The VLDB Journal (2012) 21:869–888
DOI 10.1007/s00778-012-0274-x

REGULAR PAPER

The exact distance to destination in undirected world

Lijun Chang · Jeffrey Xu Yu · Lu Qin · Hong Cheng · Miao Qiao

Received: 8 August 2011 / Revised: 22 March 2012 / Accepted: 28 March 2012 / Published online: 26 April 2012
© Springer-Verlag 2012

Abstract Shortest distance queries are essential not only in graph analysis and graph mining tasks but also in database applications, when a large graph needs to be dealt with. Such shortest distance queries are frequently issued by end-users or requested as a subroutine in real applications. For intensive queries on large graphs, it is impractical to compute shortest distances on-line from scratch, and impractical to materialize all-pairs shortest distances. In the literature, 2-hop distance labeling is proposed to index the all-pairs shortest distances. It assigns distance labels to vertices in a large graph in a pre-computing step off-line and then answers shortest distance queries on-line by making use of such distance labels, which avoids exhaustively traversing the large graph when answering queries. However, the existing algorithms to generate 2-hop distance labels are not scalable to large graphs. Finding an optimal 2-hop distance labeling is NP-hard, and heuristic algorithms may generate large size distance labels while still needing to pre-compute all-pairs shortest paths. In this paper, we propose a multi-hop distance labeling approach, which generates a subset of the 2-hop distance labels as index off-line. We can compute the multi-hop distance labels efficiently by avoiding pre-computing all-pairs shortest paths. In addition, our multi-hop distance labeling is small in size to be stored. To answer a shortest distance query between two vertices, we first generate the query-specific small set of 2-hop distance labels for the two vertices based on our multi-hop distance labels stored and compute the shortest distance between the two vertices based on the 2-hop distance labels generated on-line. We conducted extensive performance studies on large real graphs and confirmed the efficiency of our multi-hop distance labeling scheme.

Keywords Large graph · Shortest distance queries · Graph labeling

L. Chang · J. X. Yu (✉) · L. Qin · H. Cheng · M. Qiao
The Chinese University of Hong Kong, Hong Kong, China
e-mail: yu@se.cuhk.edu.hk

L. Chang
e-mail: ljchang@se.cuhk.edu.hk

L. Qin
e-mail: lqin@se.cuhk.edu.hk

H. Cheng
e-mail: hcheng@se.cuhk.edu.hk

M. Qiao
e-mail: mqiao@se.cuhk.edu.hk

1 Introduction

With the rapid growth of Internet, more and more large datasets are collected and archived. Among them, graph data are of great importance. As turning data into profit is essential in the fiber age, efficiently querying and analyzing graph data have drawn a lot of attention in the database community.

Shortest path query is one of the fundamental operations on graph data. In a social network, users are considered as vertices and edges represent friend relationship between them, and a common question to ask is how close the friendship is. Erdős distance is a well-known tongue-in-cheek measurement of mathematical prominence of researchers in scientific circles, which is the distance between a person and the mathematician Paul Erdős in a collaboration network. In biological networks, shortest paths and distance information are employed to identify optimal pathways and valid connectivity in metabolic networks [20]. A travel agency needs to find the cheapest route from one place to another destination. Other applications, such as keyword search [32], twig-pattern matching [13], and graph pattern matching [10], also involve

a lot of shortest distance computations, or even the all-pairs shortest distances.

Shortest distance queries have been extensively studied. The most well-known main memory algorithms are BFS for unweighted graphs and Dijkstra's algorithm for weighted graphs [8], which compute shortest distances by traversing the original graph from scratch. The time complexity of BFS and Dijkstra's algorithm are $O(|V| + |E|)$ and $O(|E| + |V| \log |V|)$, where $|V|$ is the number of vertices and $|E|$ is the number of edges, respectively. Therefore, it is impractical to compute shortest distances from scratch for intensive queries on large graphs. On the other extreme, a naive solution is to pre-compute and materialize the all-pairs shortest distances. This will need $O(|V|^2)$ space, which is prohibitive for large graphs.

Cohen et al. [7] propose a family of labelings over graphs to support reachability and shortest distance queries. In the distance-aware 2-hop distance labeling, it assigns distance labels to vertices in the graph, then shortest distances can be computed using the distance labels directly while avoiding exhaustively traversing the large graph. It is conjectured in [7] that the size of the optimal 2-hop distance labels is $O(|V| \cdot |E|^{1/2})$. Though it is appealing for the theoretical bound on the space complexity, unfortunately, computing optimal 2-hop distance labels is challenging. First, it needs to pre-compute the all-pairs shortest paths, which is prohibitive for large graphs. Second, for the all-pairs shortest paths, it needs to find the optimal 2-hop cover, which is NP-hard [7]. Several reported studies show that the cost of computing 2-hop cover over directed graphs is high [6,24,25].

In this paper, we focus on answering the exact distance over an undirected graph based on the labeling approach. The problem becomes harder because all vertices in an undirected graph are possibly connected pairwise, and the existing approaches [6,24,25] developed over directed graphs cannot be directly applied. We propose a new multi-hop distance labeling approach, which encodes the shortest paths compactly and can answer the exact shortest distance between two vertices in an undirected graph efficiently. In the multi-hop distance labeling, we relax the condition that all shortest paths should be covered in 2-hops, and we generate multi-hop distance labeling efficiently by avoiding both pre-computing all-pairs shortest paths and finding a 2-hop cover for all such pairs computed. To answer a shortest distance query for u and v, we first generate on-line the necessary but small set of 2-hop distance labels for u and v based on our multi-hop distance labels and then compute the exact shortest distance between u and v based on the 2-hop distance labels generated on-line.

The main contributions of this work are summarized below. First, we propose a new multi-hop distance labeling. The unique features of our multi-hop distance labeling are: it can be generated efficiently and is small in size, and it can be efficiently used to compute the exact shortest distances for any pair of vertices for a large undirected graph. Second, we give efficient algorithms to generate multi-hop distance labels based on vertex separators. Third, we propose efficient algorithms to compute shortest distances based on the 2-hop distance labels generated on-line using our multi-hop distance labels. Finally, we conducted extensive performance studies using large real and synthetic graphs and confirmed the efficiency of our multi-hop distance labeling scheme.

The remainder of the paper is organized as follows. We discuss distance labeling and our problem definition in Sect. 2. In Sect. 3, we define multi-hop distance labeling and show general steps to compute shortest distances based on our distance labels. The algorithm to generate multi-hop distance labels is introduced in Sect. 4. We give efficient algorithms to compute shortest distances based on distance labels in Sect. 5. The related works are discussed in Sect. 7. We conducted experimental studies and discuss our findings in Sect. 8. Section 9 concludes the paper.

2 Problem statement

We consider an unweighted and undirected graph, $G = (V, E)$, with a vertex set $V = \{v_1, \ldots, v_n\}$ and an edge set $E = \{e_1, \ldots, e_m\}$, where an edge $e_i = (u, v)$ is a pair of unordered vertices. A path connecting u and v is an ordered list of vertices, denoted as $P(u, v) = (w_0, \ldots, w_l)$, where $w_0 = u$, $w_l = v$, and every pair of vertices (w_{i-1}, w_i) is an edge in E, for $1 \leq i \leq l$. The length of a path $P(u, v)$ is the number of edges in $P(u, v)$. The shortest distance between u and v is the smallest length among all the paths connecting u and v, denoted as $\delta(u, v)$. Without loss of generality, we assume that G is a *simple graph*, that is, there are no self loops nor multiple edges. The numbers of vertices and edges of G are denoted as $n = |V|$ and $m = |E|$, respectively.

Definition 1 *Distance Labeling* ([7,11]): A distance labeling of a graph $G = (V, E)$ is a pair (L, F). Here, L is a *distance labeling function* that assigns a label to every vertex $v \in V$, and F is a *distance decoding function* that computes shortest distance for a pair of vertices (u, v) using the labels in a way such as $F(L(u), L(v)) = \delta(u, v)$.

In this paper, we focus on a *2-hop distance labeling* [7], which assigns a vertex u a label in the form of $L(u) = \{(w_1, \delta(u, w_1)), \ldots, (w_l, \delta(u, w_l))\}$. Here, every pair of $(w_i, \delta(u, w_i))$ implies that the shortest distance between vertex u itself and another vertex w_i is $\delta(u, w_i)$ in G. In the 2-hop distance labeling, for every u and v, the distance decoding function is defined to be as follows.

$$F(L(u), L(v)) = \min_{w \in L(u) \cap L(v)} \delta(u, w) + \delta(v, w) \quad (1)$$

Table 1 2-hop distance labels

Vertex	2-hop distance label
v_1	$\{(v_2, 1), (v_3, 1)\}$
v_2	$\{(v_1, 1), (v_3, 1), (v_4, 1), (v_5, 1)\}$
v_3	$\{(v_1, 1), (v_2, 1), (v_4, 1)\}$
v_4	$\{(v_2, 1), (v_5, 1)\}$
v_5	$\{(v_2, 1)\}$
v_6	$\{(v_1, 1), (v_2, 2)\}$
v_7	$\{(v_2, 1)\}$
v_8	$\{(v_5, 1), (v_2, 2)\}$

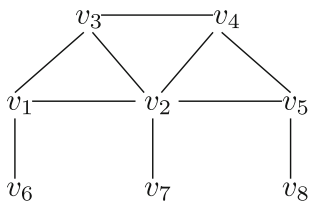

Fig. 1 An example graph G

The 2-hop distance labeling ensures that $F(L(u), L(v)) = \delta(u, v)$. In other words, the distance labeling function ensures that the shortest distance between u and v equals to the shortest distance between u and w_i plus the shortest distance between w_i and v, if there is a vertex w_i which appears in both $L(u)$ and $L(v)$ ($L(u) \cap L(v)$). Otherwise, u and v are not reachable.

Example 1 Table 1 shows the 2-hop distance labels for the graph G in Fig. 1. The shortest distance between v_6 and v_8 is $F(L(v_6), L(v_8)) = \delta(v_6, v_2) + \delta(v_8, v_2) = 4$.

The framework of 2-hop distance labeling approach consists of two phases: an off-line preprocessing phase that generates 2-hop distance labels for all vertices in G, and an on-line querying phase that computes shortest distances for (u, v) queries. The off-line preprocessing phase is only done once. The distance labels are used to answer any (u, v) queries.

Problem statement Our problem is to efficiently compute distance labels for all vertices with small space where possible, for a given graph $G(V, E)$ in a preprocessing step, in order to compute $F(\cdot)$ for answering shortest distance queries for any two vertices u and v in G online.

Below we discuss our techniques for an unweighted and undirected graph. Our techniques can be easily applied to weighted undirected graphs and weighted directed graphs (see Sect. 6).

3 Multi-hop distance labeling

Different from the 2-hop distance labeling, which directly generates 2-hop distance labels for all vertices to answer any possible (u, v) queries based on Eq. (1), in this paper, we propose a multi-hop distance labeling. In other words, we do not generate all 2-hop distance labels for all vertices like [7] in the preprocessing step, which is time-consuming even for moderately sized graphs. Instead, we generate a small subset of 2-hop distance labels, which are our multi-hop distance labels, in the preprocessing step. When answering, we generate sufficient 2-hop distance labels needed on-line for answering a specific (u, v) query efficiently using our multi-hop distance labels and then compute the shortest distance for the (u, v) query. This is motivated by the fact that for a (u, v) query only a small subset of 2-hop distance labels is needed. For example, in order to compute the shortest distance between v_3 and v_6 in Table 1, only the distance labels to v_1 are needed. Therefore, we can generate the sufficient 2-hop distance labels efficiently. Our multi-hop distance labeling is defined as follows.

Definition 2 *Multi-hop distance labeling:* A multi-hop distance labeling of a graph G is a triple $(\mathcal{L}, p, \mathcal{F})$. Here, $\mathcal{L}$ is a *distance labeling function* that assigns a vertex u a label in the form of $\mathcal{L}(u) = \{(w_1, \delta(u, w_1)), \ldots, (w_l, \delta(u, w_l))\}$. p is a *parent function* that assigns a parent vertex to every vertex. $\mathcal{F}$ is a *distance decoding function* that computes shortest distance for a pair of vertices (u, v) using the labels in a way such as $\mathcal{F}(\mathcal{L}(u), p(u), \mathcal{L}(v), p(v)) = \delta(u, v)$.

The main difference between 2-hop and multi-hop is that multi-hop relaxes the condition on $\mathcal{L}(u)$ and $\mathcal{L}(v)$ and does not request $\delta(u, v) = \min_{w \in \mathcal{L}(u) \cap \mathcal{L}(v)} \delta(u, w) + \delta(v, w)$ like Eq. (1). The main advantage is that we can generate the multi-hop distance labels efficiently while avoiding precomputing all-pairs shortest paths, because there is no need that all shortest paths are covered by $\mathcal{L}(\cdot)$. However, it implies two things. (1) There may not exist common vertices for u and v ($\mathcal{L}(u) \cap \mathcal{L}(v) = \emptyset$). (2) There are common vertices for u and v ($\mathcal{L}(u) \cap \mathcal{L}(v) \neq \emptyset$), but $\delta(u, v) = \min_{w \in \mathcal{L}(u) \cap \mathcal{L}(v)} \delta(u, w) + \delta(v, w)$ does no longer hold. In order to turn our multi-hop into a distance labeling, the parent functions play a very important role. The parent functions on-line determine a small subset of vertices in G for the two vertices u and v, denoted as S_{uv}, such that $\delta(u, v) = \min_{w \in S_{uv}} \delta(u, w) + \delta(v, w)$.

