Shortest-Path Queries for Complex Networks: Exploiting Low Tree-width Outside the Core

Takuya Akiba
The University of Tokyo
7-3-1 Hongo, Bunkyo-ku
Tokyo, 113-0033, Japan
t.akiba@is.s.u-tokyo.ac.jp

Christian Sommer
MIT CSAIL
32 Vassar Street
Cambridge, MA 02139
csom@csail.mit.edu

Ken-ichi Kawarabayashi
National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-ku
Tokyo, 101-8430, Japan
k_keniti@nii.ac.jp

ABSTRACT

We present new and improved methods for efficient shortest-path query processing. Our methods are tailored to work for two specific classes of graphs: graphs with small tree-width and complex networks. Seemingly unrelated at first glance, these two classes of graphs have some commonalities: complex networks are known to have a *core–fringe* structure with a dense core and a tree-like fringe.

Our main contributions are efficient algorithms and data structures on three different levels. First, we provide two new methods for graphs with small but not necessarily constant tree-width. Our methods achieve new tradeoffs between space and query time. Second, we present an improved tree-decomposition-based method for complex networks, utilizing the methods for graphs with small tree-width. Third, we extend our method to handle the highly inter-connected core with existing exact and approximate methods.

We evaluate our algorithms both analytically and experimentally. We prove that our algorithms for low-tree-width graphs achieve improved tradeoffs between space and query time. Our experiments on several real-world complex networks further confirm the efficiency of our methods: Both the exact and the hybrid method have faster preprocessing and query times than existing methods. The hybrid method in particular provides an improved tradeoff between space and accuracy.

Categories and Subject Descriptors

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

General Terms

Algorithms, Experimentation, Performance

Keywords

Graphs, complex networks, shortest paths, query processing

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EDBT 2012, March 26–30, 2012, Berlin, Germany.
Copyright 2012 ACM 978-1-4503-0790-1/12/03 ...$10.00

1. INTRODUCTION

Complex networks, scale-free networks, small-world graphs, and power-law graphs are just four of the popular names for the classes of networks modeling connections and interactions between entities of social, biological, technological, or information networks. Many of these networks are very large, consisting of millions or billions of nodes and edges, so that even some of the most fundamental operations one wishes to perform for such graphs are non-trivial to be executed.

One of the fundamental queries one might ask is a shortest-path (or shortest-path distance) query between two nodes. Such operations are used, for example, to support top-k keyword queries on graph-structured data [30, 52], to improve the search results in context-aware web search [53] and socially sensitive user search [57], to analyze influential people and communities in social networks [33, 4], to identify optimal pathways between compounds in metabolic networks [44, 45], and to manage resources in communication networks [42, 6]. Last but not least, shortest-path computations are of course ubiquitous in transportation networks [32, 5, 25].

A classical approach to a shortest-path query problem would be to conduct a breadth-first search (for unweighted graphs) or to run Dijkstra's algorithm (for weighted graphs). However, for large graphs, these algorithms are too slow, since they run in time linear proportional to the graph size and they are also difficult to parallelize. Moreover, in the applications mentioned above, shortest-path computations are often used in user interactions or as an important building block for other algorithms. To speed up these computations, we first precompute a *data structure* or an *index* and in a second phase we answer queries using this index. Two important performance indicators of a shortest-path query method are the space required for the index and the query time, and, in particular, the tradeoff between these two quantities. Other important factors are the preprocessing time and, if queries return approximately shortest paths, the approximation guarantee.

For structured networks such as road networks, shortest-path queries can be answered efficiently using small separators and related techniques. Complex networks, however, are referred to as "complex" since they have topological properties that are somewhat hard to grasp. In particular, they do not seem to have small separators or other structural properties that can be used to speed up algorithms. Research communities from fields as diverse as mathematics, physics, biology, computer science, or sociology, seem to

have agreed that many of these networks have a *dense core* of roughly linear size [12, 39, 40, 9, 22, 41], whose removal leaves a *tree-like fringe*. This is the main "structural" property of complex networks exploited by our method.

We design algorithms and data structures for two classes of graphs: *(i)* graphs with small *tree-width* and *(ii)* complex networks. The former class of graphs and its algorithmic properties have been investigated intensively by the theoretical computer science community. If a graph has small tree-width (see Section 3 for a definition) it means that it is somewhat tree-like, which makes it much more tractable for algorithmic questions than general graphs. For example, on graphs with bounded tree-width, many NP–hard problems can be solved efficiently [3]. There are also data structures that answer shortest-path queries more efficiently than those for general graphs [13, 20]. The second class of graphs — complex networks — has attracted vast practical interest in recent years. Complex networks seem to be quite common in the real world. Indeed, many of the applications mentioned above are supposed to work with complex networks.

At first glance, these two classes of graphs seem to be rather unrelated, however, the aforementioned core–fringe structure of complex networks is what brings them together. Researchers observed that the tree-like fringe can be dealt with efficiently in shortest-path-related computations. In this work, we extend and improve upon a recent and very promising method (see Section 2 for a more detailed account of related work). Wei [55] gave a method that constructs a *relaxed* tree decomposition of the graph using a prescribed *width* parameter w, such that all but one part (the core) of the decomposition tree have size proportional to w. Shortest-path queries are then handled by query engines for the core and for the fringe separately.

1.1 Contribution

In this paper, we first concentrate on methods for graphs with small tree-width. We propose two new methods that attain new and improved tradeoffs between space and query time: our first method uses $O(nw^2)$ space and answers queries in time $O(w^2 \log\log n)$; our second method uses $O(m)$ space and answers queries in time $O(w^5 \log^3 n)$, where n is the number of vertices, m is the number of edges and w is the tree-width.

We then discuss practical methods for complex networks. We present an improved exact method based on the method by Wei [55]. In particular, our method has a new preprocessing algorithm with better scalability. As query algorithms we can use faster algorithms based on the methods for graphs with small tree-width developed in the first part of this paper. Since, depending on the choice of the width parameter w, the core may remain quite large, we propose a new scheme composing the exact method for the fringe with existing methods for the core. In particular, we evaluate a hybrid method, consisting of a landmark-based approximation method for the core, showing better scaling performance than the exact methods.

Finally, we present the results of an experimental study on a number of real-world complex networks. The results show that *(i)* our new exact method works faster in both the preprocessing and the query phase, *(ii)* our hybrid method achieves better scalability than the exact method, and *(iii)* the hybrid method shows better accuracy and space consumption than existing landmark-based methods.

2. RELATED WORK

There is a large body of work on shortest-path query processing for *spatial* networks, in particular for road networks [18, 26, 23]. Although very efficient when applied to road networks, hierarchical techniques seem to have problems handling graphs with a large variety in node degrees. For complex networks, other techniques and heuristics have been suggested and evaluated.

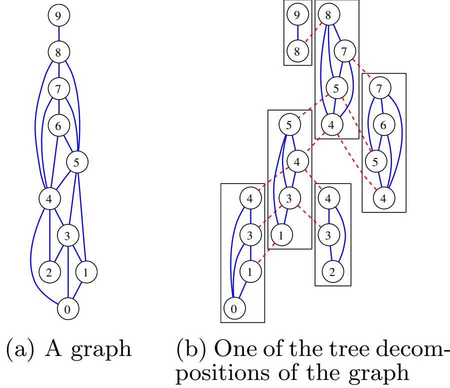
The core–fringe "structural" properties of complex networks have been successfully exploited to obtain efficient shortest-path query [55] and routing schemes [11,14]. Wei [55] explores the property that many scale-free networks have reasonably small *tree-width* outside the *core*. We cover that method in more detail in subsequent sections. In the routing scheme of Brady and Cowen [11], the algorithm first computes a shortest-path tree from the node with the highest degree. All nodes up to distance $d/2$ for some parameter d form the *core* with diameter d. The remaining nodes form the *fringe*, which is claimed to be almost a forest (up to some not-too-large set of edges E'). The scheme uses one routing tree for the core and $|E'|$ trees for the fringe – experiments using random power-law graphs [1] indicate that both d and E' can be chosen to be small simultaneously.

