$ (reachable by $u$), and for any vertex $y$ in a shortest path from $u$ to $v$ ($u \cdots y \cdots v$), there is a unique vertex $x$ with the following property: *1) $x$ is on a shortest path from $u$ to $y$, i.e., ($u \cdots x \cdots y \cdots v$) (note that $x$ can be $u$ or $y$) and 2) $x$ to $y$ has the longest shortest path in the spanning tree $T$ ($d_T(x,y) = d(x,y)$).* Based on Lemma 5, for any $u$, $y$ and $v$, we only need to find this unique $x$ and add $(u, x : d(u,x)) \in A$, $(y : d(y,v), v) \in B$, and $((u, x : d(u,x)), (y : d(y,v), v)) \in \mathcal{E}$.

Since $x$ is $y$'s ancestor in the spanning tree $T$ and $x$ is on a shortest path from $u$ to $y$, we can easily compute $x$ for every $y \in TC(u)$ using a single BFS procedure (the overall cost is $O(n(m+n))$ for all $u \in V$). Furthermore, the transitive closure $\bigcup_{u \in V} TC(u)$ and the predecessor set $\bigcup_{v \in V} TC^{-1}(v)$ ($TC^{-1}(v)$ records all vertices which can reach $v$) can be computed in $O(n(m+n))$ (similar to invoking BFS $n$ times). Given this, to construct the bipartite graph, for every $u$ and $v \in TC(u)$, we find each $y \in TC^{-1}(v)$ with $d(u,y) + d(y,v) = d(u,v)$ and add $(u, x : d(u,x)) \in A$, $(y : d(y,v), v) \in B$, and $((u, x : d(u,x)), (y : d(y,v), v)) \in \mathcal{E}$. Thus, the overall cost of the bipartite graph is $O(n(m+n) + \sum_{u \in V} \sum_{v \in TC(u)} |TC^{-1}(v)|) = O(n^3)$.
**Batch Processing:** A potential issue of the overall labeling algorithm is that it requires complete materialization of the distance matrix (and also the transitive closure) for constructing the bipartite graph. It can be hard to hold the entire distance matrix in main memory for large graphs. Furthermore, the computational costs of constructing the bipartite graph and the BSC algorithm are both rather expensive. To deal with these problems, we utilize a batch processing procedure which not only reduces the memory needs but also constructs a simpler bipartite graph. The latter can speed up both the construction and BSC algorithm.

The batch processing strategy works as follows. Instead of trying to cover all the shortest pairs in a single large bipartite graph, in each batch, we will cover all the shortest pairs starting from a subset of vertices. Specifically, for each vertex $u$ in the batch, we will online compute its BFS-tree, and then add all the relevant vertices/edges into the bipartite graph $\mathcal{G}$. Note that a BFS-tree $T$ encodes the shortest distance for any ancestor-descendent pair, i.e., $d_T(u,v) = d(u,v)$ if $u$ is an ancestor of $v$ in $T$. Because of this, the bipartite graph is further simplified in the batch processing: considering vertex $u$ and its BFS tree is available, for any vertex $v \in TC(u)$, *instead of considering every vertex $y$ in all shortest paths between $u$ and $v$ ($u \cdots y \cdots v$), we only consider one shortest path encoded in the BFS tree*. In other words, $y$ is a vertex on the path in the BFS tree from the root $u$ to $v$. Thus, for each vertex $v$, the total number of $y$ is bounded by $D$, where $D$ is the diameter of the graph (the length of the longest shortest path). Given this, the cost of building the simplified bipartite graph can be written as $O(n(n+m) + n^2 D)$. Since most real graphs are rather sparse and tend to have the small-world property ($D$ is small), the overall time complexity is close to $O(n^2)$ for constructing bipartite graph by batch processing.

We also note that in the simplified bipartite graph, each node $(u,x)$ in $A$ links to at most $Dn$ nodes $(y,v)$ in $B$ and vice versa. Thus, the BSC algorithm, in each iteration, the update cost of selecting a node in the bipartite graph is bounded by $Dn$. Then the greedy BSC algorithm cost can be written as $O((\sum_u (|L_{out}(u)| + |L_{in}(u)|))\kappa(\log|TC| + Dn)) = O(C\kappa n^2)$, where $\kappa << n$ (please refer to the analysis at the end of Section **??**), and $C = (\sum_u (|L_{out}(u)| + |L_{in}(u)|))/|V|$ is the average labeling cost for each vertex. In the empirical study, $C$ is generally less than 10 and thus can be treated as a constant. Putting these together, we can see that the computational complexity of the overall labeling algorithm is close to $O(n^2)$ for large sparse graphs.