We explain the main idea behind our multi-hop distance labeling using an example. Consider the example graph in Fig. 1. Table 2 shows our multi-hop distance labels. We call a vertex v *ancestor* of another vertex u, if and only if there is a sequence of parent relationships $(w_1, w_2, \ldots, w_l)$ from u to v such that $u = w_1$ and $v = w_l$, and $p(w_i) = w_{i+1}$ for $1 \leq i < l$. Here, u is a *descendant* of v if v is an ancestor of u. There exists one and only one vertex (the

Table 2 Multi-hop distance labels

Vertex	Distance label	Parent
v_1	$\{(v_2, 1), (v_3, 1)\}$	v_2
v_2	$\{(v_3, 1), (v_4, 1)\}$	v_3
v_3	$\{(v_4, 1)\}$	v_4
v_4	$\emptyset$	$\emptyset$
v_5	$\{(v_2, 1), (v_4, 1)\}$	v_2
v_6	$\{(v_1, 1)\}$	v_1
v_7	$\{(v_2, 1)\}$	v_2
v_8	$\{(v_5, 1)\}$	v_5

root) having $p(v) = \emptyset$ in our multi-hop distance labeling. First, consider answering a (v_6, v_3) query. Here, v_3 is an ancestor of v_6 because of the existence of the sequence of (v_6, v_1, v_2, v_3). The distance labels of v_6 contain only $\mathcal{L}(v_6) = \{(v_1, 1)\}$. The distance labels of the parent of v_6 are $\mathcal{L}(v_1) = \{(v_2, 1), (v_3, 1)\}$. The shortest distance between v_6 and v_3 is $\delta(v_6, v_3) = \delta(v_6, v_1) + \delta(v_1, v_3) = 2$, where $\delta(v_6, v_1)$ is encoded in $\mathcal{L}(v_6)$ and $\delta(v_1, v_3)$ is encoded in $\mathcal{L}(v_1)$. Next, consider answering a (v_6, v_7) query. Here, there does not exist an ancestor/descendant relationship between v_6 and v_7. Instead, v_6 and v_7 have a least common ancestor v_2 because the ancestors of v_6 are $(v_1, v_2, \ldots)$ and the ancestors of v_7 are $(v_2, \ldots)$. Via the least common ancestor v_2, the shortest distance between v_6 and v_7 is $\delta(v_6, v_7) = \delta(v_6, v_2) + \delta(v_7, v_2) = 3$, because $\delta(v_6, v_2) = \delta(v_6, v_1) + \delta(v_1, v_2) = 2$ and $\delta(v_7, v_2) = 1$. All the distance labels needed are encoded in v_6, v_7, v_1, and v_2, and they are identified by the parent function.

3.1 Vertex separator, tree decomposition, and 2-hop distance labeling

Cohen et al. [7] prove that the problem of computing an optimal 2-hop distance labeling is NP-hard, and propose an approximate algorithm. The approximate algorithm computes all-pairs shortest paths, and then reduces the problem to a set cover problem. However, both steps are time-consuming even for moderately sized graphs. In this paper, we propose a new approach to generate 2-hop distance labels. We discuss several issues that are related to our 2-hop distance labeling computing, namely vertex separator and tree decomposition, which then result in our multi-hop distance labeling.

Vertex separator For a graph $G(V, E)$, a subset of vertices $S \subset V$ is a vertex separator if its deletion splits G into multiple connected components. A vertex separator $S \subset V$ is said to be a vertex separator of $u, v \in V$, if u and v are in different connected components by the deletion of the separator S.

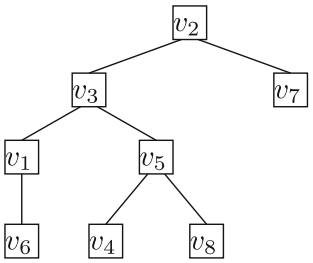

Fig. 2 Example vertex separators

A 2-hop distance labeling can be generated using vertex separators. Given a graph $G(V, E)$, without loss of generality, assume that the deletion of a vertex separator S splits G into two connected components with vertex sets A and B, respectively, that is, $A \cap B = \emptyset$, $A \cup B \cup S = V$, there is no path between any vertex in A and any vertex in B without passing through any vertices in S. We can add labels to vertices as follows. For each vertex $v \in A \cup B$, add $\{(w, \delta(v, w)) \mid w \in S\}$ to $L(v)$, and for each vertex $v \in S$, add $\{(w, \delta(v, w)) \mid w \in S \backslash v\}$ to $L(v)$. Then, we can add labels to vertices recursively by applying the above procedure to the two subgraphs induced by A and B, respectively. It is easy to verify that the resulting labeling is a 2-hop distance labeling.

Example 2 Consider G in Fig. 1. A rooted tree in Fig. 2 shows the vertex separators. Each node in the tree indicates a vertex subset of G. This tree is constructed by a process of finding a vertex separator and dividing the graph recursively. Initially, the vertex separator $\{v_2\}$ is selected to divide G into two connected components and then $\{v_3\}$ is selected to further divide the left components into two connected components $\{v_1, v_6\}$ and $\{v_4, v_5, v_8\}$, for example. Note that a subtree in Fig. 2 represents a connected component of G in Fig. 1. For example, the subtree rooted at v_3 denotes the connected subgraph induced by vertex set $\{v_1, v_3, v_4, v_5, v_6, v_8\}$ in G. Based on this tree, the labels of a vertex u are assigned in a way to include $(w, \delta(u, w))$ for every vertex w in the nodes in the rooted tree that either contain u or are ancestors of the node containing u. For example, $L(v_6) = \{(v_1, 1), (v_3, 2), (v_2, 2)\}$, $L(v_8) = \{(v_5, 1), (v_3, 2), (v_2, 3)\}$. The shortest distance between v_6 and v_8 can be computed as $\min\{\delta(v_6, v_3) + \delta(v_8, v_3), \delta(v_6, v_2) + \delta(v_8, v_2)\}$, by Eq. (1).

It is important to note that we illustrate a method to compute 2-hop distance labels using vertex separators which is hard to compute in general. Next, we show tree decomposition that assists us to compute vertex separators.

Tree decomposition [22] A tree decomposition of a graph $G(V, E)$ is a pair $(\{X_i \mid i \in I\}, T)$, where $T = (I, F)$ is a tree (I are the set of nodes and F are the set of tree edges) and $\{X_i \mid i \in I\}$ is a collection of subsets of V such that:

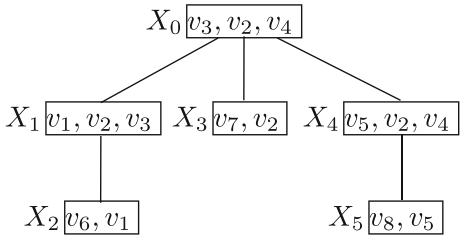

Fig. 3 Example tree decomposition

(1) $\bigcup_{i \in I} X_i = V$. (2) For every $(u, v) \in E$, there is an $i \in I$, s.t., $u \in X_i$ and $v \in X_i$. (3) For every $v \in V$, the set $I_v = \{i \mid v \in X_i\}$ forms a connected subtree of T. Here, a tree node $i \in I$ (or equivalently X_i) represents a subset of vertices in V, that is, $X_i \subset V$, and F represents the set of edges of the tree decomposition. The 1st condition requires that every vertex in V must occur in at least one tree node, and it possibly appears in multiple tree nodes. The 2nd condition requires that for every edge in E, both vertices of its end points must occur together in at least one tree node. The 3rd condition requires that, for every vertex $v \in V$, all the tree nodes that contain v must be connected. This is known to be the *continuity condition* and can be replaced by the following equivalent condition: for all $i_0, i_1, i_2 \in I$, if i_1 is on the undirected path from i_0 to i_2 in T, then $X_{i_0} \cap X_{i_2} \subseteq X_{i_1}$.

The width of a tree decomposition $(\{X_i \mid i \in I\}, T = (I, F))$ is $\max_{i \in I} |X_i| - 1$. The treewidth of a graph G, denoted as $tw(G)$, is the minimum width among all tree decompositions of G.

Example 3 Figure 3 shows a tree decomposition for the graph G shown in Fig. 1, where X_i indicates tree nodes, the vertices enclosed in a rectangle indicate the content of a tree node, for example, $X_1 = \{v_1, v_2, v_3\}$. For the 1st condition, $\bigcup_{0 \le i \le 5} X_i = V$. For the 2nd condition, for example, for edge $(v_3, v_4) \in E$, both vertices are in X_0; for edge $(v_2, v_3) \in E$, both vertices are in X_0 and X_1. For the 3rd condition, for example, v_2 appears in $\{X_0, X_1, X_3, X_4\}$ which forms a connected subtree in T. Or equivalently, $X_3 \cap X_4 = \{v_2\}$ and $\{v_2\}$ are subset of all tree nodes on the path from X_3 to X_4, that is, $\{v_2\} \subset X_0$. The width of this decomposition is 2. The treewidth of G shown in Fig. 1 is 2, that is, $tw(G) = 2$.

From now on, we refer to a vertex in graph G as *vertex* and refer to a node in tree decomposition as *node*. We refer to both i and X_i as tree node. For simplicity, we also call the resulting tree T in a tree decomposition as tree decomposition, when $\{X_i \mid i \in I\}$ can be inferred from the context. The tree T in a tree decomposition is undirected. We can choose an arbitrary node to make T directed.

A tree decomposition $T(I, F)$ is *minimal* if, for all $i, j \in I$, $X_i \nsubseteq X_j$ and $X_j \nsubseteq X_i$. We can transform a non-minimal tree decomposition into a minimal one by deleting nodes. The idea is that, if a $T(I, F)$ is not minimal, there must exist at least one pair (i, j) with $i, j \in I$ and $(i, j) \in F$ such that $X_i \subset X_j$, then we can delete i from I and connect all the neighbors of i (except j) to be reconnected to j. The deletion step continues until the resulting tree decomposition is minimal.

Below, we give a lemma to show the relationships between X_i in the tree decomposition and the vertex separator.

Lemma 1 *Given a graph G and a minimal tree decomposition $(\{X_i \mid i \in I\}, T(I, F))$ of G, an arbitrary non-leaf tree node X_i is a vertex separator of G, and the resulting subtrees by excluding vertices from X_i are also tree decompositions of connected components.*

Proof sketch Without loss of generality, we assume that the deletion of a non-leaf node X_i splits T into two connected subtrees T_1 and T_2.

We first prove that X_i is a vertex separator. Let A be the set of vertices contained in nodes of T_1 excluding vertices in X_i, and B be the set of vertices contained in nodes of T_2 excluding vertices in X_i, that is, $A = \bigcup_{j \in T_1} X_j \backslash X_i$ and $B = \bigcup_{j \in T_2} X_j \backslash X_i$. Then $A \cap B = \emptyset$ based on the third condition of tree decomposition, and $A \cup B \cup X_i = V$. For any vertex $u \in A$ and any vertex $v \in B$, $(u, v) \notin E$ based on the second condition of tree decomposition. Therefore, X_i is a vertex separator of G.

Now, we prove that T_1 and T_2 by excluding vertices from X_i are tree decompositions of the corresponding connected components. Let T_1' and T_2' be the results of deleting vertices of X_i from nodes in T_1 and T_2, respectively, and let X_j' be the corresponding resulting vertices in nodes. Then, we have $X_j' \neq \emptyset$ for any $j \in T_1 \cup T_2$, because T is a minimal tree decomposition. In the previous paragraph, we have shown that X_i separates G into disjoint sets A and B. Now, we only need to show that T_1' and T_2' are tree decomposition of the subgraphs induced by A and B, respectively. Note that, the subgraph induced by A or B may be disconnected. First, $A = \bigcup_{j \in T_1} X_j \backslash X_i = \bigcup_{j \in T_1} X_j'$. Second, for any $u, v \in A$ with $(u, v) \in E$, there must exist a $j \in T_1$ such that $u, v \in X_j$, because $u, v \notin X_i$, we have $u, v \in X_j'$. Third, for any $v \in A$, let T_v' be the set of nodes that contain v, we have $T_v' = T_v$, therefore T_v' forms a connected subtree of T_1'. So, T_1' is a tree decomposition of the subgraph induced by vertex set A. Similarly, T_2' is a tree decomposition of the subgraph induced by vertex set B. □

As an example, the tree node $X_0 = \{v_3, v_2, v_4\}$ in Fig. 3 is a vertex separator of the graph G in Fig. 1. Next we show that 2-hop distance labels can be generated from a tree decomposition of the given graph.