For *exact* shortest-path queries, a particularly promising approach is *2–hop cover* [16], where the main idea is to cover every shortest path with the concatenation of at most two paths stored in the index. See for example Cheng and Yu [15] for an efficient implementation. Many networks show some level of *symmetry*, which is exploited in [56] by treating vertices as orbits of automorphism groups. Furthermore, there are methods using an A*–variant of Dijkstra's algorithm to efficiently compute shortest paths [27]. Their performance highly depends on the selection of a set of *landmarks*, which are used to approximate distances [43] to eventually prioritize the edges at search time.

For *approximate* shortest-path queries, there are many powerful methods. In theory, these are known as approximate distance oracles [51]. Landmark-based methods [50,54] use triangulation to approximate distances, it is however not clear how many landmarks are necessary to achieve good approximation factors for complex networks [34]. As for A*, the selection of landmarks is very important [43]. Das Sarma et al. [17] provide a practical implementation of Bourgain's embedding [10], and they propose an extension of the distance oracle by Thorup and Zwick. In their extension, they omit ball computations. While the asymptotic performance is not affected, their algorithms both for preprocessing and query are simpler and potentially faster in practice than the corresponding original algorithms. The stretch bounds, however, only hold with high probability. Other embedding-based methods include [58]. Gubichev et al. [28] extend their work such that the method can also output shortest paths. The implementation can handle massive graphs with millions of nodes and edges. Rattigan et al. [46,47] approximate distances in graphs using a *structure index*. Their algorithm grows *zones* using random exploration starting from random seeds. Honiden et al. [31] provide an approximation method with very fast preprocessing times based on *Voronoi* duals.

3. PRELIMINARIES

Let G be a graph with vertex set $V(G)$ and edge set $E(G)$. Let the set of neighbors of vertex v in G be denoted by



(a) A graph (b) One of the tree decompositions of the graph

Figure 1: An example of a tree decomposition: Rectangles denote the bags; circles and blue lines denote the vertices and the edges of the original graph; red dashed lines connect the same vertices between adjacent bags.

$N_G(v)$. Let $d_G(u,v)$ denote the shortest-path distance between two nodes $u, v \in V(G)$.

3.1 Tree decompositions, width, and tree-width

The tree-width of a graph was introduced by Halin [29], but it went unnoticed until it was rediscovered by Robertson and Seymour [48], and, independently, by Arnborg and Proskurowski [3].

A *tree decomposition* of a graph G is a pair $(T, \mathscr{X})$, where T is a tree and $\mathscr{X} = \{X_t \mid t \in V(T)\}$ is a family of subsets of $V(G)$, with the following properties: *(i)* $\bigcup_{t \in V(T)} X_t = V(G)$. *(ii)* For every $(u,v) \in E(G)$, there exists $t \in V(T)$ such that $u, v \in X_t$. *(iii)* For all $v \in V(G)$, the set $\{t \mid v \in X_t\}$ induces a subtree of T.

In this paper, we call the sets X_t *bags*. For an example of a graph and one of its tree decompositions, refer to Figure 1.

The *width* of a tree decomposition $(T, \mathscr{X})$ is

$$\max_{t \in V(T)} \{|X_t| - 1\}.$$

A graph G has *tree-width* w if w is the minimum such that G has a tree decomposition of width w.

For example, the width of the tree decomposition illustrated in Figure 1b is 3. Since the graph has a tree decomposition of width 3 and since the cliques of size 4 in the graph (such as vertices 4, 5, 6, and 7) forbid the width of its tree decompositions to be less than 3, the graph shown in Figure 1a has tree-width exactly 3.

A graph has tree-width 1 if and only if it is a forest, and families of graphs with tree-width at most 2 include outerplanar graphs and series-parallel graphs. The tree-width is a good measure of the algorithmic tractability of graphs.

3.2 Distance Oracles and Labeling Schemes

A *distance oracle* is a data structure whose purpose is to serve as a replacement of the all-pairs shortest paths matrix of a graph. Given the distance oracle of a graph G, we wish to efficiently answer (approximate) distance queries $d_G(u,v)$. A *distance labeling scheme* is the distributed version of a distance oracle. More formally, a distance labeling scheme consists of two functions $\mathcal{L} : V \to \{0,1\}^\ell$ for some integer $\ell > 0$ called the *label length* and $\mathcal{D} : \{0,1\}^\ell \times \{0,1\}^\ell \to \mathbb{R}^+$ such that, for any two nodes $u, v \in V(G)$, we can compute the distance between u and v given their labels $\mathcal{L}(u), \mathcal{L}(v)$ only; more formally, we have $\mathcal{D}(\mathcal{L}(u), \mathcal{L}(v)) = d_G(u,v)$.

For most classes of graphs, distance labels are either long ($\ell = \Theta(n)$) or they provide approximations of $d_G(\cdot,\cdot)$ only. For graphs with tree-width $w(n)$, however, there exists an exact distance labeling scheme by Gavoille et al. [24], whose label length is rather short.

LEMMA 3.1 (GAVOILLE ET AL. [24, THEOREM 2.4]). *For a function $r(n)$, let $R(n) := \sum_{i=0}^{\log_{3/2} n} r(n(2/3)^i)$. For a class of graphs with a recursive $r(n)$–separator, there exists a distance labeling scheme with label length at most $O(\log^2 n + R(n)\log n)$. Using two labels, the distance between the two corresponding nodes can be computed in time $O(R(n)\log n)$.*

For positive non-decreasing functions $r(n)$, we have that $R(n) \le r(n)\log_{3/2} n$. As a corollary, we obtain:

LEMMA 3.2 (GAVOILLE ET AL. [24]). *For graphs with tree-width $w(n)$, there exists a distance labeling scheme with labels of length $O(w(n) \cdot \log^2 n)$ and query time $O(w(n)\log^2 n)$.*

4. QUERY PROCESSING FOR GRAPHS WITH SMALL TREE-WIDTH

In this section, we present space-efficient distance oracles for undirected graphs on n nodes with m edges and small (but not necessarily constant) tree-width $w(n)$. Our methods offer different tradeoffs than those of existing methods [13, 20].

4.1 Data Structure for Fast Queries

The first data structure requires space $O(w^2 n)$ and answers distance queries in time $O(w^2 \log\log n)$. Our implementation uses this first data structure.

4.1.1 Basic Method

We start with the following theorem.

THEOREM 4.1 (WEI [55]). *For any graph G on n nodes that has a tree decomposition with b bags, height h, and width w, there is a distance oracle using space $O(bw^2)$ with query time $O(hw^2)$.*

During preprocessing, we compute and save the shortest-path distance between any two vertices u, v contained together in some bag $u, v \in B_t$ (we call this the *local distance*). Since the number of vertex pairs is $O(w^2)$ per bag, the space is $O(bw^2)$.

Query answering is based on bottom-up dynamic programming. We define an arbitrary bag to be the root and we consider the tree as a rooted tree. Let x and y be the end-points of the shortest path to be computed. The following lemma is characteristic for tree decompositions; it says that shortest paths must in some sense "follow" the decomposition tree, which is essential to efficiently answer distance queries.

LEMMA 4.1. *Let G be a graph and $x, y \in V(G)$. Let $(T, \mathscr{X})$ be a rooted tree decomposition of G and $\mathscr{X} = \{X_t \mid$*

$t \in V(T)\}$. Let T_x and T_y be the subtrees of T induced by the bags including x and y, respectively. Let t_x and t_y be the vertices of T_x and T_y that is closest to the root of T, respectively. Let u be the lowest common ancestor of t_x and t_y. Any path from x to y must contain at least one vertex in the bag X_u.