# 6. EFFICIENT QUERY PROCESSING

For any two vertices $u$ and $v$, let us denote their corresponding distance labels as $L_{out}(u) = \{(x_1 : d_{x_1}), \ldots, (x_p, d_{x_p})\}$ and $L_{in}(v) = \{((y_1 : d_{y_1}), \ldots, (y_q, d_{y_q})\}$, where $d_{x_i}$ and $d_{y_i}$ are the shortest distances from $u$ to $x_i$ and from $y_i$ to $v$, respectively. A straightforward approach to answer the shortest path distance query is to perform a pairwise join on $L_{out}(u)$ and $L_{in}(v)$ to find $\min\{d_{x_i} + d_T(x_i, y_j) + d_{y_j}\}$. This takes $O(pq)$ time.

Here, we introduce an efficient query processing algorithm with $O(p+q)$ linear complexity. Our algorithm is based on the observation that the vertices in the set $L_{out} \cup L_{in}$ can form a "virtual tree", which can be used for matching vertices between $L_{out}(u)$ and $L_{in}(v)$. Note: we say vertex *"$v$ is in $L$"* if in fact $(v : d_v) \in L$. Specifically, the *virtual tree* $T_{uv}$ on vertices of $L_{out} \cup L_{in}$ is defined as follows: *For any vertex pair $x_i$ and $x_j$ in $L_{out} \cup L_{in}$, if there is a path in spanning tree $T$ from $x_i$ to $x_j$ and no other $x_k$ in $L_{out} \cup L_{in}$ in this path, then we have an edge from $x_i$ to $x_j$ in the virtual tree $T_{uv}$.* In other words, $T_{uv}$ is a tree structure which preserves all the reachability relationship of those vertices in $L_{out} \cup L_{in}$ on the original rooted directed tree $T$.

Given this, we claim that *to compute the distance between $u$ and $v$, each vertex recorded in $L_{in}(v)$ needs only to join with its nearest ancestor belonging to $L_{out}(u)$ in the virtual tree $T_{uv}$.* Basically, for any vertex $L_{in}(v)$, instead of trying to join with all the vertices in $L_{out}(u)$ through the tree $T$, at most one vertex in $L_{out}(u)$ is needed for consideration. Lemma 6 supports the claim.

LEMMA 6. *For any vertex $y_k$ in $L_{in}(v)$, and let $x_i$ in $L_{out}(u)$ be an ancestor of $y_k$, and $x_j$ in $L_{out}(u)$ be an ancestor of $x_i$ in $T_{uv}$. Then, we have*

$$d(u,x_i) + d_T(x_i,y_k) + d(y_k,v) \leq d(u,x_j) + d_T(x_j,y_k) + d(y_k,v)$$

**Proof Sketch:** Clearly, we only need to prove $d(u,x_i) + d_T(x_i,y_k) \leq d(u,x_j) + d(x_j,y_k)$. Since $x_j$ is an ancestor of $x_i$ in $T_{uv}$, we have

**Algorithm 2** Query($u$,$v$)

```
1: merge L_in(v) and L_out(u) into V_uv in postorder of T_uv;
2: stack ← ∅;
3: for each r ∈ V_uv {follow the postorder in T_uv} do
4:    if r ∈ L_in(v) then
5:       stack.push(r);
6:    else
7:       compute all distance D = {d(u,r) + d_T(r,y) + d(y,v)} be-
         tween r and its successors y in the stack;
8:       dist_uv ← min(D, dist_uv);
9:       remove all r's successors from stack;
10:   end if
11: end for
12: return dist_uv;
```

$d_T(x_j, y_k) = d_T(x_j, x_i) + d_T(x_i, y_k)$. Moreover, $d(u, x_j) + d_T(x_j, x_i) \geq d(u, x_i)$, because $d(u, x_i)$ is the shortest distance of any path from vertex $u$ to vertex $x_i$. Thus, $d(u, x_j)+d_T(x_j, y_k) =$

$d(u, x_j) + d_T(x_j, x_i) + d_T(x_i, y_k) \geq d(u, x_i) + d_T(x_i, y_k)$.□

Based on Lemma 6 and the virtual tree $T_{uv}$, we basically need at most $q = |L_{in}(v)|$ distance check: $\min\{d_{x_i} + d_T(x_i, y_j) + d_{y_j}\}$, for each $y_i$ in $L_{in}(v)$, there is at most one $x_i$ in $L_{out}(v)$, which is $y_i$'s nearest ancestor belonging to $L_{out}(u)$ in $T_{uv}$. Given this, we need to show 1) the virtual tree can be built in linear time and 2) for each $y_j$, its corresponding $x_i$ can b discovered in linear time. The query procedure with $O(p + q)$ time complexity is depicted in Algorithm 2.

