The VLDB Journal (2014) 23:677–695
DOI 10.1007/s00778-013-0338-6

REGULAR PAPER

Answering exact distance queries on real-world graphs with bounded performance guarantees

Yang Xiang

Received: 18 February 2013 / Revised: 30 August 2013 / Accepted: 7 September 2013 / Published online: 29 September 2013

Abstract The ability to efficiently obtain exact distance information from both directed and undirected graphs is desired by many real-world applications. In this work, we unified the query indexing efforts on directed and undirected graphs into one by proposing the TreeMap approach. Our approach has very tight bounds on query time, index size, and construction time for answering queries on both directed and undirected graphs. The query time complexity is close to constant for graphs with a small width of tree decomposition, and the index construction can be completed without materializing the distance matrix or other high-cost operations. In the empirical study, we demonstrated that the TreeMap approach in general performs much better than competitive methods in indexing real graphs for answering exact distance queries.

Keywords Graph databases · Tree decomposition · Exact distance queries

1 Introduction

One of the most important information processing tasks of this decade is to efficiently handle the increasingly large graph data emerging from various domains. The distance or shortest path information is highly desirable for understanding and managing these graph databases. However, many large networks, emerging from domains such as social science and biomedical science, are becoming the bottlenecks for information retrieval and knowledge discovery applications.

Current approaches to the challenge of answering distance queries on large graph data can be classified into two categories. The first includes work focused on efficiently answering exact distance queries, such as [10,11,13,38]. Although these approaches have demonstrated their powers in handling large real graphs, they suffer from unpredictable index construction cost. The second category of approaches have addressed these problems in their attempt to provide estimated results to distance queries on real graphs, e.g., [20,30,34]. These works relax the exact distance requirement and have the capacity to tolerate some errors. Their goal is to handle massive graphs by this relaxation. Empirical studies have demonstrated their effectiveness on distance estimations on large real graphs with low average error rates. However, to our knowledge, none of these approaches have been able to guarantee a tight error bound on a single distance query. This error bound guarantee is often desired by many applications because occasional large errors could lead to unexpected outcomes, especially when users are "convinced" that the system they use has a low error rate. On the other hand, many real-world graphs have similar properties, or more famously known as the theory "six degrees of separation." This makes them sensitive to even tiny errors. For example, the number of persons with Erdos number 2 is significantly smaller than with Erdos number 3, though the difference is only 1 hop. Given these facts, we can understand why exact distance queries are still highly desirable in today's graph database applications and why this work focuses on exact distance queries.

Work related to answering exact distance queries can also be further divided into two subgroups. The first subgroup focuses on answering distance queries on undirected graphs

Y. Xiang (✉)
Department of Biomedical Informatics, The Ohio State University, Columbus, OH 43210, USA
e-mail: yxiang@bmi.osu.edu

such as [10,38], while the other subgroup focuses on directed graphs such as [11,23]. Both types of subgroups frequently emerge from real-world applications. In some circumstances, although a graph is directed, people are interested in finding out both an undirected distance and a directed distance at the same time. For example, in the ontology networks, a directed distance often tells how specific one concept is with respect to a general concept, while an undirected distance provides a general idea on the closeness of two concepts. Motivated by real-world requirements, this work unifies efficiently answering both directed and undirected distances on graphs.

By reviewing the past work on exact distance queries, we have found that many of them were developed by following and extending the approaches for answering reachability queries. For example, [11] uses the general strategy of 2-hop [13], and [23] adopts the strategy of tree cover, which has been used for a long time in designing reachability index schemes [2,24,37]. Different from these methods, Wei [38] recently handled the exact distance queries on undirected graphs via tree decompositions. However, the tree decomposition is not a new concept. It has been introduced in graph theory decades ago [31] to process a graph by mapping its vertices onto a tree. This effort is very successful for many special graph families with bounded treewidth [6,7,9,21]. However, its potential in handling real graphs was largely ignored until [38]. In [38], Wei demonstrated that TEDI (short for TreE-Decomposition-based Indexing) is very effective for handling large real graphs. The widths of tree decomposition on various real graphs, reported in [38], range from below 100, to around 1,000. Coincidentally, in a recent work [14], Montgolfier et al. studied the width of tree decomposition on the Internet and found that the treewidths of several Internet graphs are no more than 300. These results suggest that although real graphs do not have a very small treewidth as special graph families do, their treewidths are often much smaller in comparison with their sizes.

Motivated by the above observations, in this work, we propose a novel indexing scheme that efficiently answers exact distance queries on both directed and undirected graphs. Our index scheme not only maps graph vertices to several trees, but also builds up the label of a vertex by segments that map to tree nodes. Thus we name our approach "TREEMAP." The advantage of our approach is that it can very efficiently locate the necessary information for answering queries. In fact, TREEMAP has tightly bounded query and construction time complexities, plus an exact number bound on the index size. An overview of the theoretical bounds of TREEMAP's and other competitive methods are given in Table 1. The empirical study on large real datasets demonstrates that the TREEMAP approach significantly outperforms competitive methods on the time efficiency of query processing and index construction.

1.1 Related works

The following provides an overview of relevant work related to the TREEMAP approach and background information pertaining to its relevance for our work.

Reachability queries on directed graphs Efficiently answering reachability queries on directed graphs has been studied for a long time in the graph database community [24,25,35, 36,40]. It is clear that answering reachability queries is a special task of answering distance queries. For directed graphs, answering distance queries also answers the corresponding reachability queries. On the other hand, the reachability information can speed up the distance queries on these graphs. If we know a vertex u cannot reach another vertex v, we can save the unnecessary distance query from u to v. In fact, our TREEMAP approach fully utilizes this technique to speed up the distance queries on directed graphs, and its empirical performance is extremely good.

Exact distance queries on directed graphs The exact distance queries on directed graphs provide much detailed information between graph vertices than reachability queries do. The 2-hop approach [13], proposed to answer reachability queries, can also be applied to answer exact distance queries. However, several works have pointed out that 2-hop suffers from scalability issues [23,25] and thus can hardly be used in handling real large graphs. [11,32] extend the 2-hop strategy to answer distance queries on directed graphs by reducing the construction time. [1] advances the 2-hop strategy by hierarchical hub labeling where the label of each vertex consists of the distances to hubs. However, these works still did not produce a low complexity bound on the construction time, even though they dropped the approximation bound on label size which was guaranteed in the original 2-hop work. Recently Jin et al. [23] proposed a highway-centric labeling approach for answering exact distance queries on directed graphs. Though [23] does not produce a tight bound on query time, index size, and construction time, its index size reduction by bipartite set cover approach was demonstrated to be close to optimal. Correspondingly, it has a good performance in empirical studies. Thus, we select the highway-centric labeling as a benchmark for our TREEMAP approach in the empirical study of this work.

Exact distance queries on undirected graphs Undirected graphs are ubiquitous. Many applications are also interested in finding undirected distances on graphs. For these applications, TEDI [38] proposed by Wei is one of the most effective methods in handling exact distance queries on undirected graphs. As discussed above, TEDI is applicable to many real graphs because they have very limited treewidths in comparison with their sizes. TEDI provides a user controllable parameter k. When $k = 0$ or closer, TEDI has done none or very little processing over the tree decomposition, and the indexing scheme degrades to a naive distance indexing scheme

Table 1 Comparisons between TreeMap and other competitive methods on three standard measurements

	Query time	Index size	Construction time
Distance matrix	$O(1)$	n^2	$O(n^2 + n * m)$
BFS	$O(n + m)$	None	None
2-hop [13][a]	$\tilde{O}(m^{1/2})$	$\tilde{O}(nm^{1/2})$	$O(n^3\lvert TC\rvert)$ or $O(n^5)$
Online exact [11]	Similar to 2-hop		
TEDI [38][b]	$O(h * k^2)$	$\le \sum_{X_i \in \mathcal{X}} *\lvert X_i\rvert^2$	$O(n^2 + n * m)$
TEDI($k = 0$) [38]	$O(1)$	$\le n^2$	$O(n^2 + n * m)$
TEDI($k = tw$) [38]	$O(h * tw^2)$	$\le n * (tw + 1)^2$	$O(n^2 + n * m)$
m-hop [10]	$O(h * tw)$	$O(\sum_{X_i \in \mathcal{X}} \lvert X_i\rvert)$	$O(n^2 + n * m)$
HighwayCentric [23][c]	$O(n)$	$O(n^2)$	$\approx O(n^3)$
TreeMap (undirected)	$O(tw)$	$\le n * (tw + 1) * (\log_2 n + 1) + 5n + 1$	$O(tw^2 * n * \log^2 n + tw * m * \log n)$
TreeMap (directed)	$O(tw)$	$\le n * (tw + 1) * (4 \log_2 n + 1) + 5n + 1$	$O(tw^2 * n * \log^2 n + tw * m * \log n)$

[a] The index size is a conjecture
[b] TEDI considers the time complexity of BFS to be $O(n)$. Although this is true for many real graphs whose number of edges are a constant times over the number of vertices, it is more accurate in our opinion to describe the BFS complexity by $O(n + m)$
[c] The authors observed the label size for each vertex is generally close to constant in empirical studies. However, no analytical result was provided to substitute the naive worst case complexity analysis on the time and index size as provided in this table

which materialize the distance matrix. It is clearly not acceptable to large graph data due to its n^2 index size. On the other hand, when we maximize k(k will never exceed tw), i.e., maximize the processing on the tree decomposition, then we fully receive the benefit from the tree decomposition and the index size upper bound is reduced to $n * (tw + 1)^2$, an acceptable level for large graph data. However, in this case, the query time increases to $O(n * tw^2)$. Recently, Chang et al. [10] proposed m-hop by improving TEDI's query time complexity from $O(h * tw^2)$ to $O(h * tw)$. As demonstrated in the empirical study of [10], m-hop has similar or slightly improved construction time and index size in comparison with TEDI. However, the query speed improvement of m-hop over TEDI is quite limited. In our empirical study, we still use the publicly available source code TEDI as a benchmark for our TreeMap approach. We demonstrate in our empirical study that our TreeMap approach has much clearer advantages over TEDI than m-hop does, especially on the large real graphs.

Distance estimation Since the current exact distance query methods have limited ability in handling large real graphs, the distance estimation approaches [20,30,34] were proposed. However, and as we mentioned above, a major limitation is that none of these approaches can provide a tight error bound on one single query. Although many of them demonstrated an average low error rate, it is still unacceptable for some applications to see an occasional large error. If exact distance query techniques can be advanced to handle sufficiently large graphs, we expect it will replace the distance estimation approaches. Our effort in this work focuses on advancing the exact distance query techniques on both directed and undirected graphs. Though it performs very well in our empirical study, it is clearly limited by the width of a tree decomposition and thus cannot effectively handle graphs with an extremely large treewidth. However, our approach could be applied to distance estimation if by allowing errors we can significantly simplify the tree decomposition, and this would be a very interesting topic in the future.

1.2 Main contributions

As listed in Table 1, TreeMap has very tight bounds on query time, index size, and construction time for answering exact distance queries on both directed and undirected graphs. In comparison with the other competitive methods, its significance is almost self-evident. Below we summarize the main contributions of this work.

1. As a strong advantage over available methods, TreeMap has very tight bounds on all the three standard measurements for graph indexing schemes. In particular, the complexity of TreeMap's construction time is much lower than the competitive methods. The index construction can be done without materializing the distance matrix or other high-cost operations.
2. The index size upper bound analysis of TreeMap is provided without the big-O notation in which large constants may hide. This is particularly helpful for budgeting the indexing cost on actual implementations.
3. TreeMap can be applied to both undirected graphs and directed graphs with almost the same bounds and can be extended to support weighted graphs and support answering distance queries in distributed environment.