Lemma 4.1 shows that the distance from x to y can be computed from the distances from x to all the vertices in the bag X_u and from all the vertices in the bag X_u to y by the following equation.

$$d_G(x, y) = \min_{v \in X_u} \{d_G(x, v) + d_G(v, y)\},$$

where $d_G(x, y)$ is the distance between x and y.

The query answering algorithm can be derived from this fact. The algorithm is based on dynamic programming. We climb the tree from t_x to u computing the distance from x to all the vertices in each bag and, analogously, climb from t_y to u computing the distance from y to all the vertices in each bag.

Let X_t be the current bag and let X_p be the parent bag of X_t. Suppose that $d_G(x, v)$ is known for the vertices $v \in X_t$. Then we can compute $d_G(x, z)$ for the vertex $z \in X_p$ by the following equation:

$$d_G(x, z) = \min_{v \in X_p \cap X_t} \{d_G(x, v) + d_G(v, z)\}. \quad (1)$$

Note that $d_G(v, z)$ in the formula above is precomputed since v and z are in the same bag X_p. The computation of the distance to vertex y is done analogously.

Computing $d_G(x, z)$ requires $O(w)$ time. Since we visit at most $2h$ bags (h bags from both t_x and t_y), d_x is computed for at most $2wh$ vertices. Hence the time complexity of processing queries is $O(w^2 h)$.

4.1.2 Query Answering in $O(w^2\sqrt{h})$ Time

In the following, we improve upon the basic method outlined in the previous section. We prove the following. Our approach is similar to [2].

THEOREM 4.2. *For any graph G on n nodes that has a tree decomposition with b bags, height h, and width w, there is a distance oracle using space $O(bw^2)$ with query time $O(w^2\sqrt{h})$.*

Let $(T, \mathscr{X})$ be the rooted tree decomposition of G. For every vertex $t \in V(T)$, let depth(t) be the depth of t. To achieve $O(w^2\sqrt{h})$ query time, we additionally precompute shortest paths from all the vertices in X_t to all the vertices in X_s for every vertex $t \in V(T)$ with depth larger than or equal to $\lfloor\sqrt{h}\rfloor$, where s is the $\lfloor\sqrt{h}\rfloor$-th ancestor of t. This additional index occupies space $O(bw^2)$, which dominates the total index size.

Queries are answered as follows: we first take *big steps* towards the least common ancestor, "jumping" up to the $\lfloor\sqrt{h}\rfloor$-th ancestor as many times as we can, then we further climb one by one using regular steps (which we call *small steps*).

Since the depth of t_x is at most h, the number of big steps is at most $h/\lfloor\sqrt{h}\rfloor = O(\sqrt{h})$. For the remainder, we use small steps, whose number is also bounded by $\lfloor\sqrt{h}\rfloor = O(\sqrt{h})$. Therefore, the total number of steps of dynamic programming is $O(\sqrt{h})$, which means that our new algorithm answer queries in $O(w^2\sqrt{h})$ time.

4.1.3 Query Answering in $O(w^2 \log h)$ Time

The height h of the tree could potentially still be large. We further reduce the dependency on h in the following.

THEOREM 4.3. *For any graph G on n nodes that has a tree decomposition with b bags, height h, and width w, there is a distance oracle using space $O(bw^2)$ with query time $O(w^2 \log h)$.*

To achieve $O(w^2 \log h)$ query time, we start with a method that precomputes shortest paths from all the vertices in X_t to all the vertices in its 2^k-th ancestor bag for each vertex $t \in V(T)$ and k, where $1 \le k \le \lfloor\log_2(\text{depth}(t))\rfloor$. Now we can jump to every 2^k-th ancestor from every bag. By repeatedly jumping to the highest ancestors that is not higher than the target bag, the number of steps is at most $\log_2 h$ (since we use each scale k for a 2^k-jump at most once). However, since we have to store $O(w^2 \log(\text{depth}(t)))$ shortest paths for every bag X_t, the size of the preprocessed data is $O(bw^2 \log h)$, which is undesirable.

We reduce the preprocessed data size as follows. We modify the method to use jumps from bag X_t to every 2^k-th ancestor where depth(t) is divisible by 2^k. We still do not jump more than twice using the same distance 2^k and we can answer queries in $O(w^2 \log h)$ time. If we assume that the depth of bags is distributed uniformly, then the number of allowed jumps is about $2b$ and the size of the preprocessed data is $O(bw^2)$. However, this assumption does not always hold.

To overcome problems with such decomposition trees, we propose to only use jumps of length 2^k if a vertex has at least 2^k children. This restriction allows us to bound the number of bags with precomputed data for jumps of length 2^k by $\lfloor b/2^k\rfloor$, obtaining overall space $O(bw^2)$. Note that we can still bound the number of jumps with length 2^k for $k > 0$ by two. Jumps of length 1 ($k = 0$) occur at most three times. Hence, the overall query time is $O(w^2 \log h)$.

4.1.4 Query Answering in $O(w^2 \log\log n)$ Time

It is known that we can transform a tree decomposition into another tree decomposition with height $O(\log n)$ without increasing the width by too much.

LEMMA 4.2 (FARZAN ET AL [20, LEMMA 1]). *Given a tree decomposition with width w for a graph with n vertices, one can obtain, in linear time, a tree decomposition for the graph with width at most $3w + 2$ and height $O(\log n)$.*

Applying the previous theorem to the tree decompositions obtained by this lemma, we obtain the following theorem.

THEOREM 4.4. *For any graph G on n nodes with tree-width w there is a distance oracle using space $O(nw^2)$ with query time $O(w^2 \log\log n)$.*

4.2 Linear-space Data Structure

In the following, we devise a distance oracle whose space requirements do *not* depend on the tree-width w. Obviously, there is some dependency on the tree-width: the query times of our data structures depend on w. We derive a very simple distance oracle that makes use of a distance labeling scheme by Gavoille et al. [24] (see Lemma 3.2 in the preliminaries).

We prove the following theorem.

THEOREM 4.5. *For any graph G on n nodes with tree-width $w(n)$ there is a distance oracle using space $O(n)$ with query time $O(w^3(n) \cdot \log^2 n \cdot \log w(n) \cdot \log\log n + w^5(n) \cdot \log^2 n)$.*

For convenience, we work with the *branch decomposition* instead of the tree decomposition. It is known that the *branch-width* is at most 1.5 times larger than the tree-width of a graph [49, Theorem 5.1]. For general graphs, there is a polynomial-time approximation algorithm for branch-width (constant-factor approximation for minor-free graphs) [21].

All the nodes of the branch decomposition tree have degree 1 or 3, making it rather convenient to work with. The following lemma implies that we can partition a branch decomposition tree into subtrees $T_1, T_2, \ldots$ of roughly equal size.

LEMMA 4.3 (COROLLARY OF [19, LEMMA 12.4.6]). *Let $k \geq 2$ be an integer. Let T be a tree of maximum degree ≤ 3. Then T has a set F of edges such that every component of $T - F$ has between k and $2k - 1$ vertices, except that one such component may have fewer vertices.*

We apply Lemma 4.3 for $k = \lceil w^2(n) \cdot \log^2 n \rceil$. Every component T_i corresponds to a subgraph G_i on at most $O(w^3(n) \cdot \log^2 n)$ nodes and edges. For each graph edge $e = (u, v)$ whose end points u and v are in only one tree T_i (more precisely, the nodes u and v are contained only in bags that correspond to nodes of one subtree T_i), we store the label i with the edge e (which requires $O(\log n)$ *bits* per edge — the overall space requirement remains a linear number of *words*).

Recall that for a graph with branch-width w, each bag of the branch decomposition obtained by Lemma 4.3 contains $O(w)$ nodes. We wish to augment the graph as follows: for each bag corresponding to the root node of a subtree T_i (T_i is a component obtained by Lemma 4.3) we store the exact distance labels for all its $O(w)$ nodes (Lemma 3.1 and its corollary).