**Constructing Virtual Tree $T_{uv}$:** Our approach is to employ the postorder sequence to organize the vertices in $L_{in}(v)$ and $L_{out}(u)$. In a postordering, if $x$ is a descendant of $y$, then $x \preceq y$. We can utilize a stack to find out the descendant-ancestor relationship in the virtual tree. Note that we do not need to compute the postordering from scratch, because we can derive it from the existing tree interval labeling (assuming each vertex $u$ in the tree is assigned an interval: $[pre(u), index(u)]$, where $pre(u)$ is $u$'s *preorder* number and $index(u)$ is the highest preorder number of $u$'s successors). The postorder can be inferred in the following way: Given vertex $x$ with interval $[a_x, b_x]$ and vertex $y$ with interval $[a_y, b_y]$, then $x \preceq y$ (in the postorder) iff $[a_x, b_x] \subseteq [a_y, b_y]$, or $[a_x, b_x] \not\subseteq [a_y, b_y] \wedge b_x < b_y$. Thus, we can sort each label set easily ($L_{in}(v)$ and $L_{out}(u)$ both are organized in postorder of the spanning tree $T$), and more importantly, we can merge the sorted $L_{out}$ and $L_{in}$ in linear time (this corresponds the merging step, line 1, in the *Query* algorithm).

**Post-Order Traversal:** Once the virtual tree $T_{uv}$ is constructed, we perform a post-order traversal of the every vertex in $T_{uv}$. This is because the postorder traversal simulates a process to visit each vertex first before visiting its predecessors in increasing order according to their distance to this vertex. This procedure guarantees that a vertex's nearest predecessor will be visited before other predecessors. Therefore, we can traverse in postorder to quickly identify the nearest ancestor in $L_{out}(u)$ for any $y_j$ in $L_{in}(v)$. Specifically, for each vertex $r$, if it belongs to $L_{in}(v)$, then it is temporarily stored in a stack (lines 4 to 5) for further computations; otherwise (the vertex belongs to $L_{out}(u)$), then we join all vertices in the stack which are its descendents, and compute a distance between $u$ and $v$ (lines 7 to 8). Furthermore, each vertex in the stack will be processed at most once (Line 9). Put together, the *query* procedure needs only $O(p + q)$ time.

# 7. EMPIRICAL STUDY

In this section, we empirically study the performance of our approach, comparing it with other state-of-the-art distance query answering schemes on real and synthetic datasets. Specifically, our benchmarks include: the classical breadth-first search **BFS** and the state-of-the-art **ALT** algorithm (combining $A^*$ search, Landmark technique and Triangle inequality) [22, 23] in the online search category; the **2HOP** exact distance labeling scheme [12]; and **Sketch**, the latest landmark-based approximate distance query algorithm [14]. For our HCL labeling scheme, we consider the following alternatives: 1) **OptHCL-1**, HCL using approximate set cover algorithm and shortest-path tree with maximal cover criterion; 2) **OptHCL-2**, HCL using approximate set cover algorithm and directed MST with Edge-Betweenness criterion; 3) **NaïveHCL**, HCL using the naïve greedy algorithm from the set-cover-with-pairs problem [37]. 4) **2HOP**, the original 2-hop algorithm introduced by Cohen *et al.* [12]; and 5) **BSC2HOP**, a new implementation of 2-hop based on the bipartite set cover framework.

| Dataset | #V | #E | Avg.Deg | Dia. | Avg.LD |
|---|---|---|---|---|---|
| AgroCyc | 13969 | 17694 | 1.27 | 20 | 13.2 |
| Ecoo157 | 13800 | 17308 | 1.25 | 22 | 13.5 |
| GO | 6793 | 13361 | 1.97 | 11 | 2.3 |
| HpyCyc | 5565 | 8474 | 1.52 | 22 | 13.2 |
| Nasa | 5704 | 7942 | 1.39 | 24 | 3.1 |
| P2PG08 | 4055 | 12443 | 3.07 | 17 | 11.9 |
| P2PG09 | 4179 | 11944 | 2.86 | 18 | 13.7 |
| Reactome | 3678 | 14447 | 3.93 | 24 | 14.9 |
| Vchocyc | 10694 | 14207 | 1.33 | 22 | 13.5 |
| Xmark | 6483 | 7654 | 1.18 | 38 | 9.0 |
| Wiki-Vote | 7115 | 103689 | 14.57 | 10 | 4.7 |
| CiteSeer.scc | 693947 | 312282 | 0.45 | 12 | 2.3 |

**Table 3: Real datasets**

In addition, note that Sketch can only approximate the distance query, utilizing sampling techniques to control the approximation accuracy. A parameter $k$ determines the number of sampling times (typically the higher the $k$, the larger the index size, and the higher the accuracy). In [14], the parameter $k$ is set between 1 to 20. To make a fair comparison with *exact distance* query, Sketch's parameter $K$ is set to be 10 as it is the smallest sampling times which can produce indices with an average additive distance error to be within 1 in most datasets. For the *approximate distance* query, when $\varepsilon$ is set to be 0.5 and 1, Sketch's parameter $K$ is set to be 5 and 8, respectively, for comparable distance approximation accuracy (with respect to the HCL). For ALT, we observe that the forward ALT is around 2 times faster than bidirectional ALT [22] in directed unweighted graphs, even though fewer vertices are visited by the latter. Thus, we use the forward ALT as the benchmark, and we also use 8 randomly selected landmarks for pruning.

