Top-K Nearest Keyword Search on Large Graphs

Miao Qiao, Lu Qin, Hong Cheng, Jeffrey Xu Yu, Wentao Tian
The Chinese University of Hong Kong, Hong Kong, China
{mqiao,lqin,hcheng,yu,wttian}@se.cuhk.edu.hk

ABSTRACT

It is quite common for networks emerging nowadays to have labels or textual contents on the nodes. On such networks, we study the problem of *top-k nearest keyword* (k-NK) search. In a network G modeled as an undirected graph, each node is attached with zero or more keywords, and each edge is assigned with a weight measuring its length. Given a query node q in G and a keyword λ, a k-NK query seeks k nodes which contain λ and are nearest to q. k-NK is not only useful as a stand-alone query but also as a building block for tackling complex graph pattern matching problems.

The key to an accurate k-NK result is a precise shortest distance estimation in a graph. Based on the latest distance oracle technique, we build a shortest path tree for a distance oracle and use the tree distance as a more accurate estimation. With such representation, the original k-NK query on a graph can be reduced to answering the query on a set of trees and then assembling the results obtained from the trees. We propose two efficient algorithms to report the exact k-NK result on a tree. One is *query time optimized* for a scenario when a small number of result nodes are of interest to users. The other handles k-NK queries for an arbitrarily large k efficiently. In obtaining a k-NK result on a graph from that on trees, a global storage technique is proposed to further reduce the index size and the query time. Extensive experimental results conform with our theoretical findings, and demonstrate the effectiveness and efficiency of our k-NK algorithms on large real graphs.

1. INTRODUCTION

Many real-world networks emerging nowadays have labels or textual contents on the nodes. For example in a road network, a location may have labels such as "McDonald's", "hospital", and "kindergarten". In a social network, a person may have information including name, interests and skills, etc.. In a bibliographic network, a paper may have keywords and abstract, and an author may have name, affiliation and email address. In this study, we consider the problem of *top-k nearest keyword* (k-NK) search on large networks. In a network G modeled as an undirected graph, each node is attached with zero or more keywords, and each edge is assigned with a weight measuring its length. Given a query node q in G and a keyword λ, a k-NK query in the form of $Q = (q, \lambda, k)$ looks for k nodes which contain λ and are nearest to q. Different from a large body of research on k-nearest neighbor (k-NN) search on spatial networks [15, 5, 6, 18, 19, 7], we define G as a general graph without coordinates. Thus our solution can apply to a wide range of networks.

Motivation. k-NK is an important and useful query in graph search. As a stand-alone query, it has a wide range of applications. Furthermore, it can serve as a building block for tackling complex graph pattern matching problems which impose both structural and textual constraints. Here we list a few applications of k-NK queries.

Consider the social network Facebook as an example, in which personalized search based on graph structure and textual contents has become increasingly popular[1]. A person looks for 20 friends or potential friends who like *hiking* to participate in a hiking activity. Intuitively, if two persons share some common friends, i.e., they are within two hops away, they are more likely to become friends. In contrast, if they are far away from each other in the network, they are less likely to establish a link. Thus the problem is to find 20 persons who like *hiking* and are nearest to the person who serves as the organizer. It can be answered by a k-NK query. More generally, we also consider a query containing multiple keywords connected by AND or OR operators to express more complex semantics, e.g., a person looks for k friends or potential friends who like *hiking* AND (OR) *photography* and are nearest to him.

Take a road network with locations associated with keywords as another example. For parents looking for k *kindergartens* nearest to their home for their children, their requirements can be expressed by a k-NK query where the query node is the home location, and the keyword is "kindergarten".

In the third example, we show how k-NK queries serve as a building block for solving the graph pattern matching problem. Consider a couple who wants to buy a house. They have some constraints like *having a kindergarten and a hospital within 3 km, and a supermarket within 1 km of their home*. These constraints can be expressed as a star pattern, and the pattern matching problem can be decomposed into three k-NK queries with keywords "kindergarten", "hospital" and "supermarket" respectively and $k = 1$ for each potential house location to be considered.

Recently, Bahmani and Goel [1] have designed a *Partitioned Multi-Indexing* (PMI) scheme to answer k-NK queries approximately. PMI is an inverted index built based on distance oracle [20] which is a distance estimation technique. Given a k-NK query $Q = (q, \lambda, k)$, it returns k nodes containing keyword λ in ascending order of their approximate distance from the query node q. PMI inherits the $2\log_2|V| - 1$ approximation factor for distance estimation from distance oracle [20], where V is the set of nodes in the

[1] https://www.facebook.com/about/graphsearch

Proceedings of the VLDB Endowment, Vol. 6, No. 10

graph. The major drawback of PMI is that its distance estimation error could be quite large in practice. This can greatly distort the ranking of the candidate nodes carrying the query keywords, and thus lead to a low result quality.