LEMMA 4.4. *The total space required to store all the labels is at most $O(n)$* words.

PROOF. The branch decomposition tree T has at most $O(n)$ nodes. There are at most $O(n/(w(n)^2 \log^2 n))$ subtrees T_i. Each root bag consists of at most $O(w(n))$ nodes. Each label has length $O(w(n) \cdot \log^2 n)$ (Lemma 3.2). □

At query time, the distance between a pair of nodes (s, t) is computed as follows. We assume that neither the distance label of s nor the distance label of t are known (the case in which one or both labels are known is a special case that can be solved in a straightforward way). We describe the first part of the query algorithm for s. Since s is not labeled, it is contained in only one subgraph G_i. We compute a single-source shortest-path tree in G_i using Dijkstra's algorithm as follows: whenever a labeled node s' is reached, its neighbors are *not* inserted into the queue (note that the search tree explores the $O(w^3(n) \cdot \log^2 n)$ nodes of G_i and the root bags of its children). The same procedure is done for t, computing the distance to its labeled separator nodes t'. We then compute the distance between all possible pairs of cut nodes (s', t'). The distance $d_G(s, t)$ is either equal to $d_{G_i}(s, t)$ (if s and t are in the same subgraph G_i and the shortest path does not leave G_i) or it satisfies

$$d_G(s,t) = \min_{s' \in G_i, t' \in G_j} d_{G_i}(s, s') + d_G(s', t') + d_{G_j}(t', t)$$
$$= \min_{s' \in G_i, t' \in G_j} d_{G_i}(s, s') + \mathcal{D}(\mathcal{L}(s'), \mathcal{L}(t')) + d_{G_j}(t', t).$$

If $T_i = T_j$ then each separator bag can be considered individually. There are $O(w^2)$ separator bags with $O(w^2)$ pairs of nodes each, yielding $O(w^4)$ pairs to check. If $T_i \neq T_j$ then for one tree, say T_i wlog, the distance from the root is at least as large as the distance for the other tree T_j. Therefore, any shortest path must go through one of the nodes in its root bag B_i. There are $O(w)$ nodes in B_i and $O(w^3)$ labeled nodes in T_j, yielding $O(w^4)$ pairs to check.[1] Computing one distance can be done in time $O(w \log^2 n)$ (Lemma 3.1).

NOTE.

If s and t are in different subgraphs $G_i \neq G_j$ then there is exactly one bag (with $O(w)$ nodes) in G_i that separates s from t. Let B_i denote this bag. The same holds for t and its subgraph G_j (let B_j denote this bag). The distance $d_G(s, t)$ satisfies

$$d_G(s,t) = \min_{s' \in B_i, t' \in B_j} d_{G_i}(s, s') + d_G(s', t') + d_{G_j}(t', t)$$
$$= \min_{s' \in B_i, t' \in B_j} d_{G_i}(s, s') + \mathcal{D}(\mathcal{L}(s'), \mathcal{L}(t')) + d_{G_j}(t', t).$$

5. APPLICATION TO COMPLEX NETWORKS

Wei proposed a method to answer shortest-path queries based on tree decompositions [55]. We build on this method.

The outline of the method is the following: the preprocessing algorithm computes a tree decomposition of the graph in a heuristic way and it computes distance matrices for each bag. At query time, we can use algorithms based on those discussed in Section 4.1.

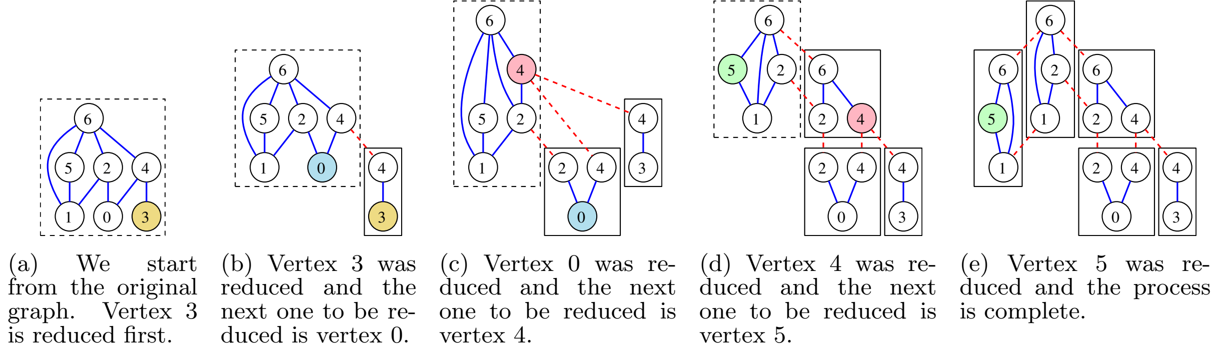
Our method depends on the core-fringe structure, which is a property that many complex networks have. As a consequence, while our method works well on complex networks like social networks or web graphs, it is not competitive with existing methods tailored for other networks like road networks, expander graphs, or Erdős-Rényi random graphs.

5.1 Relaxed Tree Decomposition

Since complex networks are believed to *not* have small separators, their tree-widths can be very large. Instead of computing a *strict* tree decomposition, we compute a *relaxed* tree decomposition, wherein all bags *except for one* (the root bag) have bounded size. The preprocessing algorithm is parameterized by an integer w, which restricts the size of each bag (except for the root bag) to $w + 1$ vertices.

On a high level, the algorithm repeatedly *reduces* a vertex with degree at most w, generating a list of bags that form a tree (details listed in Algorithm 1). For an illustration of this process, we refer to Figure 2. Note that, actually, parent relationships among bags are determined only after all the bags have been generated.

[1]Note that there is actually only one bag B_j that needs to be checked — B_j could be computed using a level ancestor query



(a) We start from the original graph. Vertex 3 is reduced first.

(b) Vertex 3 was reduced and the next one to be reduced is vertex 0.

(c) Vertex 0 was reduced and the next one to be reduced is vertex 4.

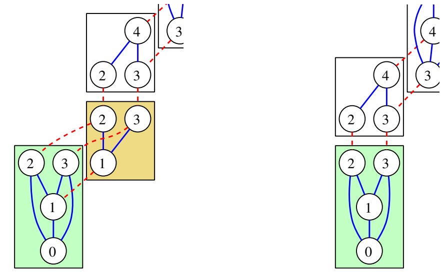
(d) Vertex 4 was reduced and the next one to be reduced is vertex 5.

(e) Vertex 5 was reduced and the process is complete.

Figure 2: Illustration of the computation of a relaxed tree decomposition.

Algorithm 1 Compute a relaxed tree decomposition

1: **procedure** DECOMPOSEGRAPH(G, w)
2: $\quad X \leftarrow$ empty list
3: $\quad$ **for** $d = 0$ to w **do**
4: $\quad\quad$ **while** $\exists v \in V(G)$ such that $deg(v) \leq d$ **do**
5: $\quad\quad\quad \triangledown$ generate a new bag B_v
6: $\quad\quad\quad G, B_v \leftarrow$ REDUCEVERTEX(G, v)
7: $\quad\quad\quad$ append B_v to list X
8: $\quad\quad$ **end while**
9: $\quad$ **end for**
10: $\quad \triangledown$ construct root bag with remaining vertices
11: $\quad R \leftarrow V(G)$
12: $\quad$ append R to list X
13: $\quad T \leftarrow$ CONSTRUCTTREE(X)
14: $\quad \mathscr{X} \leftarrow \{X_i \mid i = 1, 2, \cdots, |X|\}$
15: $\quad$ **return** $(T, \mathscr{X})$
16: **end procedure**



(a) Part of a tree decomposition with an unnecessary bag (the yellow bag including vertex 1, 2, and 3, which is a subset of the green bag below).