In each experiment, we measured the labeling time, label size, and query time needed to answer 100,000 random queries. In addition, the real value of $\alpha$ is also recorded to show how close our results are to the approximate logarithmic bound mentioned in subsection 4.2. We implemented all algorithms in C++. All experiments were conducted on a Linux server with 2.48GHz AMD Opteron processors and 24GB RAM.

## 7.1 Real Data

To validate our approaches on real-world datasets, we collected 12 datasets, listed in Table 3. We also present important characteristics of all real graphs, where $Avg.Deg$ is the average out-degree (i.e., $|E|/|V|$, which is also equivalent to the average in-degree in the directed graphs), $Dia.$ is graph diameter and $Avg.LD$ is average value of longest distances starting from vertices having immediate neighbors (i.e., vertex's in-degree is greater than 0 or vertex's out-degree is greater than 0). These graphs are generally sparse (the average degree is small). The graph size ranges from a few thousand to almost $700,000$ vertices. AgroCyc, Ecoo157, Hpy-

| Dataset | $\alpha$ | Label Size | | | | Query Time (in ms) | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | OptHCL-2 | NaiveHCL | BSC2Hop | Sketch | OptHCL-2 | NaiveHCL | BSC2Hop | Sketch | ALT | BFS |
| AgroCyc | 0.17 | 42199 | 2033601 | 138418 | 292831 | 36.432 | 7324.4 | 43.957 | 100.821 | 25865.4 | 2322.76 |
| Ecoo157 | 0.17 | 40758 | 1808305 | 130883 | 273210 | 33.429 | 5606.29 | 42.435 | 81.189 | 28972.2 | 2236.29 |
| GO | 0.10 | 20891 | 71388 | 64283 | 97227 | 28.24 | 136.732 | 33.352 | 40.263 | 2676.6 | 923.28 |
| HpyCyc | 0.18 | 23336 | 773741 | 67777 | 160026 | 47.662 | 6419.93 | 45.662 | 89.648 | 30658.4 | 1296.73 |
| Nasa | 0.04 | 10592 | 48984 | 36295 | 91792 | 14.053 | 54.241 | 24.535 | 37.292 | 3282.8 | 804.962 |
| P2PG08 | 0.13 | 120995 | 3358392 | 268565 | 327331 | 1947.17 | 205580 | 1229.47 | 217.326 | 81379.4 | 2576.75 |
| P2PG09 | 0.13 | 121983 | 3307903 | 274135 | 326105 | 1830.76 | 187586 | 1321.96 | 192.093 | 126586 | 2675.4 |
| Reactcome | 0.26 | 40755 | 6738338 | 452061 | 415157 | 395.917 | 1.06E+06 | 5.45E+02 | 4.75E+02 | 229649.6 | 4203.81 |
| VchoCyc | 0.19 | 37655 | 1801417 | 118069 | 253651 | 41.736 | 9841.03 | 47.847 | 100.287 | 47067.6 | 1991.13 |
| Xmark | 0.09 | 25617 | 654397 | 103777 | 315450 | 79.999 | 2685.32 | 132.128 | 139.823 | 43692.6 | 1330.66 |
| Wiki-Vote | 0.05 | 323444 | 15822487 | 2173473 | 517178 | 2817.94 | 2.35E+06 | 1434.8 | 204.919 | 1690304 | 10078.1 |
| CiteSeer.scc | 0.17 | 1182295 | 1165706 | 1044328 | 585114 | 22.908 | 21.735 | 47.673 | 37.448 | 463.4 | 84560.7 |

**Table 1: Label Size and Query Time of Real Datasets**

| Dataset | Label Size | | | | | Query Time | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | OptHCL-2 | $\varepsilon = 0.5$ | $\varepsilon = 1$ | Sketch (K=8) | Sketch (K=5) | OptHCL-2 | $\varepsilon = 0.5$ | $\varepsilon = 1$ | Sketch (K=8) | Sketch (K=5) |
| AgroCyc | 42199 | 32993 | 28810 | 252886 | 180886 | 36.432 | 27.313 | 22.528 | 84.585 | 58.654 |
| Ecoo157 | 40758 | 31799 | 27820 | 230037 | 168842 | 33.429 | 25.837 | 21.222 | 77.167 | 53.547 |
| GO | 20891 | 19195 | 19035 | 93808 | 70904 | 28.24 | 28.552 | 27.888 | 38.815 | 35.944 |
| HpyCyc | 23336 | 18048 | 16126 | 136267 | 99781 | 47.662 | 33.509 | 30.578 | 65.164 | 52.027 |
| Nasa | 10592 | 10382 | 10235 | 85991 | 69096 | 14.053 | 13.836 | 13.574 | 36.458 | 34.512 |
| P2PG08 | 120995 | 53735 | 31989 | 276314 | 183689 | 1947.17 | 597.509 | 266.974 | 176.473 | 103.942 |
| P2PG09 | 121983 | 55759 | 34165 | 276602 | 180089 | 1830.76 | 520.932 | 246.527 | 158.289 | 97.619 |
| Reactcome | 40755 | 33731 | 31570 | 347529 | 237151 | 395.917 | 3.38E+02 | 293.849 | 3.82E+02 | 2.44E+02 |
| VchoCyc | 37655 | 28868 | 24336 | 225920 | 161930 | 41.736 | 32.375 | 24.871 | 90.008 | 61.854 |
| Xmark | 25617 | 19843 | 17495 | 267994 | 179680 | 79.999 | 60.264 | 50.462 | 118.416 | 72.772 |
| Wiki-Vote | 323444 | 223151 | 149221 | 431653 | 288564 | 2817.94 | 1655.02 | 917.502 | 161.852 | 110.893 |
| CiteSeer.scc | 1182295 | 1182295 | 1182295 | 580183 | 564717 | 22.908 | 23.541 | 23.041 | 37.221 | 38.13 |