Lemma 2 *2-hop distance labels for G can be generated using a tree decomposition of G.*

Proof sketch This proof sketch is also a construction algorithm to generate 2-hop distance labels based on a tree

decomposition. It works recursively. Initially, the labels of all vertices are empty, that is, $L(v) = \emptyset$, for all $v \in V$.

We consider a graph $G(V, E)$ and its tree decomposition $(\{X_i \mid i \in I\}, T(I, F))$. First, if the tree decomposition is not minimal, we transform it into a minimal one. If the tree decomposition contains no more than two nodes, then for each $v \in V$, add labels $\{(w, \delta(v, w)) \mid w \in V \backslash v\}$ into $L(v)$. Otherwise, each non-leaf node is a vertex separator based on Lemma 1, we can choose an arbitrary one. Assume X_i is chosen as the vertex separator, then, for each $v \in V \backslash X_i$, add labels $\{(w, \delta(v, w)) \mid w \in X_i\}$ into $L(v)$, for each $v \in X_i$, add labels $\{(w, \delta(v, w)) \mid w \in X_i \backslash w\}$ into $L(v)$. Based on Lemma 1, the deletion of node X_i splits T into several subtrees, each of which is a tree decomposition of the corresponding connected component of G by deleting X_i. This process continues for each subgraph and its corresponding tree decomposition. □

Lemma 2 shows how to generate 2-hop distance labels using minimal tree decomposition. Related to our multi-hop distance labeling, we show a specific 2-hop distance labeling for $G(V, E)$. Our specific 2-hop distance labeling can be constructed using any tree decomposition, which does not need to be a minimal tree decomposition. We will discuss it below. We consider the tree decomposition T as a rooted tree rooting at an arbitrary tree node. For example, X_0 is selected as the root for the tree decomposition in Fig. 3. For each vertex $v \in V$, we define r_v as the root node index of the subtree in T induced by I_v $(= \{i \mid v \in X_i\})$, and define X_{r_v} as the actual node, that is, X_{r_v} is the node closest to the root of T among all the nodes containing v. For example, in Fig. 3, $r_{v_1} = 1, r_{v_2} = 0$, and $r_{v_8} = 5$. Let $Ans(i)$ denote the set of indexes of ancestor nodes of X_i including X_i itself. For example, $Ans(1) = \{0, 1\}$ and $Ans(5) = \{0, 4, 5\}$. We assign a label to every vertex $v \in V$ as $L(v) = \{(w, \delta(v, w)) \mid w \in \bigcup_{i \in Ans(r_v)} X_i \backslash v\}$. In other words, we maintain the distance from v to all vertices in tree node X_{r_v} and ancestors of X_{r_v} as labels of v. For example, $L(v_1) = \{(v_2, 1), (v_3, 1), (v_4, 2)\}$, and $L(v_8) = \{(v_2, 2), (v_3, 3), (v_4, 2), (v_5, 1)\}$.

Theorem 1 *For a tree decomposition* $(\{X_i \mid i \in I\}, T = (I, F))$ *of a graph* $G = (V, E)$*, if* $L(v)$ *is generated as* $\{(w, \delta(v, w)) \mid w \in \bigcup_{i \in Ans(r_v)} X_i \backslash v\}$ *for every* $v \in V$*, then it is a 2-hop distance labeling of* G*.*

Proof sketch This labeling can be derived by the procedure in the proof of Lemma 2, where the root node of tree decomposition is chosen as a vertex separator in every recursive step. □

3.2 Multi-hop distance labels

Theorem 1 proposes one method to compute 2-hop distance labeling based on tree decomposition, but does not guarantee the approximate ratio of the 2-hop distance labels generated to the optimal. The size of labels computed based on Theorem 1 can be very large. We propose a new multi-hop distance labeling approach, which stores only a subset of the 2-hop distance labels. We compute multi-hop distance labels based on the 2-hop distance labeling computed by Theorem 1, but do not directly compute 2-hop distance labels.

In multi-hop distance labeling, we define $\mathcal{L}(v) = \{(w, \delta(v, w)) \mid w \in X_{r_v} \backslash v\}$. Recall that X_{r_v} is the specific node in T that contains vertex v and is closest to the root of T among all the nodes containing v. We define the parent of v, $p(v)$, as the parent node of X_{r_v} in T. More precisely, let $r_{v_i} = i$, a tree decomposition can be rewritten as $(\{X_i \mid i \in I\}, T = (I, F))$, where $I = \{1, \ldots, n\}$, $X_i = \{v_i\} \cup \{w \mid w \in \mathcal{L}(v_i)\}$, and $F = \{(i, j) \mid v_i \in V, p(v_i) = v_j\}$.

Given multi-hop distance labeling (Definition 2) with $\mathcal{L}(\cdot)$ and $p(\cdot)$, we explain how the multi-hop distance labeling is used for answering distance queries and how the multi-hop distance labeling can be small in size. All the issues are closely related to the parent function $p(\cdot)$.

In brief, let u and v be two vertices in G, and X_{r_u} and X_{r_v} be two nodes in T. The parent function $u = p(v)$ is designed to indicate the parent relationship between u and v as well as the fact that X_{r_u} is the parent node of X_{r_v} in T. The idea behind is to use such $p(v)$ to trace distance labels online instead of maintaining all the required distance labels. To ensure such correspondence, we restructure T in a way that $r_u \neq r_v$ for any $u \neq v$, or in other words, r_v is unique for a specific vertex v in V. T can be restructured as follows. For a tree decomposition, if there are two vertices u and v with $r_u = r_v$, we create a new node X in T to be the child of the parent of X_{r_u} and to be the parent of X_{r_u}. By letting $X = X_v \backslash v$, we assign the new X to u and the old X_v to v. We can repeat this process until all vertices in G correspond to a unique node in T. In this way, a vertex u is an ancestor of a vertex v in G if and only if X_{r_u} is an ancestor of X_{r_v} in T. It is important to note that the restructuring of T does not change the properties of T.

When querying (u, v), we use both $\mathcal{L}(\cdot)$ and $p(\cdot)$ to generate the query-specific small set of necessary 2-hop distance labels for computing the shortest distance between vertex u and v on-line. We explain query answering using Fig. 4. Here, X_i and X_j in the tree decomposition T are for two vertices, v_i and v_j, in G, respectively. X_l is for a vertex v_l in G which is the least common ancestor of X_i and X_j in T. (Case-1)In order to compute a distance query (v_i, v_l) on G, we only need a linear scan of the multi-hop distance labels of $\mathcal{L}(v_i), \ldots, \mathcal{L}(v_l)$ as defined by $p(.)$ along the path of $X_i, \ldots, X_l$ in T. There are only 4 nodes in T from X_i to X_l. Therefore, we only need to use 4 multi-hop distance labels at most. (Case-2) In order to compute another distance query (v_i, v_j) on G, let S be the vertices of G in X_l, $\delta(v_i, v_j)$ is equal to $\min_{w \in S} \delta(v_i, w) + \delta(v_j, w)$. We can compute the

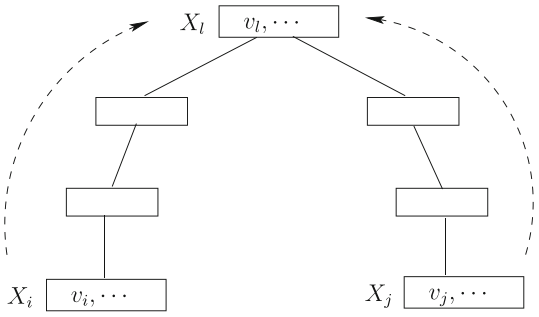

Fig. 4 Query answering

shortest distance between v_i and w, and between v_j and w, which are the necessary 2-hop distance labels to compute the shortest distance between v_i and v_j.

We give two lemmas below. Lemmas 3 and 4 explain the correctness of Case-1 and Case-2, respectively, based on vertex separators.

Lemma 3 *In the rooted tree decomposition T, for any two nodes X_i and X_l where X_l is an ancestor of X_i, consider the path $P(X_i, X_l)$ connecting X_i and X_l in T. For each $X_k \in P(X_i, X_l)$, if $v_l \notin X_k$, then $X_k \backslash v_k$ is a vertex separator of v_i and v_l.*

Proof sketch We have $v_i \notin X_k$ for $X_k \in P(X_i, X_l)$ and $k \neq i$, due to the fact that X_l is an ancestor of X_i. Therefore, for $X_k \in P(X_i, X_l)$ and $k \neq i$, if $v_l \notin X_k$, then X_k is a vertex separator of v_i and v_l based on Lemma 1. Due to the second condition of tree decomposition, v_k is not adjacent to any vertex contained in nodes in the path from X_k to X_l except those vertices contained in $X_k \backslash v_k$. Therefore, $X_k \backslash v_k$ is a vertex separator of v_i and v_l. Similarly, $X_i \backslash v_i$ is a vertex separator of v_i and v_l if $v_l \notin X_i$. □

Lemma 4 *In the rooted tree decomposition T, for any two nodes X_i and X_j, where X_i and X_j do not have ancestor–descendant relationships, Let $P(X_i, X_j)$ be the path connecting X_i and X_j, and let X_l be the least common ancestor of X_i and X_j. For each $X_k \in P(X_i, X_j)$ with $k \neq l$, $X_k \backslash v_k$ is a vertex separator of v_i and v_j. X_l is also a vertex separator of v_i and v_j.*

Proof sketch Due to our construction algorithm of T, for each $X_k \in P(X_i, X_j)$ with $k \neq i$ and $k \neq j$, we have $v_i, v_j \notin X_k$. Therefore X_k with $k \neq i$ and $k \neq j$ is a vertex separator of v_i and v_j. Because X_i and X_j do not have ancestor–descendant relationships, we have $i \neq l$ and $j \neq l$. Similar as the proof of Lemma 3, each $X_k \backslash v_k$ with $k \neq l$ is a vertex separator of v_i and v_j because we have $v_i \notin X_j$ and $v_j \notin X_i$. □

Theorem 2 *For a tree decomposition $(\{X_i \mid i \in I\}, T = (I, F))$ of a graph $G = (V, E)$, if we define $\mathcal{L}(v) = \{(w, \delta(v, w)) \mid w \in X_{r_v} \backslash v\}$ for each $v \in V$, and $p(v)$ as the parent node of X_{r_v} in T, then the multi-hop distance labeling, $(\mathcal{L}, p, \mathcal{F})$, of G, correctly computes the shortest distance for every two vertices in G.*

Proof sketch Given two vertices u and v in G. Suppose X_{r_u} and X_{r_v} are two nodes in the corresponding tree decomposition T that u and v correspond to. There are only two cases, namely Case-1 and Case-2 as discussed above. They can be proved using Lemmas 3 and 4. □

Lemma 5 *The distance labels in our multi-hop distance labeling are strictly smaller than that in the 2-hop distance labeling given by Theorem 1.*

Proof sketch In multi-hop distance labeling, for each vertex $v \in G$, we only maintain the shortest distances from v to vertices in X_{r_v}, that is, $\mathcal{L}(v) = \{(w, \delta(v, w) \mid w \in X_{r_v} \backslash v\}$. While in 2-hop distance labeling, it needs to maintain the shortest distances from v to vertices not only in X_{r_v} but also in the ancestor tree nodes of X_{r_v}, that is, $L(v) = \{(w, \delta(v, w)) \mid w \in \bigcup_{i \in Ans(r_v)} X_i \backslash v\}$. Therefore, the size of multi-hop distance labeling is strictly smaller than 2-hop distance labeling generated by Theorem 1. □

4 Computing distance labels

Our multi-hop distance labeling is based on a tree decomposition of a graph G [1], with a goal of finding a tree decomposition whose total size is as small as possible. Several upper bound heuristics for determining the treewidth of a graph and finding tree decompositions are surveyed in [4]. A family of heuristic algorithms is based on the concept of *fill-in graph*. Let π denote an elimination order of vertex set V, which defines a total ordering of V, that is, $\pi(u) < \pi(v)$ if and only if u is lower ordered than v with respect to π. Given a graph $G(V, E)$ and an elimination order π, the fill-in graph $H(V_H, E_H)$ of G with respect to π is a super graph of G, such that any higher-ordered neighbors of a vertex are connected to each other, that is, $V_H = V$ and $E_H \supseteq E$, and for any two edges $(u, v) \in E_H$ and $(u, w) \in E_H$ with $\pi(v) > \pi(u)$ and $\pi(w) > \pi(u)$, there must have $(v, w) \in E_H$. Given an elimination order π, it is easy to construct a tree decomposition with treewidth one less than the size of the maximum clique in the fill-in graph of G with respect to π [4]. Our LABELING algorithm is based on the following well-known alternative characterizations of the notion of treewidth.

Theorem 3 [4] *Let $G(V, E)$ be a graph, and let $k \leq n$ be a non-negative integer. The following three are equivalent. First, G has a reewidth at most k. Second, there is an elimination order π, such that the maximum size of a clique of the fill-in graph of G with respect to π is at most $k + 1$. Third,*

Algorithm 1 MIN- DEGREE $(G(V, E))$

Input: A graph $G = (V, E)$.
Output: An elimination order π based on min-degree heuristic.