(b) Part of a tree decomposition without the unnecessary bag.

Figure 3: An example of removal of unnecessary bags.

The reduction of a vertex v (method REDUCEVERTEX) consists of three steps. First, we create a new bag B including v and all its neighbors (note that, to ensure that $|B| \leq w + 1$, only nodes v with degree $\deg(v) \leq w$ can be reduced). Second, we change the graph G by removing the nodes u whose neighborhood is completely contained in B, i.e., $N_G(u) \subseteq B$ (note that v is always removed from G). Third, we add a clique among those vertices in B that are still in G (to ensure that we can eventually compile a valid tree decomposition).

Note that this reduction process is different from the original process in [55] as follows. Our reduction process prevents the creation of redundant bags (see Figure 3) by removing not only vertex v but all vertices u with neighborhood $N_G(u)$ completely in B.

After reducing all the vertices with degree less than or equal to w, we create a bag with all the remaining vertices, which can be very large. Then we construct the tree of the tree decomposition from the list of bags. We can always obtain a valid tree decomposition since all the neighbors of a reduced node are connected by a clique.

As the bags generated by a node reduction have size at most $w+1$, all the bags other than the last bag have size at most $w + 1$. Therefore, the tree decomposition has *relaxed* width w. We call the last bag the *root bag*, and we consider the tree of the tree decomposition as rooted at this root bag.

Assuming adjacency lists are managed in hash tables and operations on edges can be done in $O(1)$ time, each reduction takes $O(w^2)$ time. We can find the parent of every bag in $O(bw)$ time. In total, we can compute relaxed tree decompositions in $O(n + bw^2)$ time. In our experiments, we found that computing the decomposition tree is much faster since a large fraction of the reduced vertices has very low degrees (most networks we experiment with have many vertices with low degree).

5.2 Exact Distance Queries

As we described before, to answer distance queries with algorithms discussed in Section 4.1, in addition to a tree decomposition we have to precompute distance matrices for each bag (*local distance*). One simple way originally used is to compute them on the original graph after we get the tree decomposition. We propose a new and more efficient algorithm.

5.2.1 Tree Decomposition and Local Distances

During the computation of a tree decomposition described in Section 5.1, many edges are removed from and added to the graph. Therefore, the number of the edges varies. In our experiments on real-world complex networks, during the computation of a tree decomposition usually the number of

edges decreases at first, and then it begins to increase.

The number of the edges decreases initially, and the number of edges dominates the running time of the shortest path computation. These facts motivate us to compute shortest paths on the graphs with the decreased number of edges. We store the graph with the minimum number of edges at this time and compute local shortest paths on it.

The above change of the algorithm requires us to treat the graph as weighted even if the original graph is unweighted. For example, suppose that we add an edge (y, z) to reduce vertex v. If we treat the graph as unweighted, then the distance between y and z becomes 1 and we cannot compute correct local shortest paths on the reduced graph. Therefore we must manage the weight of edges. Let $c(y, z)$ denote the weight of edge (y, z). When we add an edge (y, z) to reduce vertex v, we set $c(y, z)$ as $c(y, v) + c(v, z)$.

When a vertex is being deleted, we compute distances from this vertex to all the other vertices in the bag. At that point we do not care about other pairs in the bag because we make the remaining vertices a clique and they will be in the same bag when one of these vertices is deleted.

5.2.2 Storing Graphs

We want to store the graph with the minimum number of edges. However, repeatedly stroing copies of the entires graph is expensive. One way to avoid these copies is to represent graphs using persistent data structures.

We propose a simpler method. Since the bottleneck is the computation of local shortest paths, We can afford to once compute the tree decomposition without local shortest paths in order to evaluate when the number of edges is minimized. After this stage, we run the actual preprocessing algorithm. First, we compute shortest paths on the current reduced graph, then we copy the graph when the number of edges is minimum, and after that we compute shortest paths on the stored graph.

5.2.3 Computing Distance Matrices

Now that the graph is weighted, we can no longer use breadth-first search. One simple way is to use algorithms such as Dijkstra's algorithm with a priority queue like a binary heap instead.

However, there is a faster way using normal queues. Since weights are integral and the diameter of the graph is bounded by the number of vertices (and often very small because of the small-world property of complex networks), we can prepare different queues for vertices with different distance from the source vertex. We can conduct Dijkstra's algorithm using these queues as one priority queue, without the cost of heap operations.

We also found it is important for efficiency to stop Dijkstra's algorithm when all the distances to the target vertices are computed even if the queue is not empty at that time. The idea to stop Dijkstra's algorithm is quite simple and natural, but it significantly improves the performance. This is due to the fact that vertices in the same bag are often located near each other in the original graph.

Let m be the number of edges and let n be the number of vertices. Using the implementation specified above, Dijkstra's algorithm runs in $O(m)$ time and therefore the time complexity for all the distance computations is $O(nm)$. In total, adding $O(n + bw^2)$ time for graph reduction and $O(bw)$ time for computing parents of every bag, the time complexity of the preprocessing algorithm is $O(nm + bw^2)$, and the distance computation is the bottleneck. In practice, Dijkstra's algorithm rarely takes $O(m)$ time and runs much faster, and the total precomputation time is also much faster than the worst-case time complexity.

5.2.4 Adapting Query Algorithms to Relaxed Tree Decompositions

The tree decomposition has *relaxed* width w. The size of the root bag can be much larger than w. Let the root bag be R. The algorithms discussed in Section 4.1 require $O(|R|^2)$ time when the lowest common ancestor is the root bag, which can be particularly slow whenever $|R|$ is really large.

However, we can modify the algorithms to eliminate $|R|$ from the time complexity with a bit of careful case analysis. The key point is to utilize the fact that the large bag R lies only at the root of the tree. If the lowest common ancestor is not the root bag, then it does not matter. If the lowest common ancestor is the root bag, then we consider the following three cases:

1. **Both x and y are in R.** This case is the easiest one. The answer $d_G(x, y)$ is in the precomputed data.

2. **Either x or y is in R.** Without loss of generality, we suppose $x \notin R$ and $y \in R$. Let X be the bag that is the direct child of the root on the simple path from the starting bag including x. Computing the distance from x to all the vertices in R using equation 1 takes $O(w\,|R|)$ time, which should be avoided. Because every path from x to y passes through a vertex in $X \cap R$, we can directly compute the distance using the following equation from the distance from x to the vertices in X in $O(w)$ time.
$$d_G(x, y) = \min_{v \in X \cap R} \{d_G(x, v) + d_G(v, y)\}.$$
Note that $d_G(x, v)$ is the distance from x to the vertices in X, which can be computed in the normal way, and $d_G(v, y)$ is in the precomputed data because both are in R.

3. **Neither x nor y are in R.** This case can be also processed by using a similar idea for the previous case. Let X_1 and X_2 be the two bags that are direct children of the root on the simple paths from the bags including x or y, respectively. We can compute the distance using the following equation in $O(w^2)$ time.
$$d_G(x, y) = \min_{u \in X_1 \cap R, v \in X_2 \cap R} \{d_G(x, u) + d_G(u, v) + d_G(v, y)\}.$$

The time complexity becomes the same as those for graphs with tree-width w. For example, if we use the first basic query answering algorithm then we can answer queries in $O(w^2 h)$ time, where w is not the tree-width but the parameter we supply to the heuristic tree-decomposition algorithm. Although we managed to eliminate $|R|$ from the query time, the size of R has a high impact on the preprocessing time and, in particular, on the space requirements, since the index contains shortest-path distances among $O(|R|^2)$ pairs of vertices. Hence there is a trade-off between query time and the preprocessed data size. If we set w smaller, $|R|$ gets larger and the preprocessed data size also gets larger. If we

set w larger, h also gets larger and the query time may be longer.