In this work, we study how to answer k-NK queries accurately and efficiently using compact index. The key to an accurate k-NK result is a precise shortest distance estimation in a graph. As we use a general graph model, existing k-NN solutions on spatial networks [15, 5, 6, 18, 19, 7] cannot be applied, as they usually rely on specialized structures that leverage properties of spatial data to optimize their solutions. Instead we use distance oracle [20] as the fundamental distance estimation framework. For each component of a distance oracle, we will build a shortest path tree, based on which we can estimate the shortest distance between two nodes by their tree distance. The tree distance is more accurate than the distance estimated by distance oracle, which we call *witness distance* to distinguish. As we transform a distance oracle on a graph into a set of shortest path trees, the original k-NK query on the graph can be reduced to answering the k-NK query on a set of trees. Thus we first focus on processing k-NK queries to find exact top-k answers on a tree. Then we study how to assemble the results obtained from the trees to form the approximate top-k answers on the graph.

Contributions. Our main contributions in this work are summarized as follows.

(1) Given a tree, we first consider a common scenario when users are interested in a small number of answer nodes bounded by a small constant $\overline{k}$, i.e., $k \leq \overline{k}$. We propose the first algorithm tree-boundk with query time $O(k + \log |V_\lambda|)$, where $|V_\lambda|$ is the number of nodes carrying the query keyword λ, and index size $O(\overline{k} \cdot |\mathsf{doc}(V)|)$, where $|\mathsf{doc}(V)|$ is the total number of keywords on all the nodes in the graph.

(2) Next we remove the $\overline{k}$ restriction and handle k-NK queries for an arbitrary k on a tree. We propose the second algorithm tree-pivot with query time $O(k \cdot \log |V|)$ and index size $O(|\mathsf{doc}(V)| \cdot \log |V|)$ which is independent of k, thus is more scalable.

(3) Based on our proposed tree algorithms, we present our algorithm for approximate k-NK query on a graph. We propose a global storage technique to further reduce the index size and the query time. We also show how to extend our methods to handle a query with multiple keywords.

(4) Our experimental evaluation demonstrates the effectiveness and efficiency of our k-NK algorithms on large real-world networks. We show the superiority of our methods in ranking top-k answer nodes accurately, when compared with the state-of-the-art top-k keyword search method PMI [1].

Roadmap. The rest of the paper is organized as follows. Section 2 formally defines the problem. Section 3 discusses two existing related studies and their drawbacks. Section 4 presents our framework. Sections 5 and 6 introduce two proposed algorithms to answer k-NK queries on a tree for a small k and an arbitrary k respectively using compact index structures. Section 7 elaborates on the way to answer k-NK queries on a graph by approximating the graph with a bounded number of trees. Section 8 presents extensive experimental evaluation. Section 9 reviews the previous works related to ours. Finally, Section 10 concludes the paper.

2. PROBLEM DEFINITION

We model a weighted undirected graph as $G(V, E)$, where $V(G)$ represents the set of nodes and $E(G)$ represents the set of edges in G. We use V and E to denote $V(G)$ and $E(G)$ if the context is obvious. Each edge $(u, v) \in E$ has a positive weight, denoted as $\mathsf{weight}(u, v)$. A path $p = (v_1, v_2, \cdots, v_l)$ is a sequence of l nodes in V such that for each $v_i (1 \leq i < l)$, $(v_i, v_{i+1}) \in E$. The weight of a path is the total weight of all edges on the path. For any two nodes $u \in V$ and $v \in V$, the distance of u and v on G, $\mathsf{dist}(u, v)$, is the minimum weight of all paths from u to v in G. Each node $v \in V$ contains a set of zero or more keywords which is denoted as $\mathsf{doc}(v)$. The union of keywords for all nodes in G is denoted as $\mathsf{doc}(V)$. Note that $\mathsf{doc}(V)$ is a multiset and $|\mathsf{doc}(V)| = \sum_{v \in V} |\mathsf{doc}(v)|$. We use $V_\lambda \subseteq V$ to denote the set of nodes carrying keyword λ in V.