1: $H \leftarrow G$;
2: **for** $i \leftarrow 1$ **to** n **do**
3: Let v be the vertex in H that has minimum degree;
4: Add v to be the ith vertex in ordering π;
5: Add edges to H to make all neighbors of v to be connected to each other, and then remove v and its associated edges;
6: **return** π;

Algorithm 2 LABELING (G, π)

Input: A graph $G = (V, E)$, and an elimination order π.
Output: Distance labels and parents assigned to each vertex.

1: Let $H(V_H, E_H)$ be the fill-in graph of G with respect to π;
2: **for** $i \leftarrow n$ **down to** 1 **do**
3: Let $v \leftarrow \pi^{-1}(i)$ be the ith vertex in ordering π;
4: Let C be the set of higher-ordered neighbors of v in H, i.e., $C = \{u \mid (u, v) \in E_H, \pi(u) > \pi(v)\}$;
5: Assign vertices in C and their corresponding shortest distances from v computed in G to $\mathcal{L}(v)$, i.e., $\mathcal{L}(v) = \{(w, \delta(v, w)) \mid w \in C\}$;
6: Set the parent of v to be the lowest ordered vertex in C, i.e., $p(v) = \arg\min_{u \in C} \pi(u)$;
7: **return** $\mathcal{L}(v)$, $p(v)$ for each vertex in V;

there is an elimination order π, such that no vertex $v \in V$ has more than k higher-ordered neighbors in the fill-in graph of G with respect to π.

An approximate tree decomposition can be found by choosing a specific elimination order heuristically. Min-degree is such a heuristics that finds an approximate tree decomposition efficiently [4]. It works iteratively. In every iteration, it locates the vertex v with the minimum degree and adds v to be the next vertex in ordering π and then makes the neighborhood of v to be a clique and removes v. The pseudo-code of finding an elimination order for a graph G based on the min-degree heuristic is shown in Algorithm 1, denoted as MIN- DEGREE.

The pseudocode of generating multi-hop distance labels is shown in Algorithm 2. Given a graph G and an elimination order π, LABELING assigns distance labels and parents to each vertex in V. It first builds the fill-in graph $H(V_H, E_H)$ of G with respect to π (Line 1). Then, for each vertex $v \in V$, it locates the set of higher-ordered neighbors of v in H (Line 4), that is, $C = \{u \mid (u, v) \in E_H, \pi(u) > \pi(v)\}$. Note that the induced subgraph of C in H is a clique, as ensured by the definition of fill-in graph. For each vertex $w \in C$, it assigns w and its distance from v to $\mathcal{L}(v)$ (Line 5), that is, $\mathcal{L}(v) = \{(w, \delta(v, w)) \mid w \in C\}$. Note that, the distance $\delta(v, w)$ is computed in graph G, not in the fill-in graph H. The parent of v is assigned as the lowest ordered vertex in C (Line 6), that is, $p(v) = \arg\min_{u \in C} \pi(u)$, and $p(v)$ is set to

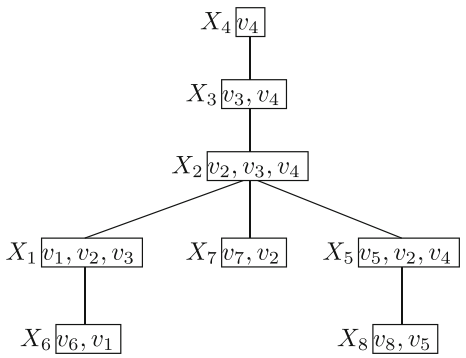

Fig. 5 Example tree decomposition

$\emptyset$ if C is a empty set. Note that, if the graph G is a connected graph, there will be exactly one vertex $v \in V$ with $p(v) = \emptyset$.

Example 4 Consider the graph G shown in Fig. 1, one possible elimination order obtained by min-degree heuristic can be $\pi = \{v_6, v_7, v_8, v_1, v_5, v_2, v_3, v_4\}$. The fill-in graph is G itself, that is, no edge needs to be added. Figure 5 shows a tree decomposition of G. For vertex v_1, the set of higher-ordered neighbors is $C_{v_1} = \{v_2, v_3\}$, therefore, $\mathcal{L}(v_1) = \{(v_2, 1), (v_3, 1)\}$ and $p(v_1) = v_2$. Table 2 shows the multi-hop distance labels corresponding to this tree decomposition.

Reconsider the graph G in Fig. 1. Suppose we delete the two edges (v_2, v_3) and (v_2, v_4) from G and have a different graph denoted as G'. Here, the elimination order and the fill-in graph of G' will be the same as G. However, distance labels for G' can be different from those for G, because they are computed on different graphs.

The tree decomposition of graph G can be constructed by the algorithm LABELING (Algorithm 2). Here, each v and the corresponding C forms a node of the tree decomposition (Lines 3–4). Note that LABELING directly generates a tree decomposition that may not be minimal, as our approach does not require to have a minimal tree decomposition.[1]

Theorem 4 *Algorithm* LABELING *correctly assigns distance labels and parents to vertices in V. The time complexity of* LABELING *is linear to the size of the fill-in graph H, that is, $O(|V_H| + |E_H|)$, and the labeling size is also $O(|V_H| + |E_H|)$.*

Proof sketch Let $C_v = \{u \mid (u, v) \in E_H, \pi(u) > \pi(v)\}$ be the set of higher-ordered neighbors of v in H. We show that $(\{X_i \mid i \in I\}, T(I, F))$ is a tree decomposition of G, where $I = \{1, \ldots, n\}$, $X_i = \{v_i\} \cup C_{v_i}$, and $F = \{(i, j) \mid v_i \in V, p(v_i) = v_j\}$. Consider the first condition of tree decomposition, $\bigcup_{i \in I} X_i = V$, because $I = \{1, \ldots, n\}$ and $v_i \in X_i$. For the second condition, for every $(v_i, v_j) \in E$, it is also

[1] If needed, we can transform the tree decomposition generated by LABELING into a minimal one.

Algorithm 3 SDISTANCE $((u, v))$

1: **if** $v \in \mathcal{L}(u)$ **or** $u \in \mathcal{L}(v)$ **then**
2: **return** $\delta(u, v)$;
3: **if** u is ancestor of v **or** v is ancestor of u **then**
4: **return** SDISTANCEAD(u, v);
5: **else**
6: **return** SDISTANCENAD(u, v);

in E_H, without loss of generality, assume $\pi(v_i) < \pi(v_j)$, then $v_j \in C_{v_i}$, therefore $v_i, v_j \in X_i$. For the third condition, we show that the set $I_v = \{i \mid v \in X_i\}$ forms a connected subtree of T for every $v \in V$. Consider an arbitrary $i \in I_v$, $v \in X_i$, if $v \neq v_i$, then $v \in C_{v_i}$, let $p(v_i)$ be v_j where $v_j \in C_{v_i}$, then either $v = v_j$ or $v \in C_{v_j}$ because C_{v_i} induces a clique in H, therefore $j \in I_v$, also we have $\pi(v_j) > \pi(v_i)$. There is exactly one $i \in I_v$ with $v_i = v$, therefore, I_v forms a connected subtree of T. Let such $i \in I$, where $p(v_i) = \emptyset$, be the root of T, then $r_{v_i} = i$ for all $v_i \in V$. Because $X_i = \{v_i\} \cup C_{v_i}$, we have $\mathcal{L}(v) = \{(w, \delta(v, w)) \mid w \in C_v\} = \{(w, \delta(v, w)) \mid w \in X_{r_v} \backslash v\}$, and $p(v)$ just record that the parent of X_{r_v} is $X_{r_{p(v)}}$ in the rooted tree.

Now, we show the time complexity of LABELING. Let $d(v)$ denote the degree of a vertex $v \in V$. Line 3 takes $O(1)$ time, and Line 4 takes $O(d(v))$ time, because we just need to loop through all the neighbors of v in H and record the higher-ordered neighbors in C, and Line 5 and Line 6 take $O(|C|)$ time that is less than $O(d(v))$. Note that, at Line 5, we do not include the time to compute shortest distances $\delta(v, w)$. We need to execute Lines 3–6 for all n vertices, so the total time complexity is $\sum_{v \in V} O(d(v)) = O(|V_H| + |E_H|)$. Because we add labels and parents at Line 5 and Line 6, respectively, the total label size is $\sum_{v \in V} O(|C_v| + 1) \leq \sum_{v \in V} O(d(v) + 1) = O(|V_H| + |E_H|)$. □

It is worth noting that Theorem 4 states that the time complexity of LABELING is linear to the size of the fill-in graph H. Here, the time to find such a fill-in graph and the time to compute shortest paths between vertex pairs are not taken into consideration. If all these computations are taken into consideration, the worst case time complexity is $O(|V|(|V| + |E|))$.

5 Shortest distance query

Algorithm 3 shows the main algorithm to compute the shortest distance between u and v. First, if v (or u) is in the distance labels of u (or v) (line 1), then $\delta(u, v)$ is maintained in the corresponding distance labels. If u and v have ancestor–descendant relationships, then we call the procedure SDISTANCEAD (Algorithm 4) to compute shortest distances, otherwise, we call the procedure SDISTANCENAD (Algorithm 5).

Ancestor–descendant queries Consider a distance query (u, v), assuming that there exists an ancestor–descendant relationship between u and v. Also assume that v is an ancestor of u. We compute shortest distance from u to v. SDISTANCEAD (Algorithm 4) is based on Lemma 3 (in Sect. 3.2). It first initializes $dis(w)$ to be $\delta(u, w)$ for all $w \in \mathcal{L}(u)$ (Line 1). Then, it computes distances for vertices along the path from X_{r_u} to X_{r_v} (Lines 3–7). Let X_{r_c} be the current node in considering. For every vertex $w \in \mathcal{L}(c)$, $dis(w)$ is updated to be $dis(c) + \delta(c, w)$, if $dis(w)$ is not computed previously or $dis(c) + \delta(c, w) < dis(w)$ (Lines 4–6). Finally, we check whether there exists any shorter path from u to v that goes through vertices in $\mathcal{L}(v)$ (Lines 8–10).

Let tw denote the treewidth of the tree decomposition.

Algorithm 4 SDISTANCEAD (u, v)

Input: Two vertices u and v, where v is an ancestor of u.
Output: Shortest distance between u and v, $\delta(u, v)$.

1: Initialize $dis(w) = \delta(u, w)$, for all $w \in \mathcal{L}(u)$;
2: Let $c \leftarrow p(u)$;
3: **while** $c \neq v$ **do**
4: **for all** $w \in \mathcal{L}(c)$ **do**
5: **if** $dis(w)$ is not computed **or** $dis(c) + \delta(c, w) < dis(w)$ **then**
6: $dis(w) \leftarrow dis(c) + \delta(c, w)$;
7: $c \leftarrow p(c)$;
8: **for all** $w \in \mathcal{L}(v)$ **do**
9: **if** $dis(w)$ is computed **and** $dis(w) + \delta(v, w) < dis(v)$ **then**
10: $dis(v) \leftarrow dis(w) + \delta(v, w)$;
11: **return** $dis(v)$;

Theorem 5 SDISTANCEAD *correctly computes shortest distance* $\delta(u, v)$*, when* v *is an ancestor of* u *in the multi-hop distance labeling. The time complexity of* SDISTANCEAD *is* $O(tw \cdot |P(X_{r_u}, X_{r_v})|)$.

Proof sketch We first prove that, for every $dis(w)$ computed by SDISTANCEAD, there is a corresponding path from u to w that has distance $dis(w)$. There are three statements (Lines 1, 6, 10) that change the values of $dis(w)$. We prove it for each case. For the first case (Line 1), $dis(w)$ is initialized as stored in the distance labels, therefore, the claim is satisfied as stated in Theorem 4. For the second case (Line 6), we have that vertex $p(c)$ is included in the distance labels of c for any c, as ensured by Line 6 of Algorithm 2. Therefore, during the while loop (Line 3), $dis(c)$ is initialized with a correct value. For $dis(w)$ which is assigned as $dis(c) + \delta(c, w)$, it satisfies the claim since both $dis(c)$ and $\delta(c, w)$ correspond to paths. For the third case (Line 10), the proof is similar to that for the second case. Therefore, the claim is correct, and $dis(w)$ is an upper bound of the shortest distance from u to w.

Now, we show that the shortest path between u and v has been considered by the algorithm when it terminates. Assume the shortest path between u and v is $w_1, \ldots, w_l$ with $w_1 = u$ and $w_l = v$. Because $X_{r_u} \backslash u$ is a vertex separator of u and v (based on Lemma 3), the path must go through a vertex in $X_{r_u} \backslash u$. Assume w_i is such a vertex in $X_{r_u} \backslash u$ that has the largest shortest distance from u among all the vertices on the

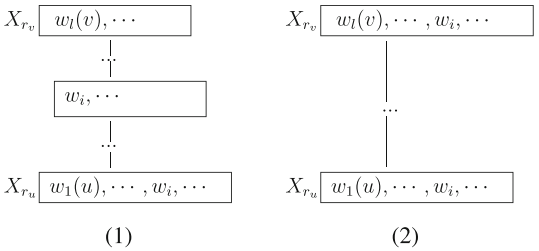

Fig. 6 Illustration of shortest path

Algorithm 5 SDISTANCENAD (u, v)

Input: Two vertices u and v, where u and v do not have ancestor–descendant relationships.
Output: Shortest distance between u and v, $\delta(u, v)$.