5.3 Hybrid with Approximate Methods

5.3.1 General Framework

In the exact method discussed in Section 5.2, we precompute and store the distances among $O(|R|^2)$ pairs of vertices, where R is the root bag. When processing larger complex networks, this can be a severe bottleneck, because larger complex networks tend to have larger cores and it may be impossible to reduce the size of the root bag even for large w. To cope with that, we devise hybrid methods combining existing methods.

We may use any distance querying method. Whenever we require a value at the distance matrix for the root bag, we ask a query instead. For example, if we use the basic query answering algorithm based on the algorithm discussed in Section 4.1.1, we can answer queries in $O(w^2h + w^2Q)$ time, where Q is the query time of the method applied to the root bag.

5.3.2 Hybrid with Landmark-Based Methods

We elaborate on a hybrid method with the approximate method based on landmarks [50, 34, 54, 43], mainly for the following two reasons:

1. despite being very simple and elegant, the method works very well on large complex networks, and
2. we can specialize the query algorithm of the method to answer the original queries more efficiently than the general framework specified above.

Combining the landmark-based method with our method, we can improve the accuracy and space efficiency of the method as described in the experimental results (see Section 6.3).

Landmark-based methods work as follows. Given a graph G, first we select a set of vertices D as landmarks, then we precompute the distances from each landmark $u \in D$ to every vertex. To answer queries, we utilize one of the triangle inequalities

$$d_G(s,t) \leq d_G(s,u) + d_G(u,t),$$

where s, t are vertices, u is a landmark, and $d_G(s,t)$ is the distance between s and t. We compute the estimated distance $\tilde{d}_G(s,t)$ using the upper bound

$$\tilde{d}_G(s,t) = \min_{u \in D}\{d_G(s,u) + d_G(u,t)\},$$

where $d_G(s,u)$ and $d_G(u,t)$ are precomputed distances.

The strategy of selecting landmarks is important for the precision. Theoretical bounds were proven for random landmark selection [34] and several strategies, which are significantly better than the random selection, are discussed in [43]. One of the strategies, called *Degree*, chooses the vertices with the highest degrees as landmarks. Despite being simple and easy to implement, this selection strategy is competitive with the best methods known. We choose the vertices according to their degrees in the original graph before the reduction, because the degree changes by reductions and we may miss the important vertices if we use the degree after all the reductions.

As stated above, we can specialize the query algorithm of the landmark-based methods for this hybrid method. We can formulate the problem to be solved in the root bag as the following: given two vertices x and y and two sets of vertices S and T, let $d_G(x,s)$ for all $s \in S$ and $d_G(t,y)$ for all $t \in T$ be already computed, compute the best pair $(s,t) \in S \times T$ minimizing $d_G(x,s)+d_G(s,t)+d_G(t,y)$. x and y correspond to the endpoints of the original query, S and T correspond to the vertices in the two bags that are direct children of the root on the simple paths from the bags containing x or y.

If we answer this problem by querying $|S| \cdot |T|$ pairs the time complexity for the root bag is $O(|S|\cdot|T|\cdot|D|)$. However, we can answer queries in $O\left((|S| + |T|)\,|D|\right)$ time by using the equation

$$\begin{aligned}
&\min_{s\in S, t\in T}\left\{d_G(x,s) + \tilde{d}_G(s,t) + d_G(t,y)\right\} \\
&= \min_{s\in S, t\in T}\left\{d_G(x,s) + \min_{v\in D}\{d_G(s,v) + d_G(v,t)\}\right. \\
&\qquad \left. + d_G(t,y)\right\} \\
&= \min_{v\in D}\left\{\min_{s\in S}\{d_G(x,s) + d_G(s,v)\}\right. \\
&\qquad \left. + \min_{t\in T}\{d_G(v,t) + d_G(t,y)\}\right\}.
\end{aligned}$$

In total, because the size of S and T is at most w, we can answer the original queries in $O(w^2h + w\,|D|)$ time if we use the first query answering algorithm, and in $O(w^2\sqrt{h} + w\,|D|)$ or $O(w^2 \log h + w\,|D|)$ time if we use the improved algorithms.

6. EXPERIMENTAL EVALUATION

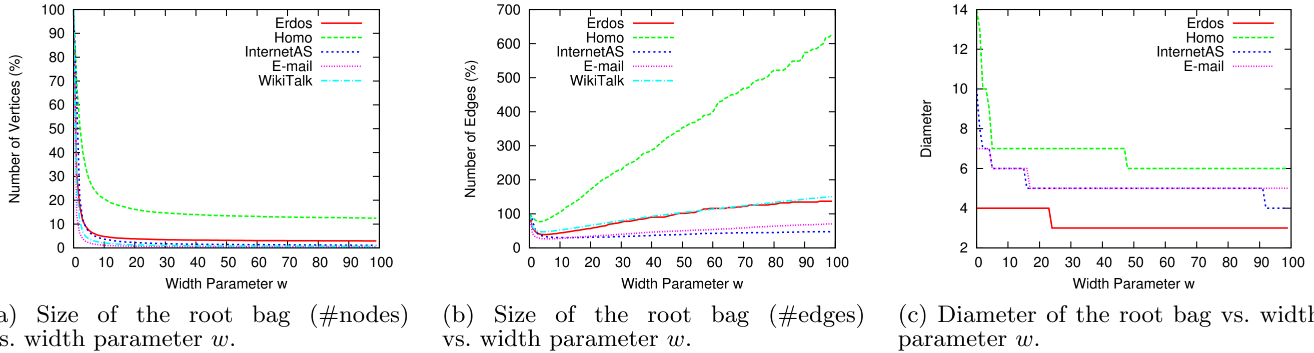
We implemented our methods in C++ using STL and the Boost C++ Libraries. The experiments were conducted on a Linux server with Intel Xeon X5670 (2.93 GHz) and 24GB of main memory. All graphs are treated as undirected unweighted graphs. Since the running time of a shortest-path query is often dominated by the time required to output the actual path, we focus on shortest-path *distance* queries (it is standard in this line of work to compare distance query times).

We conducted experiments on the real-world networks specified in Table 1. Erdős [56], E-mail [37], WikiTalk [35], Flickr [38] and LiveJournal [4] are social networks. InternetAS [56] and Skitter [36] are computer networks. Homo [56] is a biological network. IndianWeb [8, 7] is a web graph.

We used the five relatively smaller datasets with up to millions of vertices and edges (Erdős, Homo, InternetAS, E-mail and WikiTalk) for the preliminary experiments of tree decomposition and for an evaluation of the exact method. We used the other four larger datasets with millions of vertices and tens of millions of edges (Skitter, IndianWeb, Flickr and LiveJournal) for an evaluation of the hybrid approximation method, because it has better scalability than the exact method.

6.1 Tree Decomposition

We evaluate the transition of the number of vertices, number of edges and diameter of the root bags (R) against the *relaxed* tree-width (w).



(a) Size of the root bag (#nodes) vs. width parameter w.

(b) Size of the root bag (#edges) vs. width parameter w.

(c) Diameter of the root bag vs. width parameter w.

Figure 4: Statistics of the root bags against width parameter w.

Table 1: Datasets.

Dataset	Vertices	Edges	Type
Erdős	6,927	11,850	Social network
Homo	7,020	19,811	Biological network
InternetAS	22,442	45,550	Computer network
E-mail	265,214	365,025	Social network
WikiTalk	2,394,385	4,659,565	Social network
Skitter	1,696,415	11,095,298	Computer network
IndianWeb	1,382,908	16,917,053	Web graph
Flickr	1,846,198	22,613,981	Social network
LiveJournal	4,847,571	68,993,773	Social network

Number of Vertices: We start with evaluating the number of the vertices in the root bags against w (Figure 4a). At first, $|R|$ decreases quickly because there are many vertices with low degree. However, after that, $|R|$ stops decreasing so rapidly and it continues to decrease very slowly.