4. We provide two implementations with different focuses, TREEMAP- F for fast query and TREEMAP-C for compact label size.
5. The empirical study demonstrates that the overall performance of TREEMAP is much better than competitive methods.

In addition, we provide a running example throughout the paper to facilitate the understanding of our approach. The labels for the running example are exactly generated by our TREEMAP program.

2 Preliminaries

Our work is focused on answering exact distance queries in both directed and undirected graphs. There are, however, significant overlaps in treating these two types of graphs using the tree mapping strategy. To simplify our discussions, we assume graphs are undirected from this section to Sect. 6, and we extend our discussions to directed graphs in Sect. 7. We empirically study the performance of the proposed methods in both directed and undirected graphs in Sect. 8.

2.1 Tree decomposition basics

The *distance* between two vertices $u, v \in V$, denoted as $d_G(u, v)$, is the length (i.e., the number of hops) of a shortest path connecting u and v in G. Following the frequently used notations in literature, we let $|V| = n$ and $|E| = m$.

Given a graph $G = (V, E)$, a tree decomposition [15] is a pair $(\mathcal{X}, \mathcal{T})$, where $\mathcal{X}$ is a set of V's subsets, and $\mathcal{T}$ is a tree with node set $\mathcal{X}$. $(\mathcal{X}, \mathcal{T})$ must satisfy three properties:

1. $\bigcup_{X_i \in \mathcal{X}} X_i = V$;
2. for any $(u, v) \in E$, there exists a $X_i \in \mathcal{X}$ such that $u \in X_i$ and $v \in X_i$;
3. for any $v \in V$, the node set $\{X_i | v \in X_i, X_i \in \mathcal{X}\}$ induces a connected subtree on $\mathcal{T}$.

A tree decomposition on a graph maps the graph's vertices to tree nodes. A tree node is also called a tree bag because it often contains multiple vertices from the graph. The *height* of a tree is the length of a longest path starting (or ending) at the tree root. A *centroid* node of a tree is a node whose removal results in a number of subtrees, none of which contains more than one half of nodes of the original tree. As discussed in [17], a tree has either one or two centroid nodes.

The width of a tree decomposition $(\mathcal{X}, \mathcal{T})$ is $\max_{X_i \in \mathcal{X}} (|X_i| - 1)$, i.e., the size of the largest tree bag minus 1. In graph theory, the treewidth of a graph is defined as the minimum width of all possible tree decompositions for that graph. Since

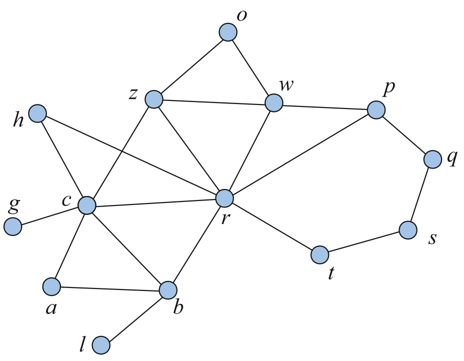

Fig. 1 Graph G, a running example

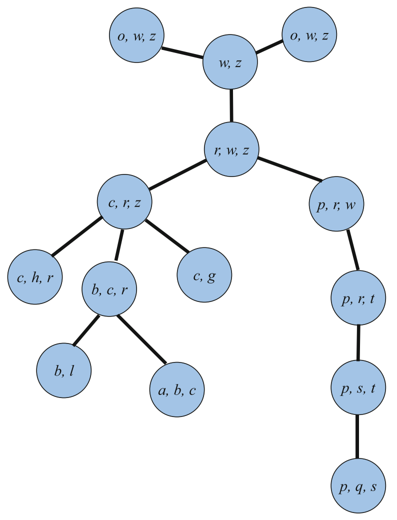

Fig. 2 Tree decomposition $(\mathcal{X}, \mathcal{T})$ for G

determining the treewidth of a graph is well known as NP-hard [3], the concept of treewidth is mainly of theoretical interest. In this paper, we will focus on the width of a tree decomposition instead of the treewidth of a graph, and we use tw to denote the width of a tree decomposition. Quite a few methods are available in literature [8,10,38] for generating a tree decomposition which satisfies $O(|\mathcal{X}|) = O(n)$, i.e., the number of tree node is proportional to the number of vertices in the original graph.

To facilitate the understanding of our method, we use a simple running example, as shown in Fig. 1 throughout the following discussions.

A graph may have many different tree decompositions with the same width tw. In Fig. 2, a tree decomposition $(\mathcal{X}, \mathcal{T})$ of $tw = 2$ for the example graph G is given, with $|\mathcal{X}| = |V|$. To be consistent, this decomposition is generated by our program which is used for the empirical study in Sect. 8. Our program implements the tree decomposition by peeling off lowest degree vertices first. The general strategy has been described in [38]. However, [38] has an option to build an incomplete tree decomposition with a large root by stopping the peel-off procedure at some point, while our program does not utilize this option and always processes the peel-off procedure completely. As a result, our program will generate a tree decomposition with $|\mathcal{X}| = |V|$. Note that on graphs consisting of disconnected subgraphs, such as isolated vertices, the resulting tree decomposition is actually a forest which still satisfies the three properties of the tree decomposition. Since our method relies on the three properties of the tree decomposition and disregards whether a decomposition results in a tree or a forest, it handles forests without a problem. In the following sections, we will present additional running results related to the given example of Fig. 1, and all of them are generated by our program.

2.2 The power of tree bag separators

Many successful graph algorithms, including graph labeling for distance queries (e.g., [11,33]), use the graph separation strategy. A graph separation removes a subset of vertices (and their associated edges) to break the graph into disconnected subgraphs and reapplies the separation in a divide and conquer style.

The tree decomposition of a graph, in fact, provides very valuable resource of graph separators, as stated by Lemma 1 in [10]. In fact, the proof of this lemma in [10] provides additional information which can be summarized as follows:

Lemma 1 [10] *Assume $\mathcal{T}_a$ with node set $\mathcal{X}_a$, and $\mathcal{T}_b$ with node set $\mathcal{X}_b$ are any two subtrees of $\mathcal{T}$ obtained by removing node X_c from $\mathcal{T}$. Assume G' be the graph obtained by removing all vertices in X_c and their associated edges from G. Then (1) $((\bigcup_{X_i \in \mathcal{X}_a} X_i) \backslash X_c) \cap ((\bigcup_{X_j \in \mathcal{X}_b} X_j) \backslash X_c) = \emptyset$ holds; and (2) there is no path connecting any two vertices u and v in G' if there are two tree nodes $X(u)$ and $X(v)$ separated as a result of removing X_c from $\mathcal{T}$, where $X(u) \in \mathcal{X}$, $X(u) \ni u$, $X(v) \in \mathcal{X}$, and $X(v) \ni v$.*

Lemma 1 tells us that each tree node is a candidate separator for the graph induced by vertices not included in that node. For any two vertices u and v that are separated by a vertex set S, their distance $d_G(u, v)$ can be obtained as:

$$d_G(u, v) = \min_{x \in S}(d_G(u, x) + d_G(x, v)).$$

This is quite obvious because any paths connecting u and v, including shortest paths must pass S. With Lemma 1, authors of [10] concluded that 2-hop distance labels for G can be generated using a tree decomposition of G, and developed m-hop method. However, we find that Lemma 1 leads to a more general and interesting distance labeling problem as described in the following.

Separator Assignment Problem: *Let $L(v)$ represent the label of vertex v. Given the separator candidates from tree decomposition $\mathcal{X} = \{X_1, X_2, \ldots, X_n\}$, what separators shall be assigned to a vertex v, such that its label size $L(v)$ (where $L(v) = \bigcup \mathcal{X}_v (\mathcal{X}_v \subseteq \mathcal{X})$) is minimized, and the distance between two vertices v and u can be answered by letting:*

$$d_G(u, v) = \min_{x \in L(v) \cap L(u)} d_G(v, x) + d_G(x, u)?$$

The property of tree centroid provides an answer to the above problem. Each tree has one or two centroid nodes, whose removal results in no subtree with size larger than half of the original tree. This technique has been successfully used in handling tree-based structures. For example, [29] used this approach to design a labeling scheme to answer exact tree distance, and [16] used this approach to design collective tree spanners for tree decompositions. Here we can follow the same strategy: Each time we select a centroid of the tree $\mathcal{T}$ and assign this node to all vertices contained by $\mathcal{T}$. Then, we remove this node from $\mathcal{T}$ and iteratively apply this approach to its subtrees. According to the property of centroid, it is easy to conclude that each vertex will be assigned at most $O(\log n)$ tree nodes (i.e., separator candidates), and correspondingly, its label size is $O(tw \log n)$. It is easy to see the time complexity of this assignment is $O(n \log n)$, which is also implied in [16]. In addition, it is quite interesting to observe that this label size is optimal with respect to the above Separator Assignment Problem, as stated in the following observation.

Observation 1 *The $O(tw \log n)$ label size complexity is optimal for the Separator Assignment Problem.*

The correctness of Observation 1 can be easily proved by combining the following two facts: A tree has a tree decomposition with $tw = 1$; the minimum tree labeling scheme for answering the exact distance by comparing the label of two tree nodes requires assigning at least a $O(\log n)$ size label to each vertex [18].

Given the above discussion, we can observe if the label of each vertex can be assigned the distances to the vertices in the $O(\log n)$ separators, we can answer the distance between any two vertices by comparing their labels. However, at this moment, it is not clear how the label assignment can be achieved efficiently. Thus, our first target is to design an efficient label assigning algorithm for the Separator Assignment Problem.

3 Efficient graph labeling via recursive separations

The recursive graph separation can be represented by a separator tree T, which is defined recursively by letting the root be the centroid of $\mathcal{T}$, and its children be the centroids of the subtrees resulting from removing the root from $\mathcal{T}$. The tree formed by the solid edges in Fig. 3 is an example T for $\mathcal{T}$ in Fig. 2. Our labeling algorithm will follow the layered

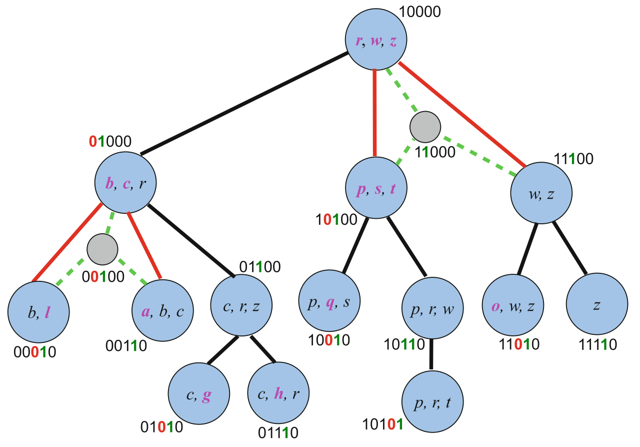

Fig. 3 (1) *Solid lines* make up T; (2) *black solid lines* and *green dashed lines* make up T′; (3) *pink vertices* in a node are vertices exclusively mapped to this node; (4) *gray* nodes are dummy nodes added in T′; (5) the binary number aside a node is the bit label assigned by TREELABEL Algorithm

Table 2 An example of L_G for each vertex in G after Algorithm 1 SEPARATORLABELING

Vertex	L_G
a	(a,0) (b,1) (c,1) (r,2) (w,3) (z,2)
b	(b,0) (r,1) **(w,3)** (z,2)
c	(b,1) (c,0) (r,1) (w,2) (z,1)
g	(b,2) (c,1) (g,0) (r,2) (w,3) (z,2)
h	(b,2) (c,1) (h,0) (r,1) **(w,3)** (z,2)
l	(b,1) (l,0) (r,2) **(w,4)** (z,3)
o	(o,0) (r,2) (w,1) (z,1)
p	(p,0) (r,1) (w,1)
q	(p,1) (q,0) (r,2) (s,1) (w,2)
r	(r,0)
s	(p,2) (r,2) (s,0) (w,3)
t	**(p,3)** (r,1) (s,1) (t,0) **(w,4)**
w	(r,1) (w,0)
z	(r,1) (w,1) (z,0)

The pairs in bold are not exact distances

approach (from root to leaves) to label the graph G. Algorithm 1 is its pseudocode.