**Table 2: Label Size and Query Time on Real Datasets with Approximation Error**

Cyc, GO and VchoCyc are from EcoCyc; Xmark and Nasa are XML documents; Reactome is a metabolic network; CiteSeer.scc is extracted from the citation graph indexed by CiteSeer; P2PG09 is extracted from a sequence of snapshots of the Gnutella network collected in September 2002; and Wiki-Vote describes the relationships between users and their related discussion from the inception of Wikipedia until January 2008.

Due to space limitation, we present results only for OptHCL-2, NaïveHCL, and BSC2HOP based on the new highway-centric labeling scheme. In the next subsection, we will show that OptHCL-1 has slightly worse performance than OptHCL-2, due to the directed MST with Edge-Betweenness criterion producing a better tree for the highway centric labeling. In addition, 2HOP cannot work on most of the datasets due to insufficient scalability. We include these approaches for comparison on small synthetic datasets in the next subsection.

Table 1 shows the label size and query time for different query answering approaches. The label sizes for the online search algorithms BFS and ALT are not applicable. We also record the $\alpha$ value for OptHCL-2. Recall that $\alpha$ is an important parameter to indicate how close the approximation bound comes to a logarithmic bound ($\frac{2}{1-\alpha}(\ln N + 1)$, Subsection 4.3.1). The smaller the $\alpha$ is, the better the bound is. In our results, we can see that the value of $\alpha$ is quite small, close to or less than $0.2$ in most cases. Thus this confirms that the new approximation algorithm has almost logarithmic approximation bound.

**Label Size:** In the table, we see that OptHCL-2 and BSC2HOP consistently have the smaller label size. The size advantage of the new greedy algorithm (Subsection 4.3.1) is clearly demonstrated. Using the new BSC algorithm, the labeling sizes of OptHCL-2 are more than one order of magnitude smaller than those from NaïveHCL, which uses the original greedy algorithm based on the set-cover-with-pairs [37]. Interestingly, even compared with Sketch ($K = 10$), an approximate landmark-based distance labeling scheme, OptHCL-2 has smaller labeling size. The OptHCL-2 is much better than Sketch in 11 out of 12 datasets.

**Query Time:** In terms of query answering time, both OptHCL-2 and BSC2HOP are much faster than other distance answering approaches. OptHCL-2 is on average around $1.5$ times faster than the Sketch method. We also found that on unweighted graphs, the advanced search algorithm ALT does not perform well, being much slower than BFS. Without the edge weight and spatial/planar properties, the advanced pruning seems not to be very helpful. Overall, OptHCL-2 (and BSC2HOP) achieves significantly faster performance compared with ALT and BFS. In many cases, the speedup is close to two orders of magnitude. In addition, OptHCL-2 and BSC2HOP are on average more than two orders of magnitude faster than NaïveHCL. Interestingly, in several datasets, BSC2HOP is faster than OptHCL-2 even it has larger label size. This is because the query processing of OptHCL-2 needs more operations (e.g. merging two sets of vertices, temporary stack operations, and checking whether vertices in the temporary stack can be reached in the spanning tree introduced in Section 6) though both with the same query time complexity.

**Labeling Time:** For labeling time, OptHCL-2 takes 180, 157, 11, 51, 10, 322, 331, 397, 141, 110, 1003, and $253104$ seconds to construct indexing for 12 real datasets in the same order with Table 3. BSC2HOP is approximately $1.5$ times slower. Since NaïveHCL spends much more time than BSC2HOP, we do not report its labeling time here.