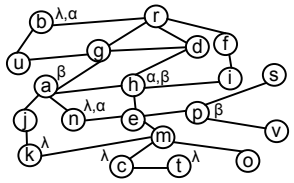


Figure 1: A Graph G with Keywords

DEFINITION 1. *Given a graph $G(V, E)$, a top-k nearest keyword (k-NK) query is a triple $Q = (q, \lambda, k)$, where $q \in V$ is a query node in G, λ is a keyword, and k is a positive integer. Given a query Q, a node $v \in V$ is a keyword node w.r.t. Q if v contains keyword λ, i.e., $v \in V_\lambda$. The result is a set of k keyword nodes, denoted as $R = \{v_1, v_2, \cdots, v_k\} \subseteq V_\lambda$, and there does not exist a node $u \in V_\lambda \setminus R$ such that $\mathsf{dist}(q, u) < \max_{v \in R} \mathsf{dist}(q, v)$. To further report the distance in the top-k result, we can use the form $R = \{v_1 : \mathsf{dist}(q, v_1), v_2 : \mathsf{dist}(q, v_2), \cdots, v_k : \mathsf{dist}(q, v_k)\}$.*

In this paper, we aim at answering a k-NK query $Q = (q, \lambda, k)$ on a graph G. For simplicity, we assume that there is only one keyword λ in the query. We will discuss how to answer a query containing multiple keywords with AND and OR semantics.

Example 1: Fig. 1 shows a graph G. Assume that the weight of each edge is 1. For a k-NK query $Q = (f, \lambda, 3)$, the keyword node set is $V_\lambda = \{b, c, k, n, t\}$. The result of Q is $R = \{b : 2, n : 4, k : 5\}$ since $\mathsf{dist}(f, b) = 2$, $\mathsf{dist}(f, n) = 4$, and $\mathsf{dist}(f, k) = 5$. □

3. EXISTING SOLUTIONS

A straightforward approach to answering a k-NK query $Q = (q, \lambda, k)$ on G is to use Dijkstra's algorithm to search from the query node q and output k nearest keyword nodes in nondecreasing order of their distances to q. The time complexity is $O(|E| + |V| \cdot \log |V|)$. Obviously, Dijkstra's algorithm is inefficient when the size of the graph is large or the keyword nodes are far away from q.

In the literature, [1] and [22] design different indexing schemes to process (top-k) nearest keyword queries on a graph or a tree. We introduce the two methods in the following two subsections.

3.1 Approximate k-NK on a Graph

Bahmani and Goel [1] find an approximate answer to a k-NK query in a graph based on a distance oracle [20].

Distance Oracle: Distance oracle is a technique for estimating the distance of two nodes in a graph [20]. Given a graph G, a distance oracle is a Voronoi partition of $V(G)$ determined by a set of randomly selected *center nodes*. More specifically, given a number n_c, we randomly select n_c nodes from $V(G)$ as the center nodes to construct a distance oracle $\mathcal{O}$. Then the partition is constructed by assigning each node $v \in V(G)$ to its nearest center node, denoted as $\mathsf{wit}_\mathcal{O}(v)$, which is called the witness node of v w.r.t. $\mathcal{O}$. If v is a center node, $\mathsf{wit}_\mathcal{O}(v) = v$. For each node $v \in V(G)$, the shortest distance from v to its witness node, i.e., $\mathsf{dist}(v, \mathsf{wit}_\mathcal{O}(v))$, is precomputed. After constructing $\mathcal{O}$, given two nodes u and v in G, if u and v are in the same partition in $\mathcal{O}$, i.e., $\mathsf{wit}_\mathcal{O}(u) =$

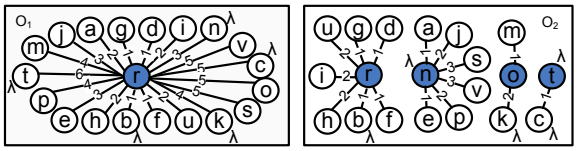


Figure 2: Two Distance Oracles $\mathcal{O}_1$ and $\mathcal{O}_2$

$\mathsf{wit}_{\mathcal{O}}(v)$, we compute the estimated distance, called *witness distance*, as $\overline{\mathsf{dist}}_{\mathcal{O}}(u,v) = \mathsf{dist}(u, \mathsf{wit}_{\mathcal{O}}(u)) + \mathsf{dist}(v, \mathsf{wit}_{\mathcal{O}}(v))$. If u and v are not in the same partition in $\mathcal{O}$, $\overline{\mathsf{dist}}_{\mathcal{O}}(u,v) = +\infty$.