1: Find the vertex separator S of u and v;
2: $dis_u(S) \leftarrow$ SDISTANCEAD- LIST(u, S);
3: $dis_v(S) \leftarrow$ SDISTANCEAD- LIST(v, S);
4: $\delta(u, v) \leftarrow \min_{w \in S} dis_u(w) + dis_v(w)$;
5: **return** $\delta(u, v)$;

shortest path, then the path $w_1, \ldots, w_i$ is considered in the distance labels $\mathcal{L}(u)$. If $X_{r_{w_i}}$ is not on the path $P(X_{r_u}, X_{r_v})$, then w_i is contained in the distance labels of every node on $P(X_{r_u}, X_{r_v})$ due to the third condition of tree decomposition, therefore the path $w_i, \ldots, w_l$ is checked in the labels $\mathcal{L}(v)$ (Lines 8–10), as illustrated in Fig. 6(2). Otherwise, $X_{r_{w_i}}$ is on the path $P(X_{r_u}, X_{r_v})$ (Fig. 6(1)), then the path $w_i, \ldots, w_l$ is checked inductively as proved above. And, the path will be found since it becomes shorter by every induction.

For the time complexity, it is easy to see that, if we use an array dis of size n to store distances and initialize $dis(w) \leftarrow \infty$ for all $w \in V$, then SDISTANCEAD takes $O(tw \cdot |P(X_{r_u}, X_{r_v})|)$ time excluding the initialization time. Here, we show an implementation of SDISTANCEAD that uses only $O(tw)$ working memory space and runs in time $O(tw \cdot |P(X_{r_u}, X_{r_v})|)$. We notice that, assume $P(X_{r_u}, X_{r_v}) = Y_1, \ldots, Y_l$, then during the execution of algorithm, only the distances for vertices contained in two consecutive nodes are needed for computation and storage. Therefore, we record vertex-distance pairs in dis for vertices in consideration, and the entries in dis are ordered by vertex id. Similarly, we also assume distance labels $\mathcal{L}(v)$ are sorted by vertex id. Therefore, Lines 4–6 and Lines 8–10 can be implemented in a merge-sort fashion in linear time. □

Non-ancestor–descendant queries Consider a distance query (u, v) and assume u and v do not have ancestor–descendant relationships. Let $lca(u, v)$ denote the least common ancestor of u and v in the distance labeling. We have $lca(u, v) \neq u$ and $lca(u, v) \neq v$. SDISTANCENAD (Algorithm 5) is based on Lemma 4 (in Sect. 3.2). It first finds the vertex separator S of u and v which is chosen between the two children of $lca(u, v)$. Then, it computes

Algorithm 6 SDISTANCEAD- LIST (u, S)

Input: A vertex u and a set of ancestor vertices S.
Output: Shortest distances between u and every vertex in S.

1: Initialize $dis(w) = \delta(u, w)$, for all $w \in \mathcal{L}(u)$;
2: Let $c \leftarrow p(u)$;
3: **while** c is not ancestor of all vertices in S **do**
4: **for all** $w \in \mathcal{L}(c)$ **do**
5: **if** $dis(w)$ is not computed **or** $dis(c) + \delta(c, w) < dis(w)$ **then**
6: $dis(w) \leftarrow dis(c) + \delta(c, w)$;
7: **if** $c \in S$ **then**
8: **for all** $w \in \mathcal{L}(v)$ **do**
9: **if** $dis(w) + \delta(c, w) < dis(c)$ **then**
10: $dis(c) \leftarrow dis(w) + \delta(c, w)$;
11: $c \leftarrow p(c)$;
12: **return** $dis(v), \forall v \in S$;

shortest distance from u and v to all the vertices in S by calling a procedure SDISTANCEAD- LIST (Lines 2–3), where all the vertices in S are ancestors of u and v. Finally, $\delta(u, v)$ is equal to $\min_{w \in S} dis_u(w) + dis_v(w)$ (Line 4).

SDISTANCEAD- LIST computes shortest distances between a vertex and a set of ancestor vertices. A naive implementation is calling the procedure SDISTANCEAD for each $w \in S$. However, it will increase the time complexity by a factor of tw. SDISTANCEAD can be extended to compute shortest distance between u and a set of ancestor vertices S with the same time complexity as SDISTANCEAD. The modifications are as follows. The while loop (Line 3 of Algorithm 4) will not terminate until it reaches the ancestor of all vertices in S, that is, the vertex with the maximum path length in T from X_{r_u}, $\max_{w \in S} |P(X_{r_u}, X_{r_w})|$. And Lines 8–10 of Algorithm 4 are executed every time when c is equal to some vertex in S, and the corresponding distance is recorded as $dis_u(c)$. The pseudocode of SDISTANCEAD- LIST is shown in Algorithm 6.

Algorithms 6 and 4 share similarities. The difference lies in the while loop (Line 3) of Algorithm 6, compared with the while loop (Line 3) of Algorithm 4. In Algorithm 6 it deals with a set of vertices S, and in Algorithm 4 it deals with a single vertex.

Theorem 6 SDISTANCEAD- LIST *correctly computes shortest distances between u and a set of vertices S, where every vertex in S is an ancestor of u in the distance labeling. The time complexity of the algorithm* SDISTANCEAD- LIST *is* $O(tw \cdot \max_{w \in S} |P(X_{r_u}, X_{r_w})|)$.

Proof sketch The correctness directly follows from the above discussions. The time complexity follows from the above discussions and the proof of Theorem 5. □

Theorem 7 SDISTANCENAD *correctly computes shortest distance between u and v, when u and v do not have an ancestor–descendant relationship. The time complexity of* SDISTANCENAD *is* $O(tw \cdot \max_{w \in S}(|P(X_{r_u}, X_{r_w})| + |P(X_{r_v}, X_{r_w})|))$.

Table 3 Execution example

Steps	$prev$	c
Line 1	–	$(v_1, 1)$
(1st) lines 4–6	$(v_1, 1)$	$(v_2, 2), (v_3, 2)$
(2nd) lines 4–6	$(v_2, 2), (v_3, 2)$	$(v_3, 2), (v_4, 3)$
Lines 8–10	$(v_3, 2), (v_4, 3)$	$(v_3, 2)$

Proof sketch The correctness follows from Theorem 6 and the fact that, S is a vertex separator of u and v, and every path from u to v must go through at least one vertex in S. Therefore $\delta(u, v) = \min_{w \in S} \delta(u, w) + \delta(v, w)$.

As stated in Theorem 6, Line 2 has time complexity $O(tw \cdot \max_{w \in S} |P(X_{r_u}, X_{r_w})|)$ and Line 3 has time complexity $O(tw \cdot \max_{w \in S} |P(X_{r_v}, X_{r_w})|)$. In Line 1, the vertex separator S can be found in $O(tw)$ time. Therefore, the time complexity of SDistanceNAD follows. □

Example 5 Consider the multi-hop distance labels in Table 2. The execution steps of SDistanceAD for computing shortest distance between v_6 and v_3 are shown in Table 3, where v_3 is an ancestor of v_6 in the distance labeling. c denote the current vertex in consideration, while $prev$ denote the vertex considered in the previous loop. Initially, it copies $\mathcal{L}(v_6)$. At the second step, $c = v_1$, it is the first time of executing Lines 4-6, $(v_2, dis(v_2))$ and $(v_3, dis(v_3))$ can be computed based on $\mathcal{L}(c)$ and the distances stored in the previous step. At the third step, $c = v_2$, it is the second time of executing Lines 4-6. At the forth step, $c = v_3$, it is also the last step, the shortest distance to v_3 can be computed based on $\mathcal{L}(v_3)$ and the distances stored at the third step. Finally, we correctly get $\delta(v_6, v_3) = dis(v_3) = 2$.

Now, we show how to compute shortest distance between v_6 and v_7. As $lca(6, 7) = 2$, therefore X_2 is a vertex separator of v_6 and v_7, also $X_1 \backslash v_1$ and $X_7 \backslash v_7$ are vertex separators of v_6 and v_7. Because we need to compute shortest distances from u and v to the vertex separator, we choose the vertex separator that will result in less running time. Here, we choose $X_7 \backslash v_7 = \{v_2\}$. Therefore, we need to compute $dis_{v_6}(v_2) = 2$ and $dis_{v_7}(v_2) = 1$, and the shortest distance between v_6 and v_7 is the sum of the two values, that is, $\delta(v_6, v_7) = 3$.

Theorem 8 SDistance *correctly computes shortest distance between vertex pairs based on the multi-hop distance labeling. The time complexity of* SDistance *is* $O(tw \cdot h)$*, where* tw *and* h *are the width and height of the tree decomposition, respectively.*

Proof sketch The correctness of SDistance directly follows from Theorems 5 and 7. As h is the height of the tree decomposition, then for any u and v where v is an ancestor of u, we have $|P(X_{r_u}, X_{r_v})| \leq h$. Therefore, the time complexity of SDistance is $O(tw \cdot h)$. □

6 Extensions

In [29], Wei proposes TEDI to compute distance queries, using a tree decomposition index structure. Our approach is different from TEDI. We store multi-hop distance labels and use different new query answering algorithms. The efficiency of our multi-hop approach is based on Lemmas 3 and 4.

More specifically, in the label construction, for each clique C as defined at Line 4 of Algorithm 2, our approach only needs to store the shortest distance from other vertices in C to v, whereas TEDI has to store the pairwise shortest distances for all vertices in C. In query processing, the time complexity of multi-hop is $O(tw \cdot h)$, whereas TEDI takes $O(tw \cdot tw \cdot h)$ time. This is because, for each tree node C on the undirected path from u to v on the tree decomposition T, multi-hop only needs a linear number of shortest distances with respect to $|C|$ which takes time $O(tw)$, whereas TEDI has to consider a quadratic number of shortest distances which takes time $O(tw \cdot tw)$.

Reducing tree height As shown by Theorems 5 and 7, the time complexity of computing shortest distance depends on both the width tw and the height h of the tree decomposition. For large sparse graphs, the width of a tree decomposition can be small, however, the height may be large, thus the query processing time increases. Here, we show a technique to trade the label size for query processing time, by increasing the label size a little while lowering down the height of tree decomposition.

Lemma 6 *Given a rooted tree decomposition* $T(I, F)$*, if we collapse any connected subtree containing the root node into a single new root node, then it is still a tree decomposition, and the height is lower.*

Proof sketch After merging a connected subtree of T into a single new node, the three conditions of tree decomposition still hold if T is a tree decomposition. The height of tree T gets lower directly following the process. □

Example 6 Figure 7 shows a collapsed tree decomposition of the tree decomposition shown in Fig. 5. It collapses the subtree induced by nodes $\{X_1, X_2, X_3, X_4, X_5\}$ into a single new root node X_r. The height of tree decomposition in Fig. 5 is 5, while the height of the collapsed tree decomposition is 2.

Given a collapsed tree decomposition T_c, we cannot generate multi-hop distance labels and answer queries directly using the previous algorithms, because the width of the collapsed tree decomposition is much larger. Instead, we generate a transitive closure for vertices in the root node of T_c and generate distance labels and parents for vertices in non-root nodes as Theorem 4. For each vertex $v \in V$, if X_{r_v} is the root node or the parent of X_{r_v} is the root node, then $p(v) = \emptyset$. In order to retrieve the shortest distances between two vertices

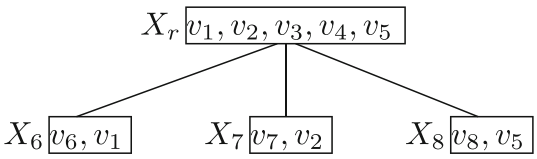

Fig. 7 Collapsed tree decomposition

Algorithm 7 Modification of SDistanceAD

1: Lines 1–2 of Algorithm 4;
2: **while** $c \neq v$ **and** $c \neq \emptyset$ **do**
3: Lines 4–7 or Algorithm 4;
4: **if** $c = v$ **then**
5: Lines 8–10 of Algorithm 4;
6: **else**
7: Let a be the node that is the child of the root on the path from X_{r_u} to the root;
8: $dis(v) \leftarrow \min\{dis(v), \min_{w \in X_{r_a} \setminus a} dis(w) + \delta(w, v)\}$;

in the root node in constant time, we also build an inverted index for the vertices in the root node.

For query processing, only algorithms SDistanceAD and SDistanceAD- List need to be modified. We discuss the modifications of SDistanceAD below, while SDistanceAD- List can be modified similarly. For a distance query (u, v) on T_c where v is an ancestor of u, we modify SDistanceAD as follows. The while loop at Line 3 iterates until $c = v$ or $c = \emptyset$. If $c = v$, then we find the shortest distance to v by Lines 8-10. If $c = \emptyset$, let a be the node that is the child of the root on the path from X_{r_u} to the root and compute $dis(v) = \min_{w \in X_{r_a} \setminus a} dis(w) + \delta(w, v)$, where $\delta(w, v)$ is stored in the transitive closure of vertices in the root node. The pseudocode is given in Algorithm 7.