Algorithm 1 SEPARATORLABELING($G = (V, E)$, T, $Label_G$)

```
1: enqueue $root$(T) into Queue Q;
2: while Q is not empty do
3:   $X = dequeue(Q)$;
4:   for all child $Y$ of $X$ on T do
5:     enqueue $Y$ onto Q;
6:   end for
7:   for all $v \in X$ do
8:     if $v$ has not been deleted then
9:       BFSLabeling($G$, $v$, $Label_G$);
10:      delete $v$ from $G$;
11:    end if
12:  end for
13: end while
```

The general workflow of Algorithm 1 is quite easy to understand as it follows the BFS order to traverse T. The key point in Algorithm 1 is the BFSLabeling procedure. It starts from a given vertex v and visits G in the BFS order. For each vertex u it visits, the BFSLabeling procedure assigns a distance label $(v, d_{L_G(v)}(x))$ to that vertex. Table 2 shows the vertex labels L_G after Algorithm SEPARATORLABELING finishes for our running example. Here and in the following discussions, we let $L_G(v)$ denote the label of v, and $d_{L_G(v)}(x)$ denote the distance paired with x in the label of v.

It is important to note that at this moment, $d_{L_G(v)}(x)$ may not be the exact distance between v and x in the original graph because some vertices may be deleted prior to the BFSLabeling starting on v, and the label of a vertex may not contain the distances to all vertices in the separators that are associated with it. However, we can use L_G to correctly compute the distance between any two vertices. In fact, the accuracy is not affected by the order of vertices being handled in the labeling process. In our recent work [39], we called the BFS-Labeing process "Decentralizing Labeling Scheme (DLS)" and we showed that by greedily decentralizing the highest degree vertices and limiting the BFSLabeling to 6 neighborhood, we will be able to handle the UMLS graph, a very large biomedical knowledge graph, in a very efficient way. In contrast to [39] which adopts a greedy approach, this work uses tree mapping information to guide the BFSLabeing process and results in very tight theoretical bounds. Thus, the tree mapping approach is compatible with the DLS approach. In Sect. 8.3, we will see that it is possible to combine the tree mapping approach with the DLS approach in indexing very large graphs for asking distance queries. The correctness is guaranteed by the fact that such a combined approach essentially determines a vertex decentralizing order (i.e., decentralizing vertices greedily first, and then decentralizing vertices according to the guidance provided by tree mapping).

Lemma 1 in [39] can be reiterated in the following to show the correctness of answering exact distance queries using L_G. For the completeness of our paper, we provide a proof sketch for the following lemma in the context of this work.

Lemma 2 [39] *Let L_G be the labels assigned by Algorithm* 1 SEPARATORLABELING. *The exact distance between any two vertices u and v can be answered by*

$$d_G(u, v) = \min_{x \in L_G(v) \cap L_G(u)} d_{L_G(v)}(x) + d_{L_G(v)}(x).$$

Proof Without loss of generality, let u and v be any two vertices in G, and let p be a shortest path connecting u and v. Again, without loss of generality, let x be the first vertex on p that BFSLabeling procedure starts with. Then we have $d_{L_G(u)}(x) = d_p(u, x)$ and $d_{L_G(v)}(x) = d_p(v, x)$. Thus, $d_{L_G(u)}(x) + d_{L_G(v)}(x) = d_p(u, x) + d_p(v, x) = d_G(u, v)$. □

Algorithm SEPARATORLABELING works efficiently because a vertex is deleted after BFSLabeling started on it, and after all vertices in a bag are deleted, the graph G is separated following Lemma 1. In the proof of Theorem 3, we show that the time complexity for SEPARATORLABELING is no more than $O(tw^2 * n * \log^2 n + tw * m * \log n)$. Since each vertex is assigned a label with at most $(tw + 1) * \log n$ vertex–distance pairs, the distance query by the linear comparison of two vertex labels has a $O(tw \log n)$ time complexity. However, at this point, the labeling and query scheme have not been sufficiently refined. In the following sections, we show that the query time complexity can be significantly improved without increasing the label size and label time complexities.

4 Pinpointing the separator of two vertices

Sections 2.2 and 3 proposed to use tree nodes to guide vertex labeling for answering exact distance queries. However, as described in Lemma 1, the separating power of the tree decomposition node tells us that for any two vertices, they only need to record the distances to vertices in one tree node that separate them, in order to answer exact distance queries. Thus, the query on two vertices would be answered very efficiently if we can quickly locate in their labels the part that is associated with one tree node separator. Thus, we have the following problem.

SEPARATOR PINPOINTING PROBLEM: *Given an assignment of separator candidates from tree decomposition* $\mathcal{X} = \{X_1, X_2, \ldots, X_n\}$ *to each vertex* (i.e., $L(v) = \bigcup \mathcal{X}_v$) *such that* $d_G(u, v) = \min_{x \in L(v) \cap L(u)} d_G(v, x) + d_G(x, u)$, *how to pinpoint the part of a tree node separator* $Y (Y \in \mathcal{X}_u \cap \mathcal{X}_v)$ *from their labels such that*

$d_G(u, v) = \min_{x \in Y} d_G(v, x) + d_G(x, u)$?

To better understand the Separator Pinpointing Problem, let us recall the SEPARATORLABELING algorithm. A vertex v will be discarded immediately after the BFSLabeling started from it. Thus, even if v's descendant tree bags contain it, it will not be processed by BFSLabeling again. In other words, v only receives distance pair information from vertices contained by tree nodes closer to the root than v. Thus, v is essentially "mapped" to the tree node that contains it and is closest to the root of T. It is interesting to observe such mapping is unique, because if it is mapped to more than one such tree nodes at the same depth to the root of T, these nodes containing v are separated by their lowest common ancestor, a contradiction to Property 3 of the tree decomposition.

The vertices mapped to nodes for our running example are highlighted in pink in Fig. 3. To facilitate the following discussions, given a vertex v, its mapped node in T is defined as:

$map_{\mathsf{T}}(v) = \text{argmin}_{X_i \ni v, X_i \in \mathcal{X}} depth_{\mathsf{T}}(X_i)$.

Thus, given two vertices v and u, the Separator Pinpointing Problem is essentially locating in their labels the information corresponding to the lowest common ancestor of $map_{\mathsf{T}}(v)$ and $map_{\mathsf{T}}(u)$ in T. However, there are two issues at this point. First, the label of a vertex v after Algorithm 1 may not contain full and accurate information to every tree node that is the ancestor of $\text{map}_{\mathsf{T}}(v)$. Second, although lowest common ancestor (LCA) problem has been studied with theoretical solutions of $O(n)$ label size and $O(1)$ constant query time [5,22], and a practical solution of $O(n \log n)$ label size and $O(1)$ constant query time [5], these solutions have not demonstrated the ability to support efficiently answering distance queries in a distributed environment. In fact, neither is the most efficient solution for the Separator Pinpointing Problem.

The first issue can be easily resolved by querying vertices using L_G to update the label information, which we will discuss in more details in Sect. 5. In the rest of this section, we focus on the second issue. We will show that in most cases, by assigning one machine word (64-bit) to each vertex, we can pinpoint the information related to the LCA separator in constant time for any two vertices. However, this does not mean that we propose a better solution for the general LCA problem. In fact, the LCA problem for our case is quite specific as the tree T is not an arbitrary tree. More importantly, our LCA solution supports distributed distance query mechanisms as we will show later.

To achieve the above goal, we first convert the tree T into a binary tree T' in the following two steps.

Step 1. For each non-leaf node which contains only one child, without loss of generality, we make the single child as the left child of the non-leaf node.

Step 2. For each non-leaf node which contains more than two children, build a sub-full binary tree connecting the non-leaf node to its children in a similar way as building a Huffman tree: Initially, each child is assigned a weight equal to the size (i.e., the number of nodes) of the subtree rooted at the child. Then, we iteratively replace two nodes with smallest summed weight by one *dummy* node whose weight is the summed weight of the two (the replacement also makes the dummy node father of the two nodes), until only two nodes are left. Then, we make the non-leaf node as the father of the two remaining nodes (we use *real* nodes in the following to emphasize non-dummy nodes).

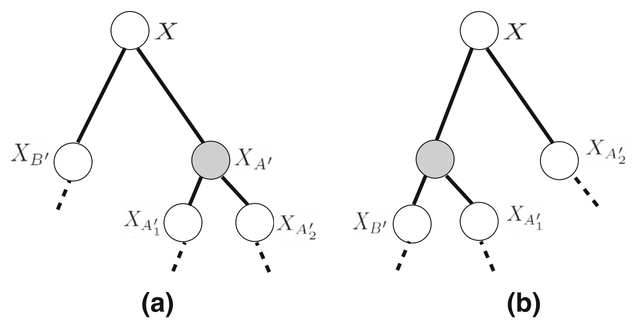

Fig. 4 Proof illustration for Lemma 3. *Gray* nodes are dummy nodes. *White* nodes can be dummy nodes or real nodes

In the above construction, if there is more than one node without a parent, these nodes are considered the children of a dummy root. By this way, we handle a forest which is a result of disconnected components in the original graph.

The T' for our running example is illustrated in Fig. 3 by black solid lines and green dashed lines. By recalling the Huffman tree construction procedure, we can easily conclude that the above procedure will introduce a limited number of dummy nodes, as described in the following observation:

Observation 2 *The number of dummy tree nodes is no more than the number of the real tree nodes in* T'.

In the following, we will show that the number of bits in a tree label can be made equal to the heigh of T' to answer LCA queries. Thus, the height of T' becomes the key. If it is excessively large, then *LCA* finding algorithm will take a significant amount of time and the index size will also be large.

The height increase in T' over T depends on the heights of the sub-full binary trees constructed in Step 2. In the worst case scenario, a Huffman coding tree can have a height linear to the number of nodes. That means, an ID could have $O(n)$ bits, which is essentially $O(n)$ numbers. Fortunately, in our case, the height increase is bounded tightly as concluded in Theorem 1. To prove Theorem 1, we first prove the following lemma (Fig 4).

Lemma 3 *Assume X is a dummy node or a real node in T' with 2 children $X_{A'}$ and $X_{B'}$. Assume $X_{A'}$ is a dummy node with two children $X_{A'_1}$ and $X_{A'_2}$. Then, the subtree rooted at $X_{A'}$ contains no more than $\frac{2}{3}$ real nodes of the total real nodes that are the descendants of X. In addition, none of the subtrees rooted at $X_{A'_1}$ or $X_{A'_2}$ contains more than $\frac{1}{2}$ real nodes of the total real nodes that are the descendants of X.*

Proof Let A' and B' denote the two subtrees rooted at $X_{A'}$ and $X_{B'}$, and let $\mathcal{X}_{A'}$ and $\mathcal{X}_{B'}$ denote the sets of real nodes they contain. Since $X_{A'}$ is a dummy node, it has exactly two children according to the construction of T', letting them be $X_{A'_1}$ and $X_{A'_2}$, and $\mathcal{X}_{A'_1}$ and $\mathcal{X}_{A'_2}$ be the sets of real nodes that are their descendants. In the following, we prove this lemma by contradiction.