One distance oracle is usually not enough for distance estimation in a graph G. It cannot estimate the distance of two nodes in different partitions. Even for two nodes in the same partition, the estimation may have a large error. Therefore, a set of $r = p \times \log|V|$ distance oracles $\{\mathcal{O}_1, \mathcal{O}_2, \cdots, \mathcal{O}_r\}$ are constructed, where p can be considered as a constant[2]. The algorithm is processed in $\log|V|$ phases. In phase i $(0 \le i < \log|V|)$, p distance oracles are constructed where each distance oracle contains 2^i randomly selected center nodes. Given r distance oracles, the distance of two nodes u and v in G can be estimated as an upper bound $\overline{\mathsf{dist}}(u,v) = \min_{1\le i\le r} \overline{\mathsf{dist}}_{\mathcal{O}_i}(u,v)$.

The time complexity to compute the estimated distance $\overline{\mathsf{dist}}(u,v)$ for any two nodes u and v in a graph G is $O(\log|V|)$. The distance oracles consume $O(|V| \cdot \log|V|)$ space. Das Sarma et al. [20] prove that when $p = \Theta(|V|^{1/\log|V|})$, the estimated distance can be bounded by $\mathsf{dist}(u,v) \le \overline{\mathsf{dist}}(u,v) \le (2\log_2|V|-1)\cdot\mathsf{dist}(u,v)$ with a high probability.

Example 2: Fig. 2 shows two distance oracles $\mathcal{O}_1$ and $\mathcal{O}_2$ for the graph shown in Fig. 1. There is one center node r in $\mathcal{O}_1$, and four center nodes r, n, o and t in $\mathcal{O}_2$. The distance of nodes j and s is estimated as $\overline{\mathsf{dist}}(j,s) = \min\{\overline{\mathsf{dist}}_{\mathcal{O}_1}(j,s), \overline{\mathsf{dist}}_{\mathcal{O}_2}(j,s)\} = \min\{\mathsf{dist}(j,r) + \mathsf{dist}(s,r), \mathsf{dist}(j,n) + \mathsf{dist}(s,n)\} = 5$. □

Answering k-NK with Distance Oracle: [1] designs a Partitioned Multi-Indexing (PMI) scheme which uses a set of distance oracles to answer a k-NK query in a graph. For each partition in a distance oracle $\mathcal{O}_i$, an inverted list is constructed for each keyword in the partition. Specifically, for a partition with a center node c and a keyword λ, the inverted list contains all nodes in the partition that contain keyword λ ranked in nondecreasing order of their distances to c. Given a k-NK query $Q = (q, \lambda, k)$ and a distance oracle $\mathcal{O}_i$, the algorithm first finds the partition that q belongs to in $\mathcal{O}_i$. The result w.r.t. $\mathcal{O}_i$ is the first k elements in the inverted list for λ in the partition, denoted as $R_{\mathcal{O}_i} = \{u_1 : \mathsf{dist}(c,u_1) + \mathsf{dist}(c,q), u_2 : \mathsf{dist}(c,u_2) + \mathsf{dist}(c,q), \cdots, u_k : \mathsf{dist}(c,u_k) + \mathsf{dist}(c,q)\}$. The final result R is computed by merging the nodes in each $R_{\mathcal{O}_i}$ and maintaining k nodes with the shortest distances to q. The query time complexity is $O(k\cdot\log|V|)$. We illustrate the algorithm using the following example.

Example 3: Consider the graph in Fig. 1 and two distance oracles in Fig. 2. For keyword λ, the inverted list for the partition centered at node r in $\mathcal{O}_1$ has 5 elements $\{b:1, n:3, k:4, c:5, t:6\}$. The inverted list for the partition centered at node o in $\mathcal{O}_2$ has 1 element $\{k:2\}$. Given a k-NK query $Q = (m, \lambda, 2)$, from $\mathcal{O}_1$, we can get a result $R_{\mathcal{O}_1} = \{b : 1 + \mathsf{dist}(r,m), n : 3 + \mathsf{dist}(r,m)\} = \{b:5, n:7\}$, and from $\mathcal{O}_2$, we can get a result $R_{\mathcal{O}_2} = \{k : 2 + \mathsf{dist}(o,m)\} = \{k:3\}$. By merging $R_{\mathcal{O}_1}$ and $R_{\mathcal{O}_2}$, the final answer is $R = \{k:3, b:5\}$. The exact answer is $R = \{c:1, k:1\}$ according to Fig. 1. □