Example 7 Consider the collapsed tree decomposition in Fig. 7, we show how to compute shortest distance between v_6 and v_7. Assume the vertex separator is chosen as $\{v_1\}$, then $dis_{v_6}(v_1) = 1$ is stored as labels in $\mathcal{L}(v_6)$. In order to compute $dis_{v_7}(v_1)$, first, we get $dis_{v_7}(v_2) = 1$, then $dis_{v_7}(v_1) = dis_{v_7}(v_2) + \delta(v_2, v_1) = 2$, where $\delta(v_2, v_1)$ is stored as transitive closure of vertices in the root node. Therefore, $\delta(v_6, v_7) = dis_{v_6}(v_1) + dis_{v_7}(v_1) = 3$.

Theorem 9 *Given our multi-hop distance labeling based on a collapsed tree decomposition, our algorithm computes shortest distances in time $O(tw' \cdot tw' + tw' \cdot h')$, where tw' is the width of tree decomposition excluding the root node, that is, $\max_{i \in I \setminus r} |X_i| - 1$. Note that $tw' \leq tw$ and $h' \leq h$.*

Proof sketch Given two query vertices u and v, if v is an ancestor of u, then the time complexity is $O(tw' \cdot h')$ as shown in the above discussions. If u and v do not have ancestor–descendant relationships, then we need to compute shortest distance from u and v to a vertex separator S for u and v, where all vertices in S are ancestors of u and v. If all vertices in S are contained in root node, then we need to compute $dis(w) = \min_{x \in X_{r_a} \setminus a} dis(x) + \delta(x, w)$ for all vertices $w \in S$. Therefore, the total time complexity is $O(tw' \cdot tw' + tw' \cdot h')$. □

Budget B Based on Theorem 9, we can reduce the query processing time by generating larger distance labels which also reduces the height of tree decomposition. When the size of root gets larger, the size of distance labels also becomes larger, because we need to store all-pairs shortest distance for vertices in the root node. Given a budget B of the size of labels, we generate a collapsed tree decomposition so that the root node is as large as possible, while the total size of label is below B. We explain it below. Consider Algorithm 2 that generates distance labels, for a specific value i of the "for" loop (Line 2). We know the size of distance labels already generated, and we also know the size of root node if we put all remaining vertices in a single root node. Therefore, given a budget B, we can terminate the "for" loop as long as the calculated total size of distance labels does not exceed B. It can be proved that, when B becomes larger, tw' and h' become smaller, therefore the query processing time gets smaller. It is worth noting that, given a graph, there is a minimal budget required, which is the index size when constructing a complete tree decomposition. If the budget given is smaller than the minimal as required, we ignore the budget parameter and build a complete tree decomposition.

Weighted and directed graph For weighted graphs, the only difference is to compute shortest distances using Dijkstra's algorithm when generating distance labels. For a weighted graph, shortest paths need to be computed using Dijkstra's algorithm instead. Therefore, it needs more time to compute distance labels for weighted graphs. However, the query time remains the same as long as the graph topologies are the same.

For directed graphs, we treat the edges as undirected and compute the fill-in graph and tree decomposition. In other words, for distance labels, we store shortest distances on both directions, that is, in $\mathcal{L}(u)$, we store both the shortest distance from u to v and the shortest distance from v to u. When answering shortest distance queries, we choose the correct directional shortest distances in $\mathcal{L}(u)$ in computation. For example, in Algorithm 4, we consider only the shortest distance from c to w at Lines 5–6 and consider only the shortest distance from w to v at Lines 9–10.

7 Related work

There are works in the literature using tree decomposition to fast shortest distance queries. Chaudhuri and Zaroliagis [5] study the problem of computing shortest paths in digraphs with bounded treewidth, by preprocessing the graph using

tree decomposition. They analyze the problem in theoretical aspect, no implementation issues are considered. Wei [29] proposes TEDI, which uses a tree decomposition index structure, while computing shortest distance in $O(tw^2 \cdot h)$ time. Although the concept of using tree decomposition index to fast compute shortest distance is not new, these existing works cannot be applied to generate distance labels that are discussed in this paper. Furthermore, the existing works have higher query time complexity compared to our algorithm, which is also confirmed in the experiments (Sect. 8).

There are other works proposing index structures for exact shortest distance queries. Xiao et al. [30] propose the concept of compact BFS-trees to index all-pairs shortest paths. First, a BFS-tree is constructed starting from every vertex. Then, the set of BFS-trees are compressed by exploiting the symmetry property of the graph. It is shown in [29] that TEDI outperforms this approach in both index size and query time. Another category of works is based on the concept of 2-hop distance labeling [7], which assigns distance labels to vertices such that, for each pair of vertices u and v, the shortest path is covered by the concatenation of a path in $L(u)$ and another path in $L(v)$. However, in order to generate 2-hop distance labels, it needs to pre-compute all-pairs shortest paths, which is prohibitive in large graphs. Furthermore, give the set of all-pairs shortest paths, generating an optimal 2-hop distance labels is NP-hard [7]. Cheng and Yu [6] propose a heuristic algorithm to generate 2-hop distance labels for directed graphs. It first constructs a DAG subgraph by removing a small set of vertices, then 2-hop distance labels are generated for these two parts, respectively. However, their techniques cannot be applied to undirected graphs. Upper and lower bounds of the 2-hop distance labeling size for several families of graphs are studied theoretically in [11,16,19]. The above-mentioned works are dealing with general graphs. There are other methods designed specifically for road networks [12,23], which make use of the near planarity of road networks or/and the existence of coordinates of nodes in road networks.

Another set of works consider generating sketches for vertices to answer approximate shortest distance queries, termed *distance oracle* [27]. Thorup and Zwick [27] give a method to construct an approximate distance oracle using space $O(c \cdot n^{1+1/c})$, that can answer queries in time $O(c)$ with a distance estimation that is at most $2c - 1$ times larger than the actual shortest distance, for any integer c. Baswana and Sen [3] improve the preprocessing time to $O(n^2)$ time for unweighted graphs. Sommer et al. [26] show a new lower bound for the approximate distance oracles in the cell-probe model for sparse graphs. Sarma et al. [9] simplify the algorithms proposed by Thorup and Zwick while providing the same theoretical guarantee, and experiments are conducted to evaluate their algorithms.

There are some initial works studying shortest path queries with additional constraints recently, which is orthogonal to our problem of general shortest path query. Rice and Tsotras [21] study shortest path queries of road networks with label restrictions, where the label restrictions specify a subset of the graph edges that the shortest path computation can be applied on. They extend the techniques of Contraction Hierarchies [12] to handle labels restrictions when building index. Optimization techniques are also studied in [21]. Liu and Wong [18] study shortest path queries in terrain datasets with slop constraint, where each data point is a three-dimensional point that adds elevation to the traditional two-dimensional data. They propose a new framework called surface simplification, under which the surface is "simplified" such that the complexity of finding shortest paths on this simplified surface is lower. Such techniques cannot be applied to the optimization of shortest path queries on general graphs.

Reachability queries have been extensively studied on large graph data. The theoretical foundation of indexing reachable vertex pairs using 2-hop index is studied by Cohen et al. [7]. Path-tree and 3-hop techniques are proposed by Jin et al. [14,15] to build index for reachability queries more practically. The techniques of path-tree and 3-hop combine the techniques of tree cover, chain cover, and 2-hop cover together. Due to the complex structures of existing techniques, they can only handle moderately sized data. Two recent works study the reachability queries for very large data. Yildirim et al. [31] use the idea of computing several random tree covers and generating a modified interval code for each generated tree cover over a condensed DAG (Directed Acyclic Graph). With these modified interval codes, they can prune non-reachable queries very efficiently. Due to the low complexity of index structure, it can handle directed graphs with hundreds of millions of nodes. However, such a condensed DAG cannot be used for computing shortest paths. Schaik and Moor [28] study a memory efficient data structure to index the reachability information between all pairs of vertices using bit vector compression. The size of bit vector index used is typically small for practical data, even though it can be very large in worst case. The bit vector compression can be also used to compute all the reachability information very efficiently. Although they can handle very large directed graphs, these techniques cannot be applied to shortest path queries in undirected graph. First, in undirected graph, all the vertex pairs are reachable. Second, these techniques do not consider the distance information associated with edges.

8 Experiments

We conduct extensive performance studies to test the efficiency of our multi-hop approach as well as 2-hop approach.

We implement our multi-hop algorithm, denoted as m-hop in this section. For comparison, we also implement the tree-decomposition-based approach TEDI in [29]. All tests are conducted on a PC with an Intel(R) Pentium(R) 2.8 GHz CPU and 2GB memory PC running CentOS 5.4. All algorithms are implemented in C++ and compiled with -O3 optimization.

m-hop consists of two phases: multi-hop distance labels generation and query answering. TEDI also consists of two phases: label construction and query answering. We compare m-hop and TEDI in two aspects, label construction time and query answering time. We also list the query answering time of a naive BFS algorithm, which computes shortest distances on the original graph directly. The query answering time is measured in milliseconds (ms). For each dataset, we randomly generate 10,000 pairs of vertices to issue queries.

8.1 M-hop vs 2-hop

We compare m-hop with 2-hop in the aspects of label construction time, label size, and query time. We implement the 2-hop distance labels construction algorithm given in [7]. The label construction of 2-hop is done on a linux server with 48G memory since it runs out of memory if running on the PC.

We generate a set of small synthetic graphs according to the BA model [2], which is a widely adopted model to simulate real graphs. In the generated graphs, the degrees follow a scale-free power-law distribution. We generate 6 graphs with average degree 2.2, where the number of vertices ranges from 1,000 to 6,000 by a step of 1,000, and denote the graph as xk if it contains $x \cdot 1{,}000$ vertices.

The construction time of 2-hop and m-hop are shown in Fig. 8. For a graph with 6,000 vertices, m-hop takes 0.1 s, whereas 2-hop takes 4.27 h. The 2-hop label construction time increases much faster than that of m-hop, for example, when the graph size increases from 1,000 to 6,000, the m-hop construction time increases 15 times, whereas 2-hop increases 300 times. This is because that, for undirected graphs, 2-hop first computes all-pairs shortest paths and then solves an instance of set cover, where the size of the ground set is quadratic to the number of vertices. The query time and label size of 2-hop and m-hop are shown in Table 4, which shows that our multi-hop distance labels are smaller than 2-hop distance labels. The query processing time of m-hop is fast, but is larger than that of 2-hop.

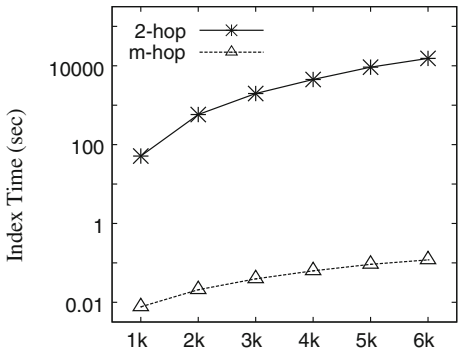

Fig. 8 Label construction time of 2-hop and m-hop

Table 4 Query time and label size of 2-hop and m-hop

Graph	Query time (ms)		Label size (KB)	
	2-hop	m-hop	2-hop	m-hop
1k	0.00013	0.00073	45.1	34.9
2k	0.00014	0.00084	107	78
3k	0.00015	0.00094	176	130
4k	0.00020	0.00105	263	195
5k	0.00025	0.00120	356	240
6k	0.00031	0.00149	452	245

Table 5 Information of real datasets

Graph	n	Average degree	Maximum distance	Median distance
Pfei	1,738	2.16	29	10
PPI	1,458	2.67	19	7
Dutch	3,621	2.38	22	8
Epa	4,253	4.18	10	4
Erdos	6,927	3.42	4	4
Eva	4,475	2.08	18	7
Geom	3,621	5.23	14	5
Cal	5,925	5.32	13	5
Homo	7,020	5.64	14	5
Inter	22,442	4.06	10	4

8.2 Small real datasets

In this subsection, we test m-hop and TEDI over the real datasets used in [29]. The real graphs include biological networks (PPI and Homo), social networks (Pfei, Geom, Erdos, Dutch, and Eva), information networks (Cal and Epa), and technological networks (Inter). All these graphs are provided by the authors of [30]. Sizes and other information of these datasets are shown in Table 5, where maximum distance and median distance are the maximum and median value of all the distances between vertex pairs, respectively.