Assume $\frac{|\mathcal{X}_{A'}|}{|\mathcal{X}_{A'}|+|\mathcal{X}_{B'}|} > \frac{2}{3}$. Recall $|\mathcal{X}_{A'}| = |\mathcal{X}_{A'_1}| + |\mathcal{X}_{A'_2}|$. Without loss of generally, let us assume $|\mathcal{X}_{A'_2}| \geq |\mathcal{X}_{A'_1}|$. Then we have $\frac{|\mathcal{X}_{A'_2}|}{|\mathcal{X}_{A'}|+|\mathcal{X}_{B'}|} > 1/3 > \frac{|\mathcal{X}_{B'}|}{|\mathcal{X}_{A'}|+|\mathcal{X}_{B'}|}$. This implies that, according to the Step 2 of T' construction, $X_{A'_1}$ and $X_{B'}$ shall be made as siblings, a contradiction to the fact that $X_{A'_1}$ and $X_{A'_2}$ are siblings.

In a similar way, we can show that none of the subtrees rooted at $X_{A'_1}$ and $X_{A'_2}$ contains more than $\frac{1}{2}$ real nodes of the total real nodes that are the descendants of X. □

Theorem 1 *The height of T' is no more than $2\log_2 n$.*

Proof Assume $P' = (X_1, X_2, \ldots, X_h)$ is a longest path from the root to a leaf in T'. Correspondingly, assume $\mathcal{X}_i$ is the set of real nodes contained by the subtree rooted at X_i in T'. If both X_i and X_{i+1} are real nodes, then we conclude $\frac{|\mathcal{X}_{i+1}|}{|\mathcal{X}_i|} \leq \frac{1}{2}$ according to the fact that X_i is a centroid of the subtree in $\mathcal{T}$, induced by X_i and the real nodes that are descendants of X_i in T'. If X_i and X_{i+j} are two real nodes with j dummy nodes between them, then, according to Lemma 3, we can conclude that $\frac{|\mathcal{X}_{i+j}|}{|\mathcal{X}_i|} \leq \frac{1}{(\sqrt{2})^j}$. Therefore, the total number of nodes on P' is no more than $\log_{\sqrt{2}} n = 2\log_2 n$. □

Theorem 1 tells us that T' is a height-bounded tree. Motivated by LCA for complete binary trees [22], we propose Algorithms 2 and 3 to label tree nodes and answer LCA queries on this height-bounded tree. By checking the labels of any two nodes plus their depths, we can find their common ancestors in constant time. The tree label assignment is accomplished by calling Algorithm 2 with parameters $Y = root(\mathsf{T}')$, $n' = 2^{height(\mathsf{T}')}$, and $l = 0$. The tree labels for our running example is shown in Fig. 3 as a bit number aside each node.

Algorithm 2 TREELABEL(T', Y, n', l)

```
1: Label $Y$ with $n'$;
2: Let $Y_{left}, Y_{right}$ be the left and right children of $Y$ on $\mathsf{T}'$, respectively.
3: $n'_{left} = n'$; $n'_{right} = n'$;
4: Turn the $l+1$ th bit (from left to right) in number $n'_{right}$ into 1;
5: Turn the $l$ th bit (from left to right) in number $n'_{left}$ into 0;
6: Turn the $l+1$ th bit (from left to right) in number $n'_{left}$ into 1;
7: TreeLabel($\mathsf{T}', Y_{left}, n'_{left}, l+1$);
8: TreeLabel($\mathsf{T}', Y_{right}, n'_{right}, l+1$);
```

The key idea of this labeling process is to assign labels that satisfy two conditions: (1) For any two tree nodes such that one is not the ancestor of the other, the two nodes have labels, from the left to right, exactly the same $l-1$ consecutive binary bits and a different binary bit at l-th position where l is the depth of their common ancestor; (2) For any two tree nodes that one at depth l is the ancestor of the other, the two nodes have labels, from the right to the left, exactly the same l consecutive binary bits. Thus, it is quite straight forward to use a few simple logic operations to find out the lowest common ancestor of two tree nodes as described in Algorithm 3. As we will see later in the proof of Theorem 3, the construction and labeling of T' takes only $O(n \log n)$ time to complete.

Algorithm 3 LOWESTCOMMONANCESTOR(Y_i, Y_j)

```
1: if $depth_{\mathsf{T}'}(Y_i) \le depth_{\mathsf{T}'}(Y_j)$ then
2:     $DLCA_{\mathsf{T}'}(Y_i, Y_j) = depth_{\mathsf{T}'}(Y_i)$; $LCA_{\mathsf{T}'}(Y_i, Y_j) = Y_i$;
3: else
4:     $DLCA_{\mathsf{T}'}(Y_i, Y_j) = depth_{\mathsf{T}'}(Y_j)$; $LCA_{\mathsf{T}'}(Y_i, Y_j) = Y_j$;
5: end if
6: if $L_{\mathsf{T}'}(Y_i) \gg (height(\mathsf{T}') + 1 - DLCA_{\mathsf{T}'}(Y_i, Y_j)) \neq$
   $L_{\mathsf{T}'}(Y_j) \gg (height(\mathsf{T}') + 1 - DLCA_{\mathsf{T}'}(Y_i, Y_j))$ then
7:     $DLCA_{\mathsf{T}'}(Y_i, Y_j) = \lfloor \log_2(L_{\mathsf{T}'}(Y_i) \oplus L_{\mathsf{T}'}(Y_j)) \rfloor$;
8:     $LCA_{\mathsf{T}'}(Y_i, Y_j) = ((L_{\mathsf{T}'}(Y_i) | L_{\mathsf{T}'}(Y_j)) \gg DLCA_{\mathsf{T}'}(Y_i, Y_j))$
           $\ll DLCA_{\mathsf{T}'}(Y_i, Y_j))$;
9: end if
10: return $LCA_{\mathsf{T}'}(Y_i, Y_j), DLCA_{\mathsf{T}'}(Y_i, Y_j)$
```

Let $L_{\mathsf{T}'}(Y)$ denote a tree label assigned to a node Y of T' by the TREELABEL Algorithm. Given two nodes Y_i and Y_j, Let

$DLCA_{\mathsf{T}'}(Y_i, Y_j)$ and $LCA_{\mathsf{T}'}(Y_i, Y_j)$ denote the depth of their lowest common ancestor, and the ID of their lowest common ancestor, respectively. Then, we have the following lemma:

Lemma 4 *Given two nodes Y_i and Y_i of tree T', Algorithm* 3 *correctly calculate $LCA_{\mathsf{T}'}(Y_i, Y_j)$.*

We provide an example in the following to facilitate the understanding of the algorithm.
Example: $map_{\mathsf{T}'}(h)$ and *$map_{\mathsf{T}'}(l)$*:

Since $depth_{\mathsf{T}'}(map_{\mathsf{T}'}(h)) > depth_{\mathsf{T}'}(map_{\mathsf{T}'}(l))$, we have: $DLCA_{\mathsf{T}'}(map_{\mathsf{T}'}(h), map_{\mathsf{T}'}(l)) = depth_{\mathsf{T}'}(map_{\mathsf{T}'}(l)) = 3$, and $LCA_{\mathsf{T}'}(map_{\mathsf{T}'}(h), map_{\mathsf{T}'}(l)) = map_{\mathsf{T}'}(l) = 00010$; Since $L_{\mathsf{T}'}(map_{\mathsf{T}'}(h)) \gg (4 + 1 - 3) = 011$ while $L_{\mathsf{T}'}(map_{\mathsf{T}'}(l)) \gg (4 + 1 - 3) = 000$, we have: $DLCA_{\mathsf{T}'}(map_{\mathsf{T}'}(h), map_{\mathsf{T}'}(l)) = \lfloor \log_2(01110 \oplus 00010) \rfloor = 3$, and $LCA_{\mathsf{T}'}(map_{\mathsf{T}'}(h), map_{\mathsf{T}'}(l)) = ((01110|00010) \gg 2^3) \ll 2^3 = 01000$.

Using the *LCA* function defined above, we can identify the depth of the lowest common ancestor of two nodes Y_i and Y_j in T'. However, we shall note that $LCA_{\mathsf{T}'}(Y_i, Y_j)$ may not be the lowest common ancestor node of Y_i and Y_j in T. It may happen that $LCA_{\mathsf{T}'}(Y_i, Y_j)$ is a dummy node. To resolve this problem, we need a mapping table to convert labels in T' to a real tree node id. Let us name the table *RealNodeMap*. It will map the bit label of a real node in T' to its tree node id. For the bit label of a dummy node in T', *RealNodeMap* will map it to the real node id that is the lowest ancestor of the dummy node in T'. In the case a dummy node does not have a real node ancestor (this happens when the original graphs consists of several disconnected subgraphs, and the root of T' is a dummy node), we let *RealNodeMap* map to an invalid number, say -1. In the following, we will see that in some cases, mapping to the depth of a real node is all that is needed, and in these cases, we use *RealNodeDepthMap* instead.

Finally, since the height of T' is no more than twice of T and $L_{T'}$ needs height(T')+1 bits, we conclude that a 64-bit machine word can support at least 2^{31} real nodes, thus at least $2^{31} \approx 2G$ vertices in the original graph. This is sufficient for most real applications. In extreme cases, including real-world scenarios, using two 64-bit machine words for a tree label will be more than sufficient.

Table 3 The vertex information table of the running example for exact distance queries

v	$L_{\mathsf{T}'}(map_{\mathsf{T}'}(v))$	$depth_{\mathsf{T}'}(map_{\mathsf{T}'}(v))$	*offset**
a	00110	3	0
b	01000	1	0
c	01000	1	1
g	01010	3	0
h	01110	3	0
l	00010	3	0
o	11010	3	0
p	10100	2	0
q	10010	3	0
r	10000	0	0
s	10100	2	1
t	10100	2	2
w	10000	0	1
z	10000	0	2

* *offset* is used for TREEMAP-C method only

5 Final label of each vertex and labeling upper bounds

With the ability to efficiently identify LCA in T', we can reorganize L_G, obtained from Algorithm SEPARATORLABELING, into $L_{G\mathsf{T}'}$ such that a vertex v's label $L_{G\mathsf{T}'}(v)$ is organized in segments corresponding to the tree node separators associated with v. $L_{G\mathsf{T}'}$ for our running example is listed in Table 4 (left part). The reorganization can be done by distance queries on the L_G table, as we have discussed at the end of Sect. 3. Since this is not the actual query but part of the label construction, we refer to it as *prequery* in the following.