**Approximation:** In Table 2, we report the effect of approximation accuracy $\varepsilon$ on the labeling size and query time. Here we compare OptHCL-2 with Sketch approximate distance labeling method. With the approximation accuracy parameter $\varepsilon$ increasing from 0 to $0.5$ and 1, we observe that both labeling size and query time are reduced. The label sizes of OptHCL-2 with approximation errors $0.5$ and 1 are only $77\%$ and $67\%$ of exact OptHCL-2. Their query performance is $1.6$ and $2.5$ time faster than that of OptHCL-2. Comparing them with Sketch, we see that query times of OptHCL-2

| #V | Query Time (in ms) | | | | | |
|---|---|---|---|---|---|---|
| | OptHCL-1 | OptHCL-2 | NaiveHCL | 2HOP | BSC2HOP | BFS |
| 200 | 92.293 | 96.569 | 1967.56 | 535.753 | 121.447 | 251.951 |
| 400 | 166.297 | 177.43 | 5106.69 | 1387.78 | 248.939 | 430.28 |
| 600 | 209.905 | 245.966 | 10088.2 | 3169.97 | 330.853 | 586.934 |
| 800 | 171.904 | 179.511 | 12206.9 | 4264.74 | 329.784 | 684.068 |
| 1000 | 261.794 | 346.508 | 26514.8 | 6630.36 | 487.206 | 899.803 |

**Table 4: Query Time of Small Synthetic Datasets**

with $\varepsilon = 0.5$ and $\varepsilon = 1$ are approximately 1.3 and 1.5 times faster, respectively, than Sketch with $K = 5$. The labeling sizes of OptHCL-2 with $\varepsilon = 0.5$ and $\varepsilon = 1$ are on average 4.7 and 5.6 times smaller, respectively, than that of Sketch with $K = 5$. However, since the approximation labeling can significantly increase the density of the bipartite graph (for BSC), for large or dense graphs (like Wiki-Vote and CiteSeer.scc) and may introduce memory thrash, it can take too long to produce the labeling. The labeling time of OptHCL-2 for the datasets (from top to bottom) in Table 6 are 245 (343), 228 (301), 12 (13), 72 (101), 10 (10), 611 (857), 627 (871), 885 (1234), 210 (268), 137 (181), 1817 (2531) and 233957 (255904) seconds, respectively when $\epsilon = 0.5$ ($\epsilon = 1$). Basically, when the $\epsilon$ increases (more flexible distance estimation), the smaller is the index size, the faster is the query time, and the larger is the construction cost. This is because as we relax the distance error ratio (see (2) in Subsection 4.2), the bipartite graph becomes denser, taking longer time to discover the optimal cover.

## 7.2 Synthetic Data

We ran three sets of experiments using synthetic random directed graphs. Here, we focus on comparing different variants of exact labeling approaches. In the first experiment, we generated a set of random directed graphs with average out-degree (and also in-degree) of 1.5, varying the number of vertices from 200 to 1000. We compared six approaches, OptHCL-1, OptHCL-2, NaïveHCL, 2HOP, BSC2HOP, and BFS in this experiment. The number of vertices is small in the first test because the original 2HOP does not handle large graphs well.

Figure 5(a) shows the label size of five approaches. Here, OptHCL-1 and OptHCL-2 always obtain the best results among all algorithms. In particular, the label size of OptHCL-1 and OptHCL-2 are significantly better than 2HOP or BSC2HOP's. Overall, OptHCL-1 and OptHCL-2 are on average about 23 times and 25 times better than 2HOP, and about 11 times and 12 times better than NaïveHCL. Moreover, the label sizes of OptHCL-1 and OptHCL-2 are on average approximately 2.9 and 3.2 times smaller than that of BSC2HOP, respectively. Especially, BSC2HOP is on average 8 times better than the original 2HOP, confirming the power of the bipartite set cover framework. On the other hand, OptHCL-2 is always better than OptHCL-1 on all datasets.

Table 4 reports the query time of six algorithms including breadth-first search. Both OptHCL-1 and OptHCL-2 are approximately 3 times faster than BFS on answering distance queries. The label size directly affects the query time results, such that both OptHCL-1 and OptHCL-2 are much faster than the queries on 2HOP and NaïveHCL by an average of 13 times and 47 times, respectively. For labeling time, as we expected, 2HOP takes much longer labeling time than other approaches. The three approaches, OptHCL-1, OptHCL-2, and NaïveHCL, which employ the same bipartite set cover framework vary only slightly on the labeling time.

Next, we evaluate the performance of the HCL approach on synthetic data with different graph sparsity. In this experiment, we generate a set of random directed graphs with 5000 vertices, while varying the average out-degree from 1.2 to 2.

The query times of all three algorithms are listed in Table 5. As expected, the large label size greatly influences the query time. In our test, OptHCL-2 is 157 times faster than NaïveHCL on answering 100,000 randomly generated queries. For the other two approaches, OptHCL-2 is on average 4.7 times faster than BFS, and 1.5 times faster than BSC2HOP. In addition, on average, the label size of OptHCL-2 is about 34 times and 3 times smaller than those of NaïveHCL and BSC2HOP, respectively. Also, in terms of the labeling time, OptHCL-2 is consistently the most efficient one on all graphs with different vertex degree. Due to the space limitation, the figures on the labeling size and time is omitted here.