In TEDI, a parameter d is used to tune the tree decomposition. TEDI generates a tree decomposition, with the size of all nodes except the root node at most d. In order to generate such a tree decomposition, in MIN- DEGREE, the for loop (Line 2) stops immediately when the next minimum degree (Line 3) is larger than d. In this set of tests, we use the same d values as specified in [29]. Our approach uses budget B instead of d. Our budget B is more general than d. Given a d

Table 6 Query time for real datasets

Graph	d	BFS (ms)	TEDI	m-hop
Pfei	6	0.0336	0.0030	0.0010
PPI	7	0.0310	0.0024	0.0010
Dutch	5	0.0634	0.0027	0.0011
Epa	8	0.0724	0.0012	0.0009
Erdos	7	0.0916	0.0013	0.0011
Eva	2	0.0544	0.0021	0.0008
Geom	6	0.0967	0.0023	0.0011
Cal	10	0.1674	0.0021	0.0015
Homo	18	0.2421	0.0033	0.0022
Inter	14	0.6951	0.0032	0.0025

Table 7 Label construction time and index size for real datasets

Graph	Construction time		Index size (KB)	
	m-hop	TEDI	m-hop	TEDI
Pfei	17	19	86	110
PPI	18	22	149	174
Dutch	76	84	718	767
Epa	200	227	3,264	3,343
Erdos	173	227	1,184	1,314
Eva	52	55	94	149
Geom	274	301	3,478	3,551
Cal	692	765	7,974	8,113
Homo	1,320	1,526	15,542	15,752
Inter	3,725	5,238	5,865	6,520

value, we can compute a budget B. However, a budget B we use can be smaller than the corresponding budget computed from d.

The query time (ms) of BFS, TEDI, and m-hop, are shown in Table 6. From Table 6, we can see that the query processing time of BFS increases dramatically when the graph size increases, because the time complexity of BFS is linear to the graph size. For m-hop, the query processing time changes very slowly when the graph size increases, for example, the query time on Pfei is 0.001 ms, and it only increases to 0.0025 ms on Inter, whose size is 13 times larger than that of Pfei. The speedups of m-hop and TEDI with respect to BFS are shown in Fig. 9. m-hop consistently outperforms TEDI in all these real graphs. This is confirmed by the time complexity of these two algorithms. The time complexity of our algorithm is $O(tw \cdot (tw + h))$, while the time complexity of TEDI is $O(tw \cdot tw \cdot h)$.

The label construction time and index size of TEDI and m-hop are shown in Table 7, where the label construction time includes the time to build a tree decomposition, and the time to compute shortest paths for vertices in nodes. Our algorithm takes less time to generate distance labels. Because, for all vertices in a node in the tree decomposition, it needs to compute all-pairs shortest paths in TEDI, while, in m-hop, we only compute shortest paths from one vertex to all other vertices. For all the datasets, our index size is smaller than that of TEDI. The difference between the two algorithms is small, because data stored for the root node are the dominating factor of the index size when d is relatively small. Therefore, we do not report the index size for the following experiments.

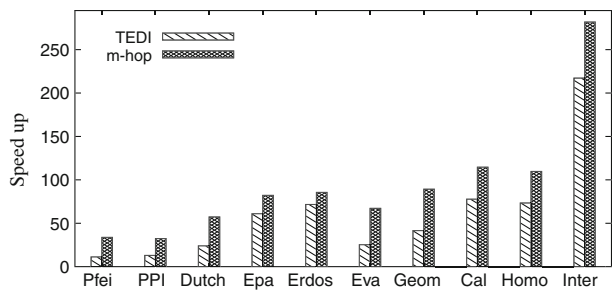

Fig. 9 Speedup of query time on small real datasets

8.3 Small synthetic datasets

We generate a set of small synthetic graphs according to the BA model [2] the same as that in Sect. 8.1. Here, we generate 10 graphs, where the number of vertices range from 1,000 to 10,000 by the step of 1,000, and denote the graph as xk if it contains $x \cdot 1{,}000$ vertices.

The query time of BFS, TEDI, and m-hop on these small synthetic graphs are shown in Table 8. Similar to the real graphs, the query time of BFS increases dramatically when the graph size increases. Actually, the query time of BFS on synthetic graphs is almost as the same as that on real graphs of the same size, as shown in Tables 6 and 8. The query time of m-hop increases very slowly as compared to BFS. For example, when the graph size increases from 1,000 to 10,000, the query time of BFS increases by a factor of 13, while the query time of m-hop increases 3 times. The speedups of m-hop and TEDI with respect to BFS are shown in Fig. 10. Consistently, the speedup of m-hop is 2 times larger than that of TEDI. For both algorithms, the speedups increase when the graph size increases, as explained for the real datasets.

The label construction time of TEDI and m-hop on these synthetic graphs are shown in Table 9. For both m-hop and TEDI, the label construction time increases when graph size increases. When graph size increases, it takes more time to build tree decomposition. Also the time to compute shortest paths increases when graph size increases, as shown by the query time of BFS in Table 8. For all the synthetic graphs, the label construction time of m-hop is only about 70–80 % of that of TEDI.

Table 8 Query time for small synthetic datasets

Graph	n	d	BFS (ms)	TEDI	m-hop
1k	1,000	3	0.0196	0.0016	0.0007
2k	2,000	5	0.0354	0.0018	0.0008
3k	3,000	6	0.0519	0.0021	0.0009
4k	4,000	7	0.0688	0.0023	0.0010
5k	5,000	8	0.0880	0.0027	0.0012
6k	6,000	9	0.1081	0.0034	0.0015
7k	7,000	9	0.1275	0.0034	0.0016
8k	8,000	9	0.1639	0.0028	0.0015
9k	9,000	9	0.2087	0.0031	0.0017
10k	10,000	9	0.2426	0.0033	0.0018

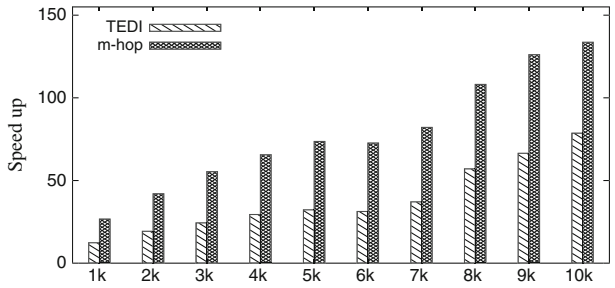

Fig. 10 Speedup of query time on small synthetic datasets

Table 9 Label construction time for small synthetic datasets

Graph	m-hop	TEDI	Graph	m-hop	TEDI
1k	7.7	9.2	6k	119	168
2k	21	26	7k	163	228
3k	39	49	8k	252	302
4k	63	78	9k	302	391
5k	92	116	10k	394	506

8.4 Large datasets

We test the performance of m-hop and TEDI under a given budget of the label size for large datasets. Specifically, given a budget B, we generate a tree decomposition, where the resulting label size is no larger than B, while the size of the root node is as large as possible (refer to the discussions on reducing tree height in Sect. 6).

Real datasets For the real datasets, we test m-hop and TEDI over two large road networks, California Road Network [17] and North America Road Networks, and a computer science bibliography graph, $DBLP$[2]. The number of vertices contained in these two road networks are 21, 048 and 175, 813, respectively. After processing as [29], the $DBLP$ dataset consists of 581K vertices. The average degree and the maximum/median distance of these three graphs are shown in Table 10.

Table 10 Information of large datasets

Graph	n	Average degree	Maximum distance	Median distance
California	21k	2.06	721	256
North America	175k	2.04	4,657	946
DBLP	581k	2.45	35	15

Table 11 Query and construction time for California Road Network

Budget size	Query time (ms)			Const time (ms)	
	BFS	TEDI	m-hop	TEDI	m-hop
10M	0.7035	0.0175	0.0027	5,781	5,133
40M	0.7035	0.0133	0.0021	11,979	11,655
70M	0.7035	0.0111	0.0017	17,052	16,518
100M	0.7035	0.0091	0.0015	21,021	20,519

The query time of m-hop and TEDI on the California road network are shown in Table 11, where the budget B of label size varies from 10M to 100M by a step of 30M. The query time of BFS is 0.7035 ms, which does not change for different budget sizes, because BFS directly works on the original graph. When the budget size B increases, the query time for both m-hop and TEDI decreases. This is because that the root node of the tree decomposition contains more vertices when budget B increases, therefore, the treewidth tw and height h decrease. The speedups of m-hop and TEDI with respect to BFS are shown in Fig. 11. For both m-hop and TEDI, the speedups increase when the budget size B increases. This is based on the fact that the query time of BFS remains unchanged, while the query time of m-hop and TEDI decreases, when B increases. The speedup of m-hop is 5–6 times larger than that of TEDI. The label construction time of m-hop and TEDI on the California road network is shown in Table 11. For a given budget size, m-hop takes

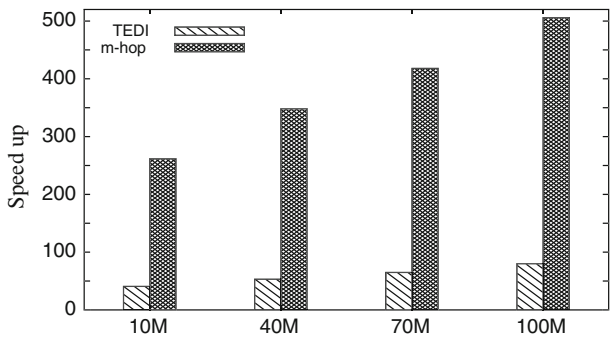

Fig. 11 Speedup of query time on California Road Network (varying B)

[2] http://www4.wiwiss.fu-berlin.de/bizer/d2rq/benchmarks.

Table 12 Query time for California Road Network with different query distance ($B = 40$M)

Query distance	BFS	TEDI	m-hop
[1,10]	0.0414	0.0033	0.0015
[11,50]	0.0564	0.0083	0.0023
[51,150]	0.1936	0.0121	0.0022
[151,800]	0.9902	0.0129	0.0020

less time to construct labels than that of TEDI because TEDI needs to compute more pairs of shortest paths and m-hop answers queries faster than TEDI.

Table 12 shows the query time of BFS, TEDI, and m-hop for California Road Network with different query distance settings, that is, we divide the queries into four ranges based on the distance of the query vertex pairs. As expected, the query time of BFS and TEDI increases when the distance of query vertex pairs becomes larger. The query time of m-hop does not change much, since the number of hops involved is not much related to the distance of a vertex pair.

The query time and label construction time of m-hop and TEDI on the North America road network are shown in Table 13, where the budget size ranges from 10M to 100M. The query time of BFS remains 26 ms, while the query time of m-hop and TEDI decreases, when budget size B increases. The construction time of m-hop increases when B increases, because the number of shortest path pairs computed in the construction of m-hop is linearly proportion to the label size. The construction time of TEDI first decreases then increases when B increases. This is because, when B is small, the treewidth tw is very large so it needs to compute a lot of shortest paths; when B becomes larger, the treewidth tw becomes almost stable while the size of root node becomes larger. The speedups of m-hop and TEDI with respect to BFS are shown in Fig. 12. When the budget size B increases from 10M to 20M, the speedup of TEDI increases from 62 to 189, while the speedup of m-hop increases from 203 to 626. The speedup of m-hop and TEDI becomes stable when the budget size increases from 20M. This is because that, although the tree decomposition becomes smaller, when B increases, the treewidth tw and height h do not change much when B is larger than 20M.

Table 13 Query and construction time for North America Road Network

Budget size	Query time (ms)			Const time (s)	
	BFS	TEDI	m-hop	TEDI	m-hop
10M	25.842	0.418	0.127	958.9	516.9
12M	25.842	0.376	0.098	979.9	543.7
14M	25.842	0.313	0.058	913.2	555.1
16M	25.842	0.190	0.050	856.5	562.1
18M	25.842	0.148	0.042	830.0	570.2
20M	25.842	0.138	0.042	853.4	584.1
30M	25.842	0.135	0.041	886.1	624.6
40M	25.842	0.134	0.043	911.4	648.2
70M	25.842	0.133	0.042	981.4	730.3
100M	25.842	0.132	0.042	1,049	789.5

For the $DBLP$ dataset, with a budget size B of 200M, our m-hop constructs labels in 7,482 s, while TEDI constructs index in 17,997 s. In query processing, the average processing time of m-hop is 0.393 ms, and the average processing time of TEDI is 1.008 ms, while it takes 96.6 ms for a BFS approach. The speedups of m-hop and TEDI overBFS are 245.2 and 95.7, respectively.

Synthetic datasets In the previous sets of experiments, we have tested our algorithm and TEDI on small synthetic graphs, that is, the graph sizes are less than 10,000. We test the algorithms on large synthetic graphs in the following. Specifically, we test the algorithms on graphs, whose sizes range from 20,000 to 100,000 by a step of 20,000, and denote the graph as xk if the size is $x \cdot 1{,}000$. The budget B of label size varies from 10M to 100M by a step of 30M.

We test the algorithms m-hop and TEDI when varying budge size B, while the graph size is chosen as 60k. The query time and construction time is shown in Table 14. When budget size B increases, the query time of m-hop and TEDI decreases, while the construction time of m-hop and TEDI increases. The speedups of m-hop and TEDI are shown

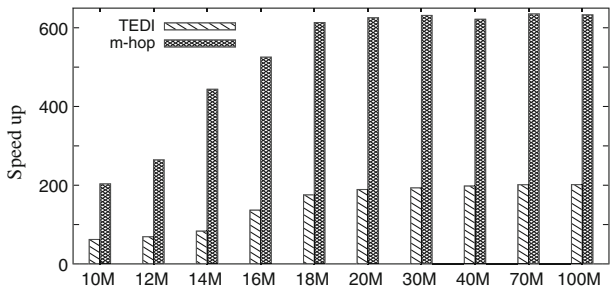

Fig. 12 Speedup of query time on North America Road Network (vary B)

Table 14 Query and construction time for synthetic graph ($n = 60{,}000$)

Budget size	Query time (ms)			Const time (s)	
	BFS	TEDI	m-hop	TEDI	m-hop
10M	6.2244	0.0040	0.0028	62	60
40M	6.2244	0.0028	0.0022	119	116
70M	6.2244	0.0025	0.0020	158	156
100M	6.2244	0.0023	0.0019	190	189

in Fig. 13, which increases when budget size B increases. The speedups of m-hop over TEDI are not as much as the previous testings, because the treewidth tw and height h are both very small for these synthetic graphs when the budget B is relative large.