In order to pinpoint the segment in the label of a vertex for an exact distance query, a vertex v also needs to record information including $L_{\mathsf{T}'}(map_{\mathsf{T}'}(v))$ and $depth_{\mathsf{T}'}(map_{\mathsf{T}'}(v))$. Table 3 shows the vertex info needed for our running exam-

Table 4 $L_{GT'}$ table for the running example

Vertex	$L_{GT'}$ for TreeMap-F				$L_{GT'}$ for TreeMap-C			
	0	1	2	3	0	1	2	3
a	$(r,2),(w,3),(z,2)$	$(b,1),(c,1),(r,2)$	$(a,0),(b,1),(c,1)$		$(r,2),(w,3),(z,2)$	$(b,1),(c,1)$	$(a,0)$	
b	$(r,1),(w,2),(z,2)$	$(b,0),(c,1),(r,1)$			$(r,1),(w,2),(z,2)$	$(b,0),(c,1)$		
c	$(r,1),(w,2),(z,1)$	$(b,1),(c,0),(r,1)$			$(r,1),(w,2),(z,1)$	$(b,1),(c,0)$		
g	$(r,2),(w,3),(z,2)$	$(b,2),(c,1),(r,2)$	$(c,1),(r,2),(z,2)$	$(c,1),(g,0)$	$(r,2),(w,3),(z,2)$	$(b,2),(c,1)$		$(g,0)$
h	$(r,1),(w,2),(z,2)$	$(b,2),(c,1),(r,1)$	$(c,1),(r,1),(z,2)$	$(c,1),(h,0),(r,1)$	$(r,1),(w,2),(z,2)$	$(b,2),(c,1)$		$(h,0)$
l	$(r,2),(w,3),(z,3)$	$(b,1),(c,2),(r,2)$	$(b,1),(l,0)$		$(r,2),(w,3),(z,3)$	$(b,1),(c,2)$	$(l,0)$	
o	$(r,2),(w,1),(z,1)$		$(w,1),(z,1)$	$(o,0),(w,1),(z,1)$	$(r,2),(w,1),(z,1)$			$(o,0)$
p	$(r,1),(w,1),(z,2)$		$(p,0),(s,2),(t,2)$		$(r,1),(w,1),(z,2)$		$(p,0),(s,2),(t,2)$	
q	$(r,2),(w,2),(z,3)$		$(p,1),(s,1),(t,2)$	$(p,1),(q,0),(s,1)$	$(r,2),(w,2),(z,3)$		$(p,1),(s,1),(t,2)$	$(q,0)$
r	$(r,0),(w,1),(z,1)$				$(r,0),(w,1),(z,1)$			
s	$(r,2),(w,3),(z,3)$		$(p,2),(s,0),(t,1)$		$(r,2),(w,3),(z,3)$		$(p,2),(s,0),(t,1)$	
t	$(r,1),(w,2),(z,2)$		$(p,2),(s,1),(t,0)$		$(r,1),(w,2),(z,2)$		$(p,2),(s,1),(t,0)$	
w	$(r,1),(w,0),(z,1)$				$(r,1),(w,0),(z,1)$			
z	$(r,1),(w,1),(z,0)$				$(r,1),(w,1),(z,0)$			

Left part and right part are for TreeMap-F and TreeMap-C methods, respectively. All vertices IDs in the pareses are listed for readers' convenience. They do not need to exist in the final labels

ple. Besides these labels, we also need the $RealNodeMap$ table and the height of T' for exact distance queries.

Since for a vertex v, $L_{GT'}$ may contain repetitive information as shown in Table 4 (left part), we propose to further compress the label size by dropping the repetitive information. Table 4 (right part) shows $L_{GT'}$ with repetitive information dropped. However, in order to support efficient exact distance queries under this condition, a vertex shall also keep the *offset* in its information table, as shown in Table 3. The *offset* of a vertex v is defined as v's order (≥ 0) of being visited by BFS-Labeling in the node $map_{\mathsf{T}}(v)$. Since this method creates a compact label $L_{GT'}$ table, we name it TreeMap-C (i.e., Compact TreeMap). For the method described above with redundant information in $L_{GT'}$, we name it TreeMap- F (i.e., Fast TreeMap). TreeMap- F is generally faster than TreeMap-C because it needs fewer memory accesses in answering an exact distance query.

In summary, the labels needed for answering exact distance queries by TreeMap- F and TreeMap-C are:

- TreeMap- F: (1) $L_{GT'}$; (2) vertex information table, including $L_{\mathsf{T}'}(map_{\mathsf{T}'}(v))$ and $depth_{\mathsf{T}'}(map_{\mathsf{T}'}(v))$; (3) $RealNodeDepthMap$; (4) height of T'.
- TreeMap-C: (1) $L_{GT'}$; (2) vertex information table, including $L_{\mathsf{T}'}(map_{\mathsf{T}'}(v))$, $depth_{\mathsf{T}'}(map_{\mathsf{T}'}(v))$, and *offset*; (3) $RealNodeMap$; (4) height of T' (5) node contents (i.e., $\mathcal{X}$) of tree decomposition $\mathcal{T}$.

The label size bounds for the two methods are concluded in the following theorem:

Theorem 2 *The total label size for* TreeMap- F *is no more than* $n * (tw + 1) * \log_2 n + 4n + 1$, *and for* TreeMap-C *is no more than* $n * (tw + 1) * (\log_2 n + 1) + 5n + 1$.

Proof To prove this theorem, we only need to list the bound for each part as follows:

(1) $L_{GT'}$ is no more than $n(tw + 1)\log_2 n$ according to the analysis at the end of Sect. 3;
(2) Vertex information table takes $2n$ numbers for TreeMap-F, and $3n$ numbers for TreeMap-C;
(3) $RealNodeDepthMap$ or $RealNodeMap$ takes no more than $2n$ numbers using a hash table (recalling Observation 2);
(4) height of T' uses one number;
(5) node contents takes $n(tw + 1)$ numbers. □

Although TreeMap-C has a higher label size upper bound than TreeMap-F, we will observe in the empirical study that the label size for TreeMap-C is significantly smaller than TreeMap-F in most cases. This is because $L_{GT'}$ is the dominant factor of label size and TreeMap-C significantly reduces it.

In the rest of this section, we will prove Theorem 3 which states that the construction time of our approach is bounded. We excluded the tree decomposition time from the theorem because users have multiple choices for implementing the tree decomposition. Wei [38] presents a linear method and our implementation generally follows this strategy and its worst case time cost complexity is no larger than that of TreeMap.

Theorem 3 *Given a tree decomposition* $(\mathcal{X}, \mathcal{T})$*, the total construction time of* TREEMAP *is* $O(tw^2 * n * \log^2 n + tw * m * \log n)$.

Proof To prove this theorem, let us first recall the steps needed for label construction:

Step 1. Construct L_G, and save *offset* information for TREEMAP-C;
Step 2. Construct vertex information table, $RealNodeDepthMap$ or $RealNodeMap$;
Step 3. Construct $L_{G\mathsf{T}'}$.

Thus, we can complete the proof of this theorem by showing the following three claims are correct.

Claim 1 *Step* 1 *finishes in no more than* $O(tw^2 * n * \log^2 n + tw * m * \log n)$ *time.*

Proof As we have discussed in Sect. 2.2, the separator assignment can be completed in $O(n \log n)$ time which can be ignored with respect to the time complexity to be proved. Now let us consider the fact that a vertex is deleted after BFSLabeling is applied to it. When all vertices in a tree bag are deleted, the graph is separated by the tree bag. Thus, we conclude that each vertex and each edge will be visited by BFSLabeling for at most $tw \log n$ times, leading to $O(tw * (n + m) * \log n)$ time cost. We also need to sort the vertex–distance pair in L_G to facilitate query on it, taking $O(n * tw * \log tw * \log n * \log \log n)$. □

Claim 2 *Step* 2 *finishes in* $O(n * \log n)$ *time.*

Proof The T' construction follows the Huffman tree construction algorithm in creating dummy nodes. The Huffman tree construction algorithm is well known to take $O(klogk)$ time where k is the number of symbols. Thus, we conclude that T' construction takes $O(n * \log n)$ time. The remaining part of Step 2, including Algorithm 2, and subsequently building vertex information table, $RealNodeDepthMap$ or $RealNodeMap$, clearly takes linear time. □

Claim 3 *Step* 3 *finishes in* $O(tw^2 * n * \log^2 n)$ *time.*

Proof The construction of $L_{G\mathsf{T}'}$ is a prequery process on L_G. For each vertex, it will needs at most $(tw + 1) \log n$ prequeries to fill up the $L_{G\mathsf{T}'}$ table. Each prequery will finish in $O(tw \log n)$ as we discussed at the end of Sect. 3. Thus, the total time cost is $O(n * tw^2 * \log^2 n)$. □

6 Answering queries

The query answering process is quite simple. Algorithm 4 is the pseudocode for answering queries by TREEMAP-F. The first part (1–6) of the algorithm pinpoints the separator depth information in the labels of the two vertices. The second part (7–12) calculates the shortest distance.

Algorithm 4 QUERY- TREEMAP- F(u, v)

```
1: Call Algorithm LOWESTCOMMONANCESTOR to obtain
   $lca\_treeid = LCA_{\mathsf{T}'}(map_{\mathsf{T}'}(u), map_{\mathsf{T}'}(v))$;
   $level = DLCA_{\mathsf{T}'}(map_{\mathsf{T}'}(u), map_{\mathsf{T}'}(v))$;
2: if $RealNodeDepthMap(lca\_treeid) = invalid$ then
3:     return $+\infty$;
4: else
5:     $level = RealNodeDepthMap(lca\_treeid)$;
6: end if
7: $distance = +\infty$;
8: for $i = 0$ to $|L_{G\mathsf{T}'}(u)[level]| - 1$ do
9:     if $distance > L_{G\mathsf{T}'}(u)[level][i] + L_{G\mathsf{T}'}(v)[level][i]$ then
10:        $distance = L_{G\mathsf{T}'}(u)[level][i] + L_{G\mathsf{T}'}(v)[level][i]$;
11:    end if
12: end for
13: return $distance$;
```

The query of TREEMAP-C follows the similar work flow except that it first locates the real LCA node of $map_{\mathsf{T}'}(u)$ and $map_{\mathsf{T}'}(v)$ and then uses the vertices in this node to guide the label comparisons. This explains why TREEMAP-C needs the node contents (i.e., $\mathcal{X}$) of the tree decomposition $\mathcal{T}$. Subsequently, TREEMAP-C uses $depth_{\mathsf{T}'}(map_{\mathsf{T}'}(x))$ and *offset* in the vertex information table (see Table 3) to locate the needed distance information in a vertex label. Algorithm 5 is the query pseudocode of TREEMAP-C.

Algorithm 5 QUERY- TREEMAP- C(u, v)

```
1: Call Algorithm LOWESTCOMMONANCESTOR to obtain
   $lca\_treeid = LCA_{\mathsf{T}'}(map_{\mathsf{T}'}(u), map_{\mathsf{T}'}(v))$;
   $level = DLCA_{\mathsf{T}'}(map_{\mathsf{T}'}(u), map_{\mathsf{T}'}(v))$;
2: if $RealNodeMap(lca\_treeid) = invalid$ then
3:     return $+\infty$;
4: else
5:     $X = \mathcal{X}(RealNodeMap(lca\_treeid))$;
6: end if
7: $distance = +\infty$;
8: for $x \in X$ do
9:     if $distance > L_{G\mathsf{T}'}(u)[depth_{\mathsf{T}'}(map_{\mathsf{T}'}(x))][offset(x)] + L_{G\mathsf{T}'}(v)[depth_{\mathsf{T}'}(map_{\mathsf{T}'}(x)][offset(x)]$ then
10:        $distance = L_{G\mathsf{T}'}(u)[depth_{\mathsf{T}'}(map_{\mathsf{T}'}(x))][offset(x)] + L_{G\mathsf{T}'}(v)[depth_{\mathsf{T}'}(map_{\mathsf{T}'}(x)][offset(x)]$;
11:    end if
12: end for
13: return $distance$;
```

To facilitate the understanding of the query algorithms, we provide a query example of TREEMAP-F as follows:

Example Query the distance between o and q on the running example. First, by calling LOWESTCOMMONANCESTOR, we obtain $lca_{treeid} = 11{,}000$ (line 1). Then, by looking at $RealNodeDepthMap$, we get $level = 0$ (line 5). Now, we can start the distance calculation by looking at the left part of Table 4: $distance(o, q) = \min(2+2, 1+2, 1+3) = 3$. The correctness of the query algorithms and their time complexities is stated in Theorem 4.

Theorem 4 TREEMAP-F *and* TREEMAP-C *correctly answer distance query on any two vertices u and v in $O(tw)$ time.*

Proof According to Lemma 2, we can use prequery to correctly build the exact distance information in $L_{GT'}$, and according to Lemma 4, we can correctly locate in $L_{GT'}$ the distance information corresponding to the LCA node separator for the two vertices. Thus, we can correctly calculate the distance between u and v because any shortest path between u and v must pass the LCA node separator.