Number of Edges: Next, we evaluate the number of edges in the root bags (Figure 4b). At first, the number of edges also decreases but next it starts to increase and it continues increasing. It is because when we reduce a vertex with degree d we add up to d^2 edges, therefore if we reduce vertices with lower degree then we do not add many edges but when we reduce those with higher degree then several edges are added.

Diameter: Finally, we evaluate the diameter of the root bag with respect to the width parameter w (Figure 4c). The diameter is of particular interest for our hybrid method, since the error of many approximation methods depends directly on the diameter. We can confirm that the diameter decreases with the reduction. This can be good for the hybrid framework with approximation methods, because the error is bounded by the diameter of the root bag.

6.2 Exact Method

We implemented the exact distance query answering method with our new preprocessing algorithm described in Section 5 and our new $O(w^2\sqrt{h})$ time query answering algorithm described in Section 4.1.2. We compare the result with two existing exact methods: the previous tree-decomposition-based method (TEDI) [55] and the method exploiting symmetry (SYMM) [56]. The experiments of TEDI were also conducted on the same environment specified above using the implementation of the author of the original paper. The result of SYMM was taken from the paper [56] and therefore the comparison between SYMM and other methods is not completely fair because of the difference of environments. SYMM was implemented in C++ and the experiments were conducted on a Windows server with an Intel Pentium 2.0GHz CPU and 2GB of main memory.

Statistics of the tree decompositions that were computed in the preprocessing are shown in Table 2. We chose the value of w empirically to achieve a good tradeoff between the preprocessing time and the size of the preprocessed data. We used the same value of w for TEDI.

Preprocessing Time: First, we compare the preprocessing times (Table 3). Our algorithm works faster than previous methods, especially for larger graphs. Our method preprocess the WikiTalk dataset in about 2,500 seconds, while the previous tree decomposition based method took about 60,000 seconds.

Preprocessed Data Size: Next, we compare the sizes of preprocessed data (Table 4). Because we improved the vertex reduction rule to decrease the number of bags, the size may be smaller than the original method. On the other hand, because we store additional information for the improved query answering algorithm, the size may also be a little larger than the original method.

Query Time: Finally, we compare the average query answering times for random pairs of vertices (Table 5). The number of pairs we used was 10,000 for our method and TEDI and 1,000 for BFS. It shows that the query time is improved not only theoretically but also empirically.

6.3 Hybrid Approximate Method

We implemented the hybrid method with the landmark based method [43] as described in Section 5.3.2 with $O(w^2h+|D|w)$ time query answering algorithm where D is the set of landmarks. For the width parameter w, we chose $w = 20$ based on our experiments (Figure 4), which indicate that, for all the networks we consider, the root bag is substantially smaller than the initial graph at $w = 20$. We used three hundred landmarks and the degree strategy for landmark selection. The statistics of the tree decompositions that were computed in the preprocess are shown in Table 6.

Preprocessed Data Size: First, we compare the numbers of pairs whose distance was stored in the preprocessed data and the sizes of the preprocessed data (Table 7). The number of the stored distance values significantly decreases to about 10% to 30% when we use the hybrid method, show-

ing the potential of this hybrid method. The total size of the preprocessed data also decreases, but, it decreases to at most about half. It is because the amount of the information other than distance increases.

Preprocessing Time and Query Time: Next, we compare the preprocessing times and the average query times for 1,000 random pairs of vertices (Table 8). Compared to a standard landmark-based method, both preprocessing and query times increase, but they remain manageable.

Accuracy: Finally, we compare the accuracy. First, we compare the average relative errors for 1,000 random pairs of vertices (Table 10). We calculate the relative error dividing the absolute error by the actual distance. The average accuracy does not improve.

Second, we compare the average relative error for 1,000 random pairs of vertices with small distance of one, two, three, and four (Table 11). We observe that the accuracy improves. When using the hybrid method for close pairs, the paths may not pass through the root bag and we can compute the distance exactly. For some applications like social search, queries about close pairs of vertices can be more important than those for far pairs and this improvement can be helpful.

Compared to the original method, our hybrid method is more accurate and uses much less space, while maintaining the fast query time performance. Overall, the hybrid method provides an improved tradeoff between index size (space) and query time.

7. REFERENCES

[1] W. Aiello, F. R. K. Chung, and L. Lu. A random graph model for massive graphs. In *STOC*, 2000.

[2] N. Alon and B. Schieber. Optimal preprocessing for answering on-line product queries. Technical Report 71/87, Tel Aviv University, 1987.

[3] S. Arnborg and A. Proskurowski. Linear time algorithms for NP-hard problems restricted to partial k-trees. *Discrete Applied Mathematics*, 23(1):11–24, 1989.

[4] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan. Group formation in large social networks: membership, growth, and evolution. In *KDD*, 2006.

[5] H. Bast, S. Funke, P. Sanders, and D. Schultes. Fast routing in road networks with transit nodes. *Science*, 316(5824):566, 2007.

[6] S. Boccaletti, V. Latora, Y. Moreno, M. Chavez, and D. Hwang. Complex networks: Structure and dynamics. *Physics reports*, 424(4-5):175–308, 2006.

[7] P. Boldi, M. Rosa, M. Santini, and S. Vigna. Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks. In *WWW*, 2011.

[8] P. Boldi and S. Vigna. The webgraph framework I: compression techniques. In *WWW*, 2004.

[9] B. Bollobás and O. Riordan. Robustness and vulnerability of scale-free random graphs. *Internet Mathematics*, 1, 2003.

[10] J. Bourgain. On lipschitz embedding of finite metric spaces in Hilbert space. *Israel Journal of Mathematics*, 52(1-2):46–52, 1985.

[11] A. Brady and L. Cowen. Compact routing on power law graphs with additive stretch. In *ALENEX*, 2006.

[12] D. S. Callaway, M. E. J. Newman, S. H. Strogatz, and D. J. Watts. Network robustness and fragility: Percolation on random graphs. *Physical Review Letters*, 85:5468–5471, 2000.

[13] S. Chaudhuri and C. Zaroliagis. Shortest paths in digraphs of small treewidth. Part I: Sequential algorithms. *Algorithmica*, 27(3):212–226, 2000.

[14] W. Chen, C. Sommer, S.-H. Teng, and Y. Wang. Compact routing in power-law graphs. In *DISC*, 2009.

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

[16] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. In *SODA*, 2002.

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

[18] D. Delling, P. Sanders, D. Schultes, and D. Wagner. Engineering route planning algorithms. In *Algorithmics of Large and Complex Networks - Design, Analysis, and Simulation*, pages 117–139, 2009.

[19] R. Diestel. *Graph Theory*. Springer, August 2005.

[20] A. Farzan and S. Kamali. Compact navigation and distance oracles for graphs with small treewidth. In *ICALP*, 2011.

[21] U. Feige, M. T. Hajiaghayi, and J. R. Lee. Improved approximation algorithms for minimum weight vertex separators. *SIAM J. Comput.*, 38(2):629–657, 2008.

[22] A. D. Flaxman, A. M. Frieze, and J. Vera. Adversarial deletion in a scale free random graph process. In *SODA*, 2005.

[23] L. Fu, D.-H. Sun, and L. R. Rilett. Heuristic shortest path algorithms for transportation applications: State of the art. *Computers & Operations Research*, 33(11):3324–3343, 2006.

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

[25] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *WEA*, 2008.

[26] A. V. Goldberg. Point-to-point shortest path algorithms with preprocessing. In *SOFSEM*, 2007.

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

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

[29] R. Halin. S-functions for graphs. *Journal of Geometry*, 8(1-2):171–186, 1976.

[30] H. He, H. Wang, J. Yang, and P. S. Yu. Blinks: ranked keyword searches on graphs. In *SIGMOD*, 2007.