| Avg.Deg | Query Time (in ms) | | | |
|---|---|---|---|---|
| | OptHCL-2 | NaiveHCL | BSC2HOP | BFS |
| 1.2 | 115.2 | 9727.06 | 293.65 | 1793.43 |
| 1.4 | 715.1 | 134390 | 1353.48 | 3534.48 |
| 1.6 | 3599.9 | 553867 | 3912.90 | 5033.43 |
| 1.8 | 6898.5 | 1.22E+06 | 6558.09 | 6042.27 |
| 2 | 9528.2 | 1.74E+06 | 9464.09 | 6541.99 |

**Table 5: Query Time of Synthetic Dataset (5K) varying sparsity**

| Dataset | Query Time (in ms) | | | |
|---|---|---|---|---|
| | OptHCL-2 | NaiveHCL | BSC2HOP | BFS |
| DAG200K | 41.134 | 58.301 | 64.793 | 13609.9 |
| DAG400K | 36.545 | 54.953 | 55.291 | 43275.8 |
| DAG600K | 32.902 | 56.14 | 64.938 | 94253.7 |
| DAG800K | 40.752 | 56.199 | 184.936 | 144556 |
| DAG1M | 53.387 | 67.224 | 245.925 | 185319 |

**Table 6: Query Time of Large DAG Datasets**

To test the scalability of our highway-centric approach on large DAGs (directed acyclic graphs), we generate very large DAGs, with the number of vertices ranging from $200K$ to $1M$, while keep the average out-degree at 1.5. Figure 5(b) and Figure 5(c) show the label size and labeling time of three algorithms. For label size, we can see that OptHCL-2 is better than NaïveHCL and BSC2HOP, outperforming them by approximately 20% and 33% label size, respectively. We observe that HCL's performance on large DAGs is not as impressive as it was with the previous random directed graphs. Our approximate set cover with pairs algorithm tends to work very well on graphs with large transitive closure, because one label potentially covers many shortest paths in this case. However, in directed acyclic graphs, the transitive closure size is significantly smaller than in random directed graphs. This eventually leads to a decrease of improvement on label size compression rate. As in previous experiments, OptHCL-2 is still the fast labeling approach among three algorithms. Table 6 lists the query time of four algorithms. Overall, OptHCL-2 is significantly faster than BFS, outperforming it by more than 2000 times. It is on average 1.5 times and 2.8 times faster than NaïveHCL and BSC2HOP.

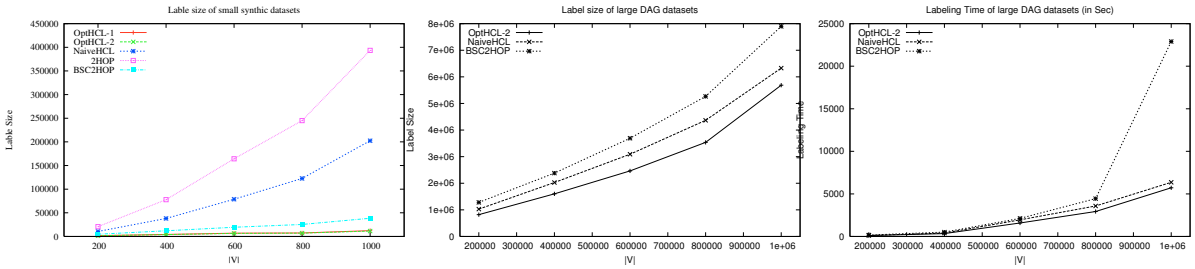
# 8. CONCLUSION

In this paper, we propose to answer distance queries in large sparse graphs by a highway-centric labeling scheme. We find an interesting link between our labeling problem and Bipartite Set Cover (BSC) problem. To find a near optimal solution to the BSC problem, we propose an elegant yet simple algorithm, and prove rigorously that our algorithm yields a non-trivial logarithmic bound. We show both theoretically and empirically that HCL is better than 2-hop, the state-of-the-art labeling scheme for exact distance queries, in all key aspects – labeling size, query time, and indexing time. We plan to study how to further scale HCL and its incremental maintenance on dynamic graphs.

# 9. 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] I. Abraham, A. Fiat, A. V. Goldberg, and R. F. Werneck. Highway dimension, shortest paths, and provably efficient algorithms. In *SODA '10*, 2010.




(a) Small Synthetic Datasets (200-1K) (b) Large Synthetic DAG (200K-1M) (c) Large Synthetic DAG (200K-1M)

**Figure 5: Experimental Results of Synthetic Datasets**


[3] R. Agrawal and H. V. Jagadish. Algorithms for searching massive graphs. *TKDE*, 6(2):225–238, 1994.

[4] Stephen Alstrup, Philip Bille, and Theis Rauhe. Labeling schemes for small distances in trees. In *SODA*, pages 689–698, 2003.

[5] K. Anyanwu and A Sheth. P-queries: enabling querying for semantic associations on the semantic web. In *WWW '03*, 2003.

[6] D. A. Bader, S. Kintali, and M. Madduri, K.and Mihail. Approximating betweenness centrality. In *WAW'07: Proc. 5th Int'l Conf. Algor. and models for the web-graph*, 2007.