The time complexity analysis is quite straight forward. The LOWESTCOMMONANCESTOR algorithm contains a constant number of standard computing operations, and thus, the time complexity is constant. The distance calculation (i.e., 7–12 of Algorithms 4 and 5) loops no more than $tw + 1$ times, and each time the calculation involves a constant number of direct memory accesses plus an addition operation. Thus, the time complexity is $O(tw)$. □

7 Using bit status to speed up query processing in directed graphs

In the above sections, we have presented TreeMap-F and TreeMap-C for answering exact distance queries on undirected graphs. In fact, our approach is more suitable for efficiently answering queries on directed graphs. To understand this, let us first see what special treatment we need for indexing a directed graph.

For the tree decomposition, we can always treat a directed graph as an undirected one by ignoring the edge direction. The labels for directed graphs shall be separated into in and out. Correspondingly, we will have L_{in_G} and L_{out_G} in replacement of L_G, and $L_{in_{GT'}}$ and $L_{out_{GT'}}$ in replace of $L_{GT'}$. Thus, in the L_{in_G} and L_{out_G} construction, the BFSLabeling starts twice on a vertex, following the outgoing edges and then following the incoming edges. In addition, distance prequery from u to v will be performed between $L_{out_G}(u)$ and $L_{in_G}(v)$, and distance query will be between $L_{out_{GT'}}(u)$ and $L_{in_{GT'}}(v)$.

An important advantage of our approach is that we can use a *bit status* of at most $tw+1$ bit for a separator segment s of a vertex label v, recording whether v can reach a vertex in the separator (the corresponding bit is set to 1 in $L_{out_G}(v)(s)$), or can be reached by a vertex in the separator (the corresponding bit is set to 1 in $L_{in_G}(v)(s)$). Thus, we can use the bit AND operation to quickly exclude the unreachable case during an exact distance query. This explains why our methods TreeMap-F and TreeMap-C perform extremely well on the directed graphs in the empirical study.

Finally, it is easy to see that the construction time complexity (Theorem 3) and the query time complexity (Theorem 4) for directed graphs remain the same as the undirected graphs. However, the index size bound of Theorem 2 shall be fixed as follows:

Corollary 1 *For directed graphs, the total label size for* TREEMAP-F *is no more than* $4n*(tw+1)*\log_2 n + 4n + 1$, *and for* TREEMAP-C *is no more than* $n*(tw+1)*(4\log_2 n + 1) + 5n + 1$.

8 Empirical study

We are interested in the performance of TREEMAP-F and TREEMAP-C for answering exact distance queries on both directed and undirected real graphs. To understand how well TREEMAP-F and TREEMAP-C perform with respect to other competitive methods, we chose the highway-centric labeling (HighwayCentric) [23] as a benchmark for exact distance queries on directed graphs and TEDI [38] as a benchmark for exact distance queries on undirected graphs. The two methods are good representations of the state-of-the-art distance indexing schemes for directed and undirected graphs, respectively. We implemented TREEMAP-F and TREEMAP-C in C++. The main platform for our empirical study is a Linux cluster with 2.4 GHz AMD Opteron processors.

We used the original source code developed by the authors of HighwayCentric and TEDI in the following study. To ensure no error was introduced by us to the two methods, we did not make any essential changes on the original code of HighwayCentric and TEDI, and we tested them under their default parameters. For TEDI, k for each dataset was selected from the original publication. The "North America" is a new dataset, and since its size is comparable to "bay", we apply bay's parameter $k = 80$ to "North America". The index size of TREEMAP-F or TREEMAP-C is the total numbers contained in the labels of TREEMAP-F or TREEMAP-C (See Sect. 5). For HighwayCentric, the source code counts the index size as the total numbers contained in L_{in} and L_{out}. TEDI's source code does not output an index size count so we did not have an opportunity to report it in our tests, but readers can refer to its original publication [38] for this information. Meanwhile, TEDI's source code does not output an overall construction time but a stack building time and a tree construction time, and we add them together as TEDI's construction time.

The query time listed for each test is the total time recorded by the corresponding program for answering 100,000 random distance queries. Query results by TREEMAP-F and

Table 5 Characters of real datasets

For directed query tests			For undirected query tests		
Dataset	# *V*	# *E*	Datset	# *V*	# *E*
aMaze	11,877	28,700	Dutsch	3,621	4,310
Citeseer	693,947	312,282	Eva	4,475	4,652
HpyCyc	5,565	8,474	Geom	3,621	9,461
Kegg	14,271	35,170	Homo	7,020	19,811
P2PG08	4,055	12,443	Inter	22,442	45,550
P2PG09	4,179	11,944	Pfei	1,738	1,876
Reactome	3,678	14,447	PPI	1,458	1,948
Vchocyc	10,694	14,207	Yeast	2,284	6,646
Wiki-Vote	7,115	103,689	Bay	321,270	397,415
Xmark	6,483	7,651	North America	175,813	179,102

TreeMap-C were checked against the results by standard BFS approach to ensure our implementation of TreeMap-F and TreeMap-C correctly answered exact distance queries.

We compare TreeMap-F and TreeMap-C with HighwayCentric and TEDI on 20 real datasets, in which 10 for query tests on directed graphs and the other 10 for query tests on undirected graphs. These datasets have been used in [10,23,38,40]. The basic characters of these datasets, including number of vertices and edges, are listed in Table 5.

8.1 Exact distance queries on directed graphs

The test results of TreeMap-F, TreeMap-C, and HighwayCentric on directed real graphs are listed in Table 6. From this table, we can make several exciting observations.

First, the query time of TreeMap-F and TreeMap-C are much faster than HighwayCentric for most of the tests. The average speedup of TreeMap-F and TreeMap-C over HighwayCentric are 21 times and 6.4 times, respectively. In particular, TreeMap-F reaches a speedup of 48 times over HighwayCentric on the dataset Wiki-Vote. In addition, we can also observe that the TreeMap-F has a very stable performance on query time, which ranges between 29 and 66.6 ms over these datasets. In contrast, the query time performance of HighwayCentric is very susceptible over different datasets.

Second, the construction time of TreeMap-F and TreeMap-C is much shorter than HighwayCentric. On average, TreeMap-F and TreeMap-C are 962 times and 948 times faster than HighwayCentric in terms of index construction. For the Citeseer dataset, HighwayCentric takes more than 70 h to complete the indexing, while TreeMap-F and TreeMap-C take less than 1 min The sharp difference between the construction time can also be explained by their construction time complexities as listed in Table 1.

Table 6 Indexing and query test results on directed graphs

Dataset	*tw*	Query time (ms)			Index size			Construction time (ms)		
		TreeMap-F	TreeMap-C	HC	TreeMap-F	TreeMap-C	HC	TreeMap-F	TreeMap-C	HC
aMaze	264	59.3	177.4	1281.1	19,708,036	7,100,349	259,891	10,374	10,127	7,887,850
Citeseer	1	32.1	55.5	24.2	8,432,253	10,064,804	1,182,295	46,340	48,236	254,554,000
HpyCyc	116	29	40.5	49.3	2,568,018	1,483,405	23,336	605	605	55,693
Kegg	268	66.6	206.7	2180.5	25,851,927	8,932,977	4,12,189	15,901	15,573	14,432,100
P2PG08	807	45.6	188.9	2016.3	15,442,558	6,342,191	120,995	25,126	25,204	315,501
P2PG09	836	44.4	187.2	1879.6	16,462,625	6,391,024	121,983	28,204	28,244	367,697
Reactome	25	28.7	45.4	444.7	546,702	340,211	41,610	196	194	380,045
Vchocyc	149	32.8	45.9	46.4	5,488,794	3,506,380	37,655	1,506	1,559	138,288
Wiki-Vote	1,411	61.1	259.7	2,940.3	87,193,306	21,357,821	346,761	229,167	228,872	796,488
Xmark	55	31.3	43.8	82.9	2,356,274	921,637	25,617	367	344	114,056

HC HighwayCentric, *tw* is the width of the tree decomposition by TreeMap

Although TreeMap-F and TreeMap-C are much better than HighwayCentric in terms of query time and construction time, their index size are on average 102 times and 42 times the size of HighwayCentric. However, with the increasing of storage space, the index size is no longer as critical as the query time and the construction time. Even though the index size of TreeMap is much larger than HighwayCentric, the largest index size in the above table is just 87M (Wiki-Vote). Since we have proven that the index size of TreeMap is bounded, we are interested to know whether the index size of TreeMap is stable in practice.

To understand this, we perform the index and query tests on a scenario which HighwayCentric has successfully completed in [23]. In this scenario, five random graphs of 5k vertices were generated with densities ranging from 1.2 to 2.0. We plot the ratio of TreeMap index size over HighwayCentric in Fig. 5, and the ratio of TreeMap query time over HighwayCentric in Fig. 6. From these two figures, we can observe that TreeMap has a much faster query time but larger index size in comparison with HighwayCentric. But for the index size, the increase rate of TreeMap-F and TreeMap-C over density increases is much lower than HighwayCentric. The index sizes of TreeMap-F and TreeMap-C at density 2.0 are 2.9 and 3.1 times over the index sizes at density 1.2, respectively. In contrast, this ratio is 26.2 for HighwayCentric. The index size difference between TreeMap and HighwayCentric reduces significantly with the density increase. At the same time, the query speedup of TreeMap over HighwayCentric increases drastically with the density increases. These observations suggest that with respect to the index size and the query time, the performances of TreeMap-F and TreeMap-C are much stabler than HighwayCentric over density changes. However, we noticed that although TreeMap-F and TreeMap-C are much faster than HighwayCentric in index construction, the speedup decreases over the density increase as shown in Fig. 7. This can be explained by the construction time complexity of TreeMap in which tw^2 is an important factor. The tw of the five random graphs are 389, 585, 749, 900, and 1,068, respectively, from density low to high.

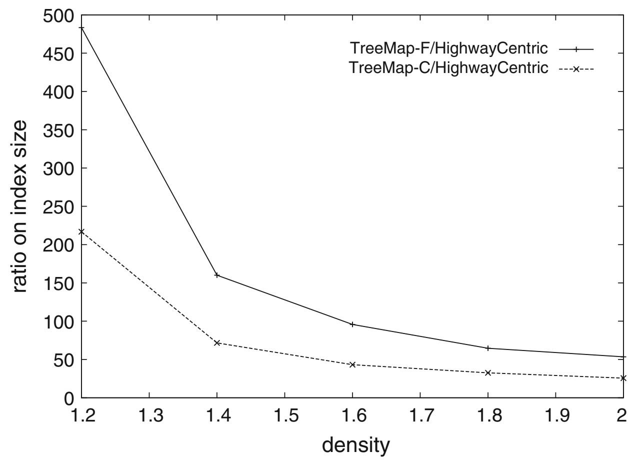

Fig. 5 The index size ratio of TreeMap over HighwayCentric

8.2 Exact distance queries on undirected graphs

In the above, we have seen the impressive performance of TreeMap in indexing and answering exact distance queries on directed graphs. We are also interested in its performance on undirected graphs. For this purpose, we compared TreeMap and TEDI over 10 datasets and the results are listed in Table 7. The TEDI's source code does not count the index size; therefore, we did not have an opportunity to record the results and list them in the table. Readers may refer to the original TEDI's paper [38] for index size information, which shows that the index size of TEDI is generally comparable to TreeMap.