[31] S. Honiden, M. E. Houle, C. Sommer, and M. Wolff. Approximate shortest path queries in graphs using Voronoi duals. *Transactions on Computational Science*, 9:28–53, 2010.

[32] N. Jing, Y.-W. Huang, and E. A. Rundensteiner. Hierarchical optimization of optimal path finding for transportation applications. In *CIKM*, 1996.

[33] D. Kempe, J. Kleinberg, and E. Tardos. Maximizing

Table 2: Statistics of the tree decompositions for the exact method.

Dataset	w	$\|R\|$	$\|V(T)\|$	h
Erdős	7	446	6,479	6
Homo	15	1,276	5,691	8
InternetAS	13	732	21,682	9
E-mail	40	1,445	248,103	10
WikiTalk	100	16,457	2,375,001	9

Table 3: Preprocessing times of the exact methods (sec). Our preprocessing algorithm scales better.

Dataset	Ours	TEDI [55]	SYMM [56]
Erdős	0.2	0.6	90.5
Homo	3.0	2.8	54.0
InternetAS	1.3	5.4	1,709.6
E-mail	9.0	237.6	-
WikiTalk	2,473.4	57,072.1	-

Table 4: Preprocessed data sizes of the exact methods (MB). Our space requirements scale better.

Dataset	Ours	TEDI [55]	SYMM [56]
Erdős	0.7	0.5	32.3
Homo	1.8	6.9	32.6
InternetAS	3.0	1.7	744.1
E-mail	25.7	58.0	-
WikiTalk	416.1	3647.2	-

Table 5: Query times of the exact methods (μs). Our query times are consistently faster, despite smaller index sizes.

Dataset	Ours	TEDI [55]	BFS
Erdős	0.48	0.68	2.1×10^2
Homo	0.63	1.80	2.9×10^2
InternetAS	0.81	1.43	7.2×10^2
E-mail	0.46	1.45	1.0×10^4
WikiTalk	0.79	2.48	2.3×10^5

Table 6: Statistics of the tree decompositions for the hybrid method.

Dataset	$\|R\|$	$\|V(T)\|$	h
Skitter	256,665	1,414,950	21
IndianWeb	228,292	1,035,324	204
Flickr	190,624	1,608,268	17
LiveJournal	1,452,228	3,314,977	24

Table 8: Preprocessing times of the hybrid method and the original landmark-based method (sec).

Dataset	Hybrid	Original
Skitter	5,318	195
IndianWeb	308	134
Flickr	17,262	287
LiveJournal	31,793	1,038

Table 9: Query times of the hybrid method and the original landmark-based method (μs).

Dataset	Hybrid	Original	BFS
Skitter	43.2	1.8	1.8×10^5
IndianWeb	50.5	1.7	9.7×10^4
Flickr	20.2	1.6	3.0×10^5
LieJournal	23.7	1.8	9.5×10^5

Table 10: Average relative errors of the hybrid method and the original landmark-based method for totally random pairs.

Dataset	Hybrid	Original
Skitter	0.041	0.041
IndianWeb	0.042	0.044
Flickr	0.024	0.024
LiveJournal	0.060	0.060

Table 7: The numbers of pairs whose distance was stored in the preprocessed data and the preprocessed data sizes for the original landmark-based method and the hybrid method. We use significantly less space than the original method.

	Number of Pairs		Size (MB)	
Dataset	Hybrid	Original	Hybrid	Original
Skitter	89,799,305	508,924,500	313	512
IndianWeb	75,883,512	414,872,400	250	417
Flickr	65,782,185	553,859,400	219	526
LiveJournal	481,819,369	1,454,271,300	923	1,461

Table 11: Average relative errors of the hybrid method and the original landmark-based method for random pairs with distance d ($d = 1, 2, 3, 4$). For short distances, the distance estimates of the hybrid method are more accurate.

	Original				Hybrid			
Dataset	$d = 1$	$d = 2$	$d = 3$	$d = 4$	$d = 1$	$d = 2$	$d = 3$	$d = 4$
Skitter	2.979	1.049	0.384	0.189	1.555	0.713	0.320	0.162
IndianWeb	5.983	2.627	1.796	1.254	1.759	0.717	0.782	0.533
Flickr	3.278	1.205	0.380	0.111	1.029	0.508	0.233	0.083
LiveJournal	3.744	1.458	0.661	0.297	2.718	1.253	0.623	0.291

the spread of influence through a social network. In *KDD*, 2003.

[34] J. Kleinberg, A. Slivkins, and T. Wexler. Triangulation and embedding using small sets of beacons. In *FOCS*, 2004.

[35] J. Leskovec, D. Huttenlocher, and J. Kleinberg. Predicting positive and negative links in online social networks. In *WWW*, 2010.

[36] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graphs over time: Densification laws, shrinking diameters and possible explanations. In *KDD*, 2005.

[37] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM Transactions on Knowledge Discovery from Data*, 1(1), 2007.

[38] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and analysis of online social networks. In *IMC*, 2007.

[39] M. E. J. Newman, S. H. Strogatz, and D. J. Watts. Random graphs with arbitrary degree distributions and their applications. *Physical Review E (Statistical, Nonlinear, and Soft Matter Physics)*, 64(2):026118 1–17, 2001.

[40] M. E. J. Newman, D. J. Watts, and S. H. Strogatz. Random graph models of social networks. *Proceedings of the National Academy of Sciences*, 99:2566–2572, 2002.

[41] I. Norros and H. Reittu. On the robustness of power-law random graphs in the finite mean, infinite variance region. arXiv:0801.1079, 2008.

[42] R. Pastor-Satorras and A. Vespignani. *Evolution and structure of the Internet: A statistical physics approach*. Cambridge University Press, 2004.

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

[44] S. A. Rahman, P. Advani, R. Schunk, R. Schrader, and D. Schomburg. Metabolic pathway analysis web service (pathway hunter tool at cubic). *Bioinformatics*, 21(7):1189–1193, 2005.

[45] S. A. Rahman and D. Schomburg. Observing local and global properties of metabolic pathways: 'load points' and 'choke points' in the metabolic networks. *Bioinformatics*, 22(14):1767–1774, 2006.

[46] M. J. Rattigan, M. Maier, and D. Jensen. Using structure indices for efficient approximation of network properties. In *KDD*, 2006.

[47] M. J. Rattigan, M. Maier, and D. Jensen. Graph clustering with network structure indices. In *ICML*, 2007.

[48] N. Robertson and P. D. Seymour. Graph minors. III. Planar tree-width. *J. Comb. Theory, Ser. B*, 36(1):49–64, 1984.

[49] N. Robertson and P. D. Seymour. Graph minors. X. obstructions to tree-decomposition. *J. Comb. Theory, Ser. B*, 52(2):153–190, 1991.

[50] L. Tang and M. Crovella. Virtual landmarks for the internet. In *SIGCOMM*, 2003.

[51] M. Thorup and U. Zwick. Approximate distance oracles. *Journal of the ACM*, 52(1):1–24, 2005.

[52] T. Tran, H. Wang, S. Rudolph, and P. Cimiano. Top-k exploration of query candidates for efficient keyword search on graph-shaped (RDF) data. In *ICDE*, 2009.

[53] A. Ukkonen, C. Castillo, D. Donato, and A. Gionis. Searching the wikipedia with contextual information. In *CIKM*, 2008.

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

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

[56] Y. Xiao, W. Wu, J. Pei, W. Wang, and Z. He. Efficiently indexing shortest paths by exploiting symmetry in graphs. In *EDBT*, 2009.

[57] S. A. Yahia, M. Benedikt, L. V. S. Lakshmanan, and J. Stoyanovich. Efficient network aware search in collaborative tagging sites. In *VLDB*, 2008.

[58] X. Zhao, A. Sala, C. Wilson, H. Zheng, and B. Y. Zhao. Orion: shortest path estimation for large social graphs. In *WOSN*, 2010.