Here, given a budget B of label size, we test the performance of m-hop and TEDI on graphs of different sizes. The query time and construction time of m-hop and TEDI are shown in Table 15, where B is equal to 40M. As expected, with the same label size, when graph size increases, the query time and construction time of all three algorithms increase. As shown in Fig. 14, the speedups of both m-hop and TEDI increase when the graph size increases. This is because that, although the query time of all three algorithms increases when the graph size increases, the query time of BFS increases much faster than that of m-hop and TEDI.

8.5 Density testing

We test the performance of m-hop and TEDI by varying the average degree of graphs. The synthetic graphs are generated with $n = 40{,}000$, and the average degrees ($\bar{d}$) vary from 2.2 to 4, denote the four graphs as $\bar{d} = 2.2$, $\bar{d} = 2.8$, $\bar{d} = 3.4$, and $\bar{d} = 4$. In all these testings, the budgets B for m-hop and TEDI are set as 70M.

Table 16 shows the query time and index construction time of the two algorithms on the four graphs. In general,

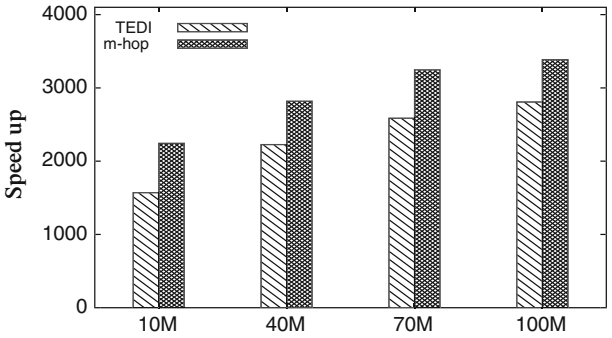

Fig. 13 Speedup of query time on large synthetic graph (varying B, $n = 60{,}000$)

Table 15 Query and construction time for synthetic graph ($B = 40$M)

Graph size	Query time (ms)			Const time (s)	
	BFS	TEDI	m-hop	TEDI	m-hop
20k	0.9867	0.0012	0.0010	18	19
40k	3.6326	0.0022	0.0018	63	62
60k	6.2244	0.0028	0.0022	119	116
80k	10.4778	0.0033	0.0026	209	203
100k	14.0068	0.0038	0.0029	294	279

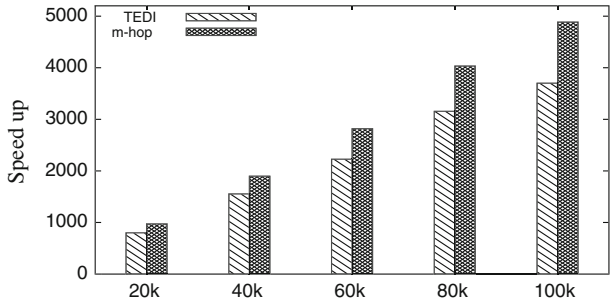

Fig. 14 Speedup of query time on large synthetic graph (varying n, $B = 40$M)

Table 16 Query and construction time for density testing ($n = 40{,}000$, $B = 70$ M)

Average degree	Query time (ms)			Const time (s)	
	BFS	TEDI	m-hop	TEDI	m-hop
2.2	3.6693	0.0019	0.0015	84	86
2.8	3.6156	0.0025	0.0021	92	94
3.4	3.5594	0.0041	0.0032	107	102
4.0	3.2254	0.0094	0.0069	143	104

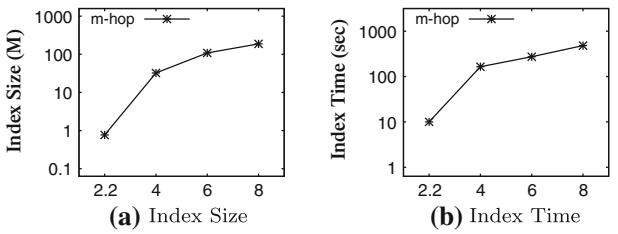

Fig. 15 Index size and time for different densities

when the average degree increases, the query time of BFS decreases while the query time of m-hop and TEDI increases. The BFS time decreases because the distances between query pairs become smaller, therefore BFS can find the shortest path with few hops of exploration. The query time and index construction time of m-hop and TEDI increase because the graph becomes larger. Consistently, our m-hop algorithm can answer shortest distance queries much faster than TEDI.

Indexing time and size Here, we test the growing trends of index size and index time of m-hop, when density increases. The synthetic graphs are generated with $n = 40{,}000$, and the average degrees ($\bar{d}$) vary from 2.2 to 8, namely, 2.2, 4, 6, and 8, as indicated in Fig. 15. In this testing, we do not specify a budget of index size, and the tree decomposition is built such that the index size is minimal for our algorithm. The index size and index time for different densities are shown in Fig. 15a, b, respectively. From Fig. 15, we can see that both the index size and index time increase very fast when the graph becomes denser. Our current implementation will run out of memory when building tree decomposition for dense

Table 17 Query and construction time for weighted California Road Network

Budget size	Query time (ms)			Const time (ms)	
	Dijkstra	TEDI	m-hop	TEDI	m-hop
10M	1.9830	0.0163	0.0026	11,956	11,371
40M	1.9830	0.0131	0.0021	25,072	24,650
70M	1.9830	0.0105	0.0017	35,605	34,345
100M	1.9830	0.0089	0.0014	43,365	43,166

graphs, we are planning to work on dealing with dense graphs as our future work.

8.6 Weighted graph

Here, we test the performance of index and query algorithms of our approach and TEDI. The dataset is California Road Network, and edge weights are obtained from the original dataset.[3]

The query time and index construction time of m-hop and TEDI on this weighted dataset are shown in Table 17. Comparing the approaches on weighted and unweighted graphs, the only difference lies in shortest path computation when computing distance labels. Therefore, it will need more time to compute distance labels on weighted graph, and the query time remains almost the same. This is confirmed by comparing Table 17 with Table 11.

9 Conclusion

In this paper, we studied a small distance labeling scheme to fast query shortest distances. In our multi-hop distance labeling, instead of directly generating 2-hop distance labels as index, we generate a small set of query-specific 2-hop distance labels on-line efficiently based on our stored multi-hop distance labels. The multi-hop distance labels stored in our approach is only a subset of that generated by a 2-hop distance labeling, so it is small in size. Furthermore, our multi-hop distance labels generating algorithm avoids pre-computing all-pairs shortest paths. We give efficient algorithms to generate the query-specific 2-hop distance labels based on our stored multi-hop distance labels. We conducted extensive performance studies to compare our approaches with the up-to-date existing approaches, using a large number of small/large real/synthetic graphs, and confirmed the efficiency of our approach. Although we confirmed the efficiency of our approach on large sparse graphs, one limitation of our current version of multi-hop distance labeling is that it may run out of memory when building tree decomposition for dense graphs. We are planning to work on dealing with dense graphs in a memory constrained environment as our future work.

[3] http://www.cs.utah.edu/~lifeifei/SpatialDataset.htm.

References

1. Arnborg, S., Corneil, D.G., Proskurowski, A.: Complexity of finding embeddings in a k-tree. SIAM J. Algebraic Discret Methods **8**(2), 277–284 (1987)
2. Barabasi, A.-L., Albert, R.: Emergence of scaling in random networks. Science. 286, 509–512 (1999)
3. Baswana, S., Sen, S.: Approximate distance oracles for unweighted graphs in expected o(n2) time. ACM Trans. Algorithms **2**(4), 557–577 (2006)
4. Bodlaender, H.L., Koster, A.M.C.A: Treewidth computations I. Upper bounds. Inf. Comput. **208**(3), 259–275 (2010)
5. Chaudhuri, S., Zaroliagis, C.D.: Shortest paths in digraphs of small treewidth. Part I: Sequential algorithms. Algorithmica. **27**(3), 212–226 (2000)
6. Cheng, J., Yu, J.X.: On-line exact shortest distance query processing. In: Proceedings of EDBT'09, pp. 481–492 (2009)
7. Cohen, E., Halperin, E., Kaplan, H., Zwick, U.: Reachability and distance queries via 2-hop labels. In: Proceedings of SODA'02, pp. 937–946 (2002)
8. Cormen, T.H., Stein, C., Rivest, R.L., Leiserson, C.E.: Introduction to Algorithms. McGraw-Hill Higher Education, USA (2001)
9. Das Sarma, A., Gollapudi, S., Najork, M., Panigrahy, R.: A sketch-based distance oracle for web-scale graphs. In: Proceedings of WSDM'10, pp. 401–410 (2010)
10. Fan, W., Ma, S., Tang, N., Wu, Y., Wu, Y.: Graph pattern matching: from intractable to polynomial time. PVLDB **3**(1), 264–275 (2010)
11. Gavoille, C., Peleg, D., Perennes, S., Raz, R.: Distance labeling in graphs. In: Proceedings of SODA'01, pp. 210–219 (2001)
12. Geisberger, R., Sanders, P., Schultes, D., Delling, D.: Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In: WEA, pp. 319–333. (2008)
13. Gou, G., Chirkova, R.: Efficient algorithms for exact ranked twig-pattern matching over graphs. In: Proceedings of SIGMOD'08, pp. 581–594 (2008)
14. Jin, R., Xiang, Y., Ruan, N., Fuhry, D.: 3-hop: a high-compression indexing scheme for reachability query. In: Proceedings of SIGMOD'09, pp. 813–826 (2009)
15. Jin, R., Xiang, Y., Ruan, N., Wang, H.: Efficiently answering reachability queries on very large directed graphs. In: Proceedings of SIGMOD'08, pp. 595–608 (2008)
16. Katz, M., Katz, N.A., Peleg, D.: Distance labeling schemes for well-separated graph classes. In: Proceedings of STACS'00, pp. 516–528 (2000)
17. Li, F., Cheng, D., Hadjieleftheriou, M., Kollios, G., Teng, S.-H.: On trip planning queries in spatial databases. In: Proceedings of SSTD'05, pp. 273–290 (2005)
18. Liu, L., Wong, R.C.-W.: Finding shortest path on land surface. In: Proceedings of SIGMOD'11, pp. 433–444 (2011)
19. Peleg, D.: Proximity-preserving labeling schemes. J. Graph Theory **33**(3), 167–176 (2000)
20. Rahman, S.A., Advani, P., Schunk, R., Schrader, R., Schomburg, D.: Metabolic pathway analysis web service (pathway hunter tool at cubic). Bioinformatics **21**(7), 1189–1193 (2005)
21. Rice, M., Tsotras, V.J.: Graph indexing of road networks for shortest path queries with label restrictions. PVLDB **4**(2), 69–80 (2010)
22. Robertson, N., Seymour, P.D.: Graph minors. iii. Planar tree-width. J. Comb. Theory Ser. B **36**(1), 49–64 (1984)
23. Sankaranarayanan, J., Samet, H.: Roads belong in databases. IEEE Data Eng. Bull. **33**(2), 4–11 (2010)

24. Schenkel, R., Theobald, A., Weikum, G.: Hopi: an efficient connection index for complex xml document collections. In: Proceedings of EDBT'04, pp. 237–255 (2004)
25. Schenkel, R., Theobald, A., Weikum, G.: Efficient creation and incremental maintenance of the hopi index for complex xml document collections. In: Proceedings of ICDE'05, pp. 360–371 (2005)
26. Sommer, C., Verbin, E., Yu, W.: Distance oracles for sparse graphs. In: Proceedings of FOCS'09, pp. 703–712 (2009)
27. Thorup, M., Zwick, U.: Approximate distance oracles. In: Proceedings of STOC'01, pp. 183–192 (2001)
28. van Schaik, S.J., de Moor, O.: A memory efficient reachability data structure through bit vector compression. In: Proceedings of SIGMOD'11, pp. 913–924 (2011)
29. Wei, F.: TEDI: efficient shortest path query answering on graphs. In: Proceedings of SIGMOD'10, pp. 99–110 (2010)
30. Xiao, Y., Wu, W., Pei, J., Wang, W., He, Z.: Efficiently indexing shortest paths by exploiting symmetry in graphs. In: Proceedings of EDBT'09, pp. 493–504 (2009)
31. Yildirim, H., Chaoji, V., Zaki, M.J.: Grail: scalable reachability index for large graphs. PVLDB **3**(1), 276–284 (2010)
32. Yu, J.X., Qin, L., Chang, L.: Keyword Search in Databases. Morgan & Claypool Publishers, San Francisco (2010)