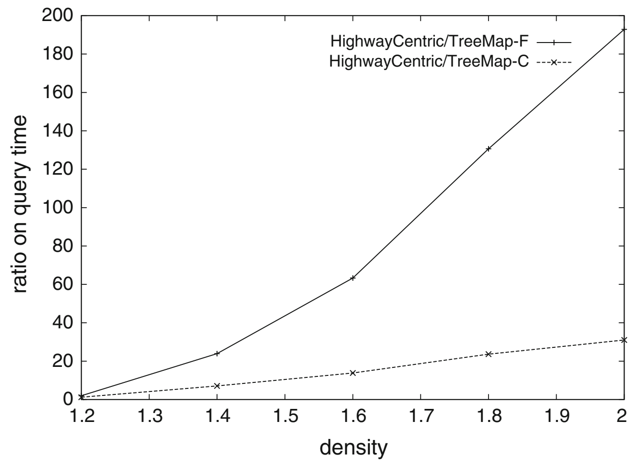

Fig. 6 The query time ratio of HighwayCentric over TreeMap

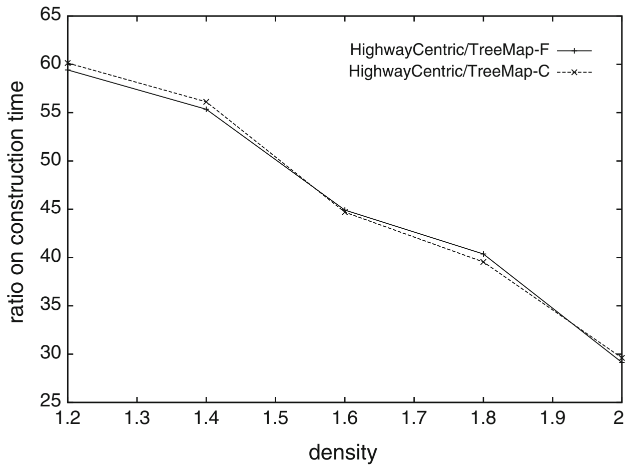

Fig. 7 The construction time ratio of HighwayCentric over TreeMap

Table 7 Indexing and query test results on undirected graphs

Dataset	tw	Query time (ms)			Index size		Construction time (ms)		
		TreeMap-F	TreeMap-C	TEDI	TreeMap-F	TreeMap-C	TreeMap-F	TreeMap-C	TEDI
Dutsch	110	30.7	36	135.7	1,029,861	469,847	405	253	126
Eva	12	25	26.4	91.6	169,949	109,892	65	61	68
Geom	168	34.9	41.8	136.1	1,910,690	695,009	916	500	595
Homo	757	81.3	108.8	286.8	18,986,645	5,862,447	30,660	13,028	7,546
Inter	180	48.2	51.9	202	15,003,355	4,795,658	8,015	3,940	7,984
Pfei	29	22.7	26.1	147.2	145,521	81,644	43	34	33
PPI	49	23.9	28.1	113.2	216,504	94,718	59	41	40
Yeast	270	39.2	50.1	110.8	1,692,084	687,732	1,186	696	353
Bay	88	62.9	60.3	15,508.5	94,952,805	69,323,281	82,170	65,241	4,161,168
NorthAmerica	40	53.5	49.3	694.1	22,257,221	17,239,415	15,172	13,155	205,798

tw is the width of the tree decomposition by TreeMap

In terms of the query time, TreeMap-F and TreeMap-C again demonstrate clear advantages over TEDI, especially on the large datasets "Bay" and "North America." TreeMap-F and TreeMap-C have an average 29 and 30 query time speedups over TEDI on these datasets, respectively. Again, TreeMap-F's query performance is quite stable over different datasets, but this is not the case for TEDI. Out of our expectation, in this group of studies, we observed that TreeMap-C has even a slightly better query performance than TreeMap-F on the large datasets "Bay" and "North America". We conjecture that this phenomenon was caused by the computer architecture. Because TreeMap-C has a much reduced index size than TreeMap-F, its index data is more likely to be cached for fast retrieval. This suggests that TreeMap-C is suitable for large real datasets. Next, we will study the performances of these methods on construction time.

By looking at the construction time columns of Table 7, we can see that TreeMap-F and TreeMap-C have a similar performance with TEDI on small datasets. However, on the two large datasets "Bay" and "North America," TreeMap-F and TreeMap-C perform much better than TEDI. On average, the construction time speedups of TreeMap-F and TreeMap-C are 6.9 and 8.7, respectively, over TEDI. This time we also observed TreeMap-C has a shorter construction time than TreeMap-F for all the datasets, while this is not the case in the directed graphs (Table 6). This is because for undirected graphs, there is no need to build the bit status (see Sect. 7) to speed up queries. Thus, TreeMap-C requires less prequeries to build up the $L_{GT'}$ than TreeMap-F does.

m-hop [10] also compares with TEDI, but its performance improvement is very limited. Given the performance of TreeMap-F and TreeMap-C, especially the latter, we conclude that they demonstrate an overall better performance than these competitive methods on the undirected real graphs.

8.3 Exact distance queries on very large graphs

To understand how TreeMap approaches perform on very large real graphs, we test TreeMap-F and TreeMap-C on three very large real graphs (over 1 million vertices) downloaded from Stanford Large Network Dataset Collection (http://snap.stanford.edu). The graph properties are listed in Table 8. We tested TEDI on the two undirected graphs roadNet-PA and roadNet-TX and HighwayCentric on the directed graph cit-Patents. Unfortunately, TEDI fails to run on the two undirected graphs, and HighwayCentric cannot finish running on the directed graph after a reasonable long time (over 80 h). Thus, we compare TreeMap with the standard BFS approach. Since bidirectional BFS (i.e., breath first search from both source and destination) is often considered a very fast approach and has been widely used in practice, we also compare TreeMap to bidirectional BFS. The result is listed in Table 8. Again, the query time listed is the total time for answering 100,000 random queries. All experiments are carried out on the same platform as above except for cit-Patents whose testing platform will be discussed in the following.

TreeMap-F and TreeMap-C is about five orders of magnitude faster than BFS and BiBFS on datasets roadNet-PA and roadNet-TX as shown in Table 8. It took TreeMap-F and TreeMap-C, around 60 min and 50 min, respectively, to finish index construction on roadNet-PA, and around 88 and 73 min , respectively, to complete index construction on roadNet-TX.

However, TreeMap test on cit-Patents does not work as well as the other two datasets. We found that the tree decomposition is a bottleneck for using the TreeMap approach. Our greedy implementation of tree decomposition fails to work on this very large graph. To circumvent the limitation of the tree decomposition, we take an alternative approach

Table 8 Indexing and query test results on very large graphs

Dataset	type	# V	# E	tw	Query time (ms)				Index size	
					TreeMap-F	TreeMap-C	BFS	BiBFS	TreeMap-F	TreeMap-C
roadNet-PA [27]	Indirected	1,088,092	1,541,898	396	106.4	99.9	8.2E+06	6.5E+06	1,613,662,761	1,155,631,022
roadNet-TX [27]	Undirected	1,379,917	1,921,660	308	96	90.6	1.3E+07	6.8E+06	1,528,325,792	1,090,136,877
cit-Patentsx [26]	Directed	3,774,768	16,518,947	N/A	*110.5	*149.5	2.9E+04	2.8E+04	*1,453,410,168	*1, 464, 324, 661

tw is the width of the tree decomposition by TreeMap
"BiBFS" stands for "Bidirectional BFS"
* are results of the combined approach of TreeMap and DLS

which combines both the Decentralizing Labeling Scheme (DLS) [39] and TreeMap. The main idea is that as soon as we detect the tree decomposition has a bag size greater than a threshold ts, we stop the decomposition and call DLS to decentralize a part of vertices. In our implementation, DLS keeps decentralizing the highest degree vertex until the vertex to be decentralize has a degree no more than 80 % of the first vertex decentralized in this round. Then, we restart the tree decomposition and repeat the process. When this index construction finishes, the index of the graph consists of both the DLS part and the TreeMap part. The distance query will be processed separately by DLS and TreeMap query algorithms, and the minimum distance is the answer. The correctness of this combined approach has been discussed in Sect. 3.

The purpose of the above approach is to circumvent the tree decomposition bottleneck. Since most of the real graphs we studied have a tw less than 1,000, we set the threshold ts = 1,000. By setting the experiment on a large memory Linux server with 2.1 GHz AMD Opteron processors, we successfully finished the experiments on cit-Patents with about 277 min construction time for TreeMap-F, and 268 min for TreeMap-C. In addition, we have observed that the peak physical memory consumption of our program (including tree decomposition, index construction, and answering queries) uses about 40 GB memory. The query time and index size are listed in Table 8. These results demonstrate that the combined approach still has a large speedup over the BFS and Bidirectional BFS on the query time.

9 TreeMap for distributed environment and weighted graphs

In addition to the main results described above, we will show in this section that TreeMap can be extended to support distance queries under distributed environments and distance queries on weighted graphs.

Distributed graph processing is becoming an important approach for handling extremely large graphs. Google Pregel [28], to give an example, is a famous distributed graph processing system developed recently. Many efforts have been made to develop distributed versions of classical graph algorithms, including basic algorithm elements used in TreeMap label construction such as tree decomposition, Breadth First Search approach (Algorithm 1), and Depth First traversal approach (Algorithm 2). Interested readers may refer to [4,19,41] for distributed versions of these algorithms. Providing a complete distributed construction of TreeMap is out of the scope of this work. However, we will show that we can extend the TreeMap-F method such that by assigning each vertex a compact label, we are able to answer a distance query on any two vertices using *only* the labels of these two vertices. As a consequence, TreeMap index can be segmented and saved in a distributed matter while answering distance queries can still be done efficiently by merely transmitting the compact labels of the vertices being queried. This mechanism is also known as "Distance Labeling Scheme" in some works (e.g., [12,18]).

By studying the index structure of TREEMAP-F, we can see that both the $L_{G\mathsf{T}'}$ table (Table 4) and the vertex information table (Table 3) can be divided and saved on a per vertex basis without affecting the query algorithm (Algorithm 4). The only centralized information, as can be observed in Algorithm 4, is the $RealNodeDepthMap$ data structure, which maps a node id on T', to the depth of its corresponding real node. However, in the following, we propose an alternative approach to locate this information via an additional short label assigned to each vertex.

To locate the depth of the lowest real node ancestor of $LCA_{\mathsf{T}'}(Y_i, Y_j)$ in T', we assign each vertex an ancestor mask variable whose number of bits is equal to the height of T'. Let $AncestorMask(v)$ denote such a variable for vertex v. Then, we assign 0 to ith bit in $AncestorMask(v)$, if the ancestor of $map_{T'}(v)$ at depth i is a dummy node, otherwise 1. Table 9 provides an example of $AncestorMask$ for each vertex. It is easy to see the construction of $AncestorMask$ can be incorporated into Algorithm 2 without increasing its time complexity. Now, for a given vertex v, if the ith bit of $AncestorMask(v)$ is 0, it implies its depth ith ancestor is a dummy node, and the depth of the lowest real node can be calculated by

$$f_{RealNodeDepth}(v, i) = i - (height(\mathsf{T}') - 1 - \lfloor \log_2(AncestorMask(v) \ll (height(\mathsf{T}') - 1 - i)) \rfloor).$$

"$\ll$" is a left shift operation which equals timing a power of 2. For example (Please refer to Table 9), for vertex p, the $AncestorMask$ bit of $map_{T'}(p)$ at depth 1 is 0, which implies the ancestor of $map_{T'}(p)$ at depth 1 is a dummy node. The lowest real ancestor of the dummy node is at depth:

$$1 - (5 - 1 - \lfloor \log_2(\star\star 101 \ll (5 - 1 - 1)) \rfloor) = 0.$$

In summary, for a vertex v, its complete label includes the following:

(1) $L_{G\mathsf{T}'}(v)$ (Table 4);
(2) $L_{\mathsf{T}'}(map_{\mathsf{T}'}(v))$ (Table 3);
(3) $depth_{\mathsf{T}'}(map_{\mathsf{T}'}(v))$ (Table 3);
(4) $AncestorMask(v)$ (Table 9);
(5) $height(\mathsf{T}')$, a constant.