[7] H. Bast, S. Funke, P. Sanders, and D. Schultes. Fast Routing in Road Networks with Transit Nodes. *Science*, 316:566–, April 2007.

[8] 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.

[9] U. Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25:163–177, 2001.

[10] J. Cheng and J. X. Yu. On-line exact shortest distance query processing. In *EDBT '09*, 2009.

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

[12] 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.

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

[14] A. Das Sarma, S. Gollapudi, and R. Najork, M.and Panigrahy. A sketch-based distance oracle for web-scale graphs. In *WSDM '10*, 2010.

[15] D. Delling, P. Sanders, D. Schultes, and D. Wagner. Algorithmics of large and complex networks. chapter Engineering Route Planning Algorithms. 2009.

[16] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, December 1959.

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

[18] U. Feige. A threshold of ln n for approximating set cover. *J. ACM*, 45(4):634–652, 1998.

[19] C. Gavoille, D. Peleg, S. Pérennes, and R. Raz. Distance labeling in graphs. *J. Algorithms*, 53(1):85–112, 2004.

[20] Cyril Gavoille, David Peleg, Stéphane Pérennes, and Ran Raz. Distance labeling in graphs. *J. Algorithms*, 53(1):85–112, 2004.

[21] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction hierarchies: faster and simpler hierarchical routing in road networks. In *Proceedings of the 7th international conference on Experimental algorithms*, 2008.

[22] A. V. Goldberg and C. Harrelson. Computing the shortest path: A search meets graph theory. In *SODA '05*, 2005.

[23] A. V. Goldberg, H. Kaplan, and R. F. Werneck. Reach for a*: Efficient point-to-point shortest path algorithms. In *IN WORKSHOP ON ALGORITHM ENGINEERING and EXPERIMENTS*, pages 129–143, 2006.

[24] R. Goldman, N. Shivakumar, S. Venkatasubramanian, and H. Garcia-Molina. Proximity search in databases. In *VLDB '98*, 1998.

[25] G. Gou and R. Chirkova. Efficiently querying large xml data repositories: A survey. *TKDE*, 19(10):1381–1403, 2007.

[26] A. Gubichev, S. Bedathur, S. Seufert, and G. Weikum. Fast and accurate estimation of shortest paths in large graphs. In *CIKM '10*, 2010.

[27] R. J. Gutman. Reach-based routing: A new approach to shortest path algorithms optimized for road networks. In *ALENEX/ANALC*, pages 100–111, 2004.

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

[29] N. Jing, Y. Huang, and E. A. Rundensteiner. Hierarchical encoded path views for path query processing: An optimal model and its performance evaluation. *TKDE*, 10(3):409–432, 1998.

[30] S. Jung and S. Pramanik. An efficient path computation model for hierarchically structured topographical road maps. *TKDE*, 14(5):1029–1046, 2002.

[31] J. Kleinberg, A. Slivkins, and T. Wexler. Triangulation and embedding using small sets of beacons. In *FOCS '04*, 2004.

[32] 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.

[33] C. Castillo M. Potamias, F. Bonchi and A. Gionis. Fast shortest path distance estimation in large networks. In *CIKM '09*, 2009.

[34] T. S. Eugene Ng and H. Zhang. Predicting internet network distance with coordinates-based approaches. In *INFOCOM*, 2001.

[35] David P. Proximity-preserving labeling schemes and their applications. *Journal of Graph Theory*, 33(3):167–176, 2000.

[36] I. Pohl. Bi-directional search. *Machine Intelligence*, 6:124–140, 1971.

[37] H. Refael and S. Danny. The set cover with pairs problem. In *FSTTCS 2005: Foundations of Software Technology and Theoretical Computer Science, 25th Int'l Conference*, 2005.

[38] H. Samet, J. Sankaranarayanan, and H. Alborzi. Scalable network distance browsing in spatial databases. In *SIGMOD'08*, 2008.

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

[40] J. Sankaranarayanan, H. Samet, and H. Alborzi. Path oracles for spatial networks. *PVLDB*, 2, August 2009.

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

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

[43] G. Swamynathan, C. Wilson, B. Boe, K. Almeroth, and B. Y. Zhao. Do social networks improve e-commerce?: a study on social marketplaces. In *WOSP '08: Proc. 1st workshop on Online social networks*, 2008.

[44] Y. Tao, C. Sheng, and J. Pei. On $k$-skip shortest paths. In *SIGMOD'11*, 2011.

[45] M. Thorup and U. Zwick. Approximate distance oracles. *J. ACM*, 52(1):1–24, 2005.

[46] M. V. Vieira, B. M. Fonseca, R. Damazio, P. B. Golgher, D. de C. Reis, and B. Ribeiro-Neto. Efficient search ranking in social networks. In *CIKM '07*, 2007.

[47] F. Wei. Tedi: efficient shortest path query answering on graphs. In *SIGMOD'10*, 2010.