Correspondingly, Algorithm 6 is the revised version of the TreeMap-F query algorithm (Algorithm 4) for answering queries based on the above vertex label format.

As an extension of Theorems 2 and 4, we have the following lemma:

Lemma 5 *By assigning a label of* $(tw + 1)\log_2 n + 4$ *size to each vertex in an undirected graph, we are able to answer distance query on any two vertices u and v in* $O(tw)$ *time using only their labels.*

Table 9 Ancestor mask for vertex information table (Table 3)

vertex	$depth_{\mathsf{T}'}(map_{\mathsf{T}'}(v))$	$AncestorMask$				
		4	3	2	1	0
a	3	⋆	1	0	1	1
b	1	⋆	⋆	⋆	1	1
c	1	⋆	⋆	⋆	1	1
g	3	⋆	1	1	1	1
h	3	⋆	1	1	1	1
l	3	⋆	1	0	1	1
o	3	⋆	1	1	0	1
p	2	⋆	⋆	1	0	1
q	3	⋆	1	1	0	1
r	0	⋆	⋆	⋆	⋆	1
s	2	⋆	⋆	1	0	1
t	2	⋆	⋆	1	0	1
w	0	⋆	⋆	⋆	⋆	1
z	0	⋆	⋆	⋆	⋆	1

⋆ Denotes an immaterial bit, i.e., being 0 or 1 does not affect our method

Algorithm 6 QUERY- TREEMAP- F- VERTEX- LABEL(u, v)

```
1: Call Algorithm LOWESTCOMMONANCESTOR to obtain
   $lca\_treeid = LCA_{\mathsf{T}'}(map_{\mathsf{T}'}(u), map_{\mathsf{T}'}(v))$;
   $level = DLCA_{\mathsf{T}'}(map_{\mathsf{T}'}(u), map_{\mathsf{T}'}(v))$;
2: if $f_{RealNodeDepth}(v, level) < 0$ then
3:     return $+\infty$;
4: else
5:     $level = f_{RealNodeDepth}(v, level)$;
6: end if
7: $distance = +\infty$;
8: for $i = 0$ to $|L_{G\mathsf{T}'}(u)[level]| - 1$ do
9:     if $distance > L_{G\mathsf{T}'}(u)[level][i] + L_{G\mathsf{T}'}(v)[level][i]$ then
10:        $distance = L_{G\mathsf{T}'}(u)[level][i] + L_{G\mathsf{T}'}(v)[level][i]$;
11:    end if
12: end for
13: return $distance$;
```

From the above analysis and Lemma 5, we can see that the distributed version of TreeMap has nearly the same total label size as TreeMap-F. However, the distributed version of TreeMap does not have the compact $L_{G\mathsf{T}'}(v)$ as the TreeMap-C does.

Finally, we will show that TREEMAP is almost ready for weighted graphs with very little modifications. For the tree decomposition, separator assignment, construction of T', and the subsequent vertex information table, we only need to treat the weighted graphs as unweighted. For weighted graphs with nonnegative weights, the separator labeling (Algorithm 1) needs to follow the order of Dijkstra's algorithm (time complexity: $O(m+n\log n)$) instead of BFS (time complexity: $O(m + n)$). The remaining part of index construction and query are essentially the same. The correctness of the method can be proved by a trivial extension of the proof of Lemma 2. It is interesting to observe the time complexity increase in Dijkstra's algorithm over BFS does not affect the overall time complexity of TreeMap construction (recall the proof of Theorem 3), and we conclude that all the analyses remain the same.

For weighted graphs with negative weights, TreeMap does not have a fast construction time because there is no fast single-source shortest paths algorithm that is available for the separator labeling algorithm.

10 Conclusion and future work

In this paper, we have proposed TreeMap, a very efficient graph indexing scheme that utilizes the tree decomposition to build and organize the labels for efficiently answering distance queries on both directed and undirected graphs. TreeMap is theoretical bounded in query time, index size, and construction time. The empirical study also demonstrates the huge advantages of TreeMap over state-of-the-art methods including Highway-Centric and TEDI.

We believe there are many possible fine tuning techniques that can make TreeMap perform even better. For example, we can also utilize the strategy of TEDI to introduce and maintain a balance between the tree decomposition and the distance matrix materialization for TreeMap.

Similar as other tree decomposition-based methods, a major limitation of TreeMap is that its performance is tied to the width of a tree decomposition. Consequently, TreeMap cannot effectively handle graphs with an extremely large treewidth. We are wondering if it is possible to effectively tackle this issue without compromising the accuracy of the results. Or, if it is possible to simplify the tree decomposition to support these "large treewidth" graphs by allowing for some errors. We are interested in investigating these questions in the future.

References

1. Abraham, I., Delling, D., Goldberg, A.V., Werneck, R.F.F.: Hierarchical hub labelings for shortest paths. In: ESA, pp. 24–35 (2012)
2. Agrawal, R., Borgida, A., Jagadish, H.V.: Efficient management of transitive relationships in large data and knowledge bases. In: SIGMOD Conference, pp. 253–262 (1989)
3. Arnborg, S., Corneil, D.G., Proskurowski, A.: Complexity of finding embeddings in ak-tree. SIAM J. Algebraic Discrete Methods **8**(2), 277–284 (1987)
4. Awerbuch, B.: A new distributed depth-first-search algorithm. Inf. Process. Lett. **20**(3), 147–150 (1985)
5. Bender, M.A., Farach-Colton, M.: The lca problem revisited. In: LATIN, pp. 88–94 (2000)
6. Bodlaender, H.: Planar graphs with bounded treewidth. Technical Report RUU-CS, (88–14) (1988)
7. Bodlaender, H.L.: Discovering treewidth. In: SOFSEM, pp. 1–16 (2005)
8. Bodlaender, H.L., Gilbert, H.L., Hafsteinsson, H., Kloks, T.: Approximating treewidth, pathwidth, frontsize, and shortest elimination tree. J. Algorithms **18**(2), 238–255 (1995)
9. Bodlaender, H.L., Koster, A.M.C.A.: Combinatorial optimization on graphs of bounded treewidth. Comput. J. **51**(3), 255–269 (2008)
10. Chang, L., Yu, J.X., Qin, L., Cheng, H., Qiao, M.: The exact distance to destination in undirected world. VLDB J. **21**(6), 869–888 (2012)
11. Cheng, J., Yu, J.X.: On-line exact shortest distance query processing. In: EDBT, pp. 481–492 (2009)
12. Chepoi, V., Dragan, F.F., Estellon, B., Habib, M., Vaxès, Y., Xiang, Y.: Additive spanners and distance and routing labeling schemes for hyperbolic graphs. Algorithmica **62**(3–4), 713–732 (2012)
13. Cohen, E., Halperin, E., Kaplan, E., Zwick, U.: Reachability and distance queries via 2-hop labels. SIAM J. Comput. **32**(5), 1338–1355 (2003)
14. de Montgolfier, F., Soto, M., Viennot, L.:. Treewidth and hyperbolicity of the internet. In: NCA, pp. 25–32 (2011)
15. Diestel, R.: Graph theory. Springer, Berlin (2005)
16. Dragan, F.F., Yan, C.: Collective tree spanners in graphs with bounded parameters. Algorithmica **57**(1), 22–43 (2010)
17. Fraigniaud, P.: Greedy routing in tree-decomposed graphs. In: ESA, pp. 791–802 (2005)
18. Gavoille, C., Peleg, D., Pérennes, S., Raz, R.: Distance labeling in graphs. J. Algorithms **53**(1), 85–112 (2004)
19. Grumbach, S., Wu, Z.: Distributed tree decomposition of graphs and applications to verification. In: Parallel and Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on, pp. 1–8. IEEE (2010)
20. Gubichev, A., Bedathur, S.J., Seufert, S., Weikum, G.: Fast and accurate estimation of shortest paths in large graphs. In: CIKM, pp. 499–508 (2010)
21. Harary, F., Uhlenbeck, G.: On the number of husimi trees: I. Proc. Natl. Acad. Sci. USA **39**(4), 315 (1953)
22. Harel, D., Tarjan, R.E.: Fast algorithms for finding nearest common ancestors. SIAM J. Comput. **13**(2), 338–355 (1984)
23. Jin, R., Ruan, N., Xiang, Y., Lee, V.E.: A highway-centric labeling approach for answering distance queries on large sparse graphs. In: SIGMOD Conference, pp. 445–456 (2012)
24. Jin, R., Ruan, N., Xiang, Y., Wang, H.: Path-tree: an efficient reachability indexing scheme for large directed graphs. ACM Trans. Database Syst. **36**(1), 7 (2011)
25. Jin, R., Xiang, Y., Ruan, N., Fuhry, D.: 3-hop: a high-compression indexing scheme for reachability query. In: SIGMOD Conference, pp. 813–826 (2009)
26. Leskovec, J., Kleinberg, J., Faloutsos, C.: Graphs over time: densification laws, shrinking diameters and possible explanations. In: Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining, pp. 177–187. ACM (2005)
27. Leskovec, J., Lang, K.J., Dasgupta, A., Mahoney, M.W.: Community structure in large networks: natural cluster sizes and the absence of large well-defined clusters. Internet Math. **6**(1), 29–123 (2009)
28. Malewicz, G., Austern, M.H., Bik, A.J., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: a system for large-scale graph processing. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, pp. 135–146. ACM (2010)
29. Peleg, D.: Proximity-preserving labeling schemes and their applications. In: WG, pp. 30–41 (1999)

30. Potamias, M., Bonchi, F., Castillo, C., Gionis, A.: Fast shortest path distance estimation in large networks. In: CIKM, pp. 867–876 (2009)
31. Robertson, N., Seymour, N.: Graph minors. III. planar tree-width. J. Comb. Theory Ser. B **36**(1), 49–64 (1984)
32. Schenkel, R., Theobald, A., Weikum, G.: Hopi: an efficient connection index for complex xml document collections. In: EDBT, pp. 237–255 (2004)
33. Thorup, M.: Compact oracles for reachability and approximate distances in planar digraphs. J. ACM **51**(6), 993–1024 (2004)
34. Tretyakov, K., Armas-Cervantes, A., García-Bañuelos, L., Vilo, J., Dumas, M.: Fast fully dynamic landmark-based estimation of shortest path distances in very large graphs. In: CIKM, pp. 1785–1794 (2011)
35. Trißl, S., Leser, U.: Fast and practical indexing and querying of very large graphs. In: SIGMOD Conference, pp. 845–856 (2007)
36. van Schaik, S.J., de Moor, O.: A memory efficient reachability data structure through bit vector compression. In: SIGMOD Conference, pp. 913–924 (2011)
37. Wang, H., He, H., Yang, J., Yu, P.S., Yu, J.X.: Dual labeling: answering graph reachability queries in constant time. In: ICDE, p. 75 (2006)
38. Wei, F.: Tedi: efficient shortest path query answering on graphs. In: SIGMOD Conference, pp. 99–110 (2010)
39. Xiang, Y., James, S.L., Borlawsky, T.B., Huang, K., Payne, P.R.: k-neighborhood decentralization: a comprehensive solution to index the UMLS for large scale knowledge discovery. J. Biomed. Inf. **45**(2), 323–336 (2012)
40. Yildirim, H., Chaoji, V., Zaki, M.J.: Grail: scalable reachability index for large graphs. PVLDB **3**(1), 276–284 (2010)
41. Yoo, A., Chow, E., Henderson, K., McLendon, W., Hendrickson, B., Catalyurek, U.: A scalable distributed parallel breadth-first search algorithm on bluegene/l. In: Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference, pp. 25–25. IEEE (2005)