Limitation: Although in theory, the witness distance used by [1] can be bounded by a factor of $2\log_2|V|-1$ of the exact distance with a high probability, in practice, however, we find the distance estimation error can be quite large. For example, for the graph G in Fig. 1 and two distance oracles $\mathcal{O}_1$ and $\mathcal{O}_2$ in Fig. 2, for two nodes s and v, the witness distance in $\mathcal{O}_1$ is $\overline{\mathsf{dist}}_{\mathcal{O}_1}(s,v) = \mathsf{dist}(s,r) + \mathsf{dist}(v,r) = 10$, and that in $\mathcal{O}_2$ is $\overline{\mathsf{dist}}_{\mathcal{O}_2}(s,v) = \mathsf{dist}(s,n) + \mathsf{dist}(v,n) = 6$. However, the exact distance is $\mathsf{dist}(s,v) = 2$ in G, which is much smaller than both $\overline{\mathsf{dist}}_{\mathcal{O}_1}(s,v)$ and $\overline{\mathsf{dist}}_{\mathcal{O}_2}(s,v)$. The inaccurate distance estimation can greatly distort the ranking of the nodes carrying the query keyword, and thus lead to a low result quality, as illustrated in Example 3.

[2]In [20], the set $\{\mathcal{O}_1, \mathcal{O}_2, \cdots, \mathcal{O}_r\}$ is defined as a distance oracle.

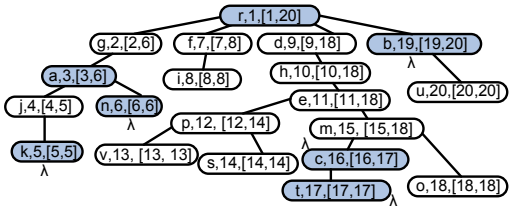


Figure 3: A Tree T with Preorder and Interval on Each Node

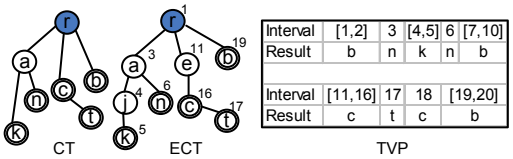


Figure 4: $\mathsf{CT}(\lambda)$, $\mathsf{ECT}(\lambda)$ and $\mathsf{TVP}(\lambda)$ for Keyword λ

3.2 Exact 1-NK on a Tree

Tao et al. [22] compute the exact answer to a 1-NK query on a tree $T(V,E)$. Given a query $Q = (q, \lambda, 1)$, the result is the nearest node in T that contains keyword λ, denoted as $\mathsf{NN}(q,\lambda)$. The basic idea is as follows. We label a node v with the sequence number of v in the preorder traversal of T. For a certain keyword λ, all nodes with the preorder label in the interval $[1,|V|]$ can be partitioned into several disjointed intervals, such that any node v in the same interval shares an identical $\mathsf{NN}(v,\lambda)$. The partition is called tree Voronoi partition of λ, denoted as $\mathsf{TVP}(\lambda)$. By precomputing $\mathsf{TVP}(\lambda)$ for all keywords λ on the tree, a query $Q = (q,\lambda,1)$ can be answered in $O(\log|V_\lambda|)$ time using a binary search in $\mathsf{TVP}(\lambda)$.

In order to compute $\mathsf{TVP}(\lambda)$ for all keywords λ in T efficiently, two new data structures, namely, Compact Tree $\mathsf{CT}(\lambda)$ and Extended Compact Tree $\mathsf{ECT}(\lambda)$, are proposed in [22].

DEFINITION 2. **(Compact Tree and Extended Compact Tree)** *For a tree T and a keyword λ, a compact tree $\mathsf{CT}(\lambda)$ is a tree that keeps only two types of nodes in T: a keyword node that contains keyword λ, and a node that has at least two direct subtrees containing nodes carrying keyword λ. In the preorder traversal of T, for two successive nodes u and v, if $\mathsf{NN}(u,\lambda) \ne \mathsf{NN}(v,\lambda)$, v is called a change node. An extended compact tree $\mathsf{ECT}(\lambda)$ is a tree constructed by adding all change nodes into the compact tree $\mathsf{CT}(\lambda)$.*

Using $\mathsf{ECT}(\lambda)$, $\mathsf{TVP}(\lambda)$ can be constructed easily. In [22], the authors prove that the total size of all compact trees and all extended compact trees for all keywords in the tree $T(V,E)$ is bounded by $O(|\mathsf{doc}(V)|)$. The time to compute all compact trees and all extended compact trees for all keywords in the tree $T(V,E)$ is bounded by $O(|\mathsf{doc}(V)|\cdot\log|V|)$.

Example 4: Fig. 3 shows a tree with the preorder label from 1 to 20 on its nodes. For keyword λ, there are 5 keyword nodes b, c, k, n, t. For node s, $\mathsf{NN}(s,\lambda) = c$. The compact tree of λ, $\mathsf{CT}(\lambda)$, is shown on the left part of Fig. 4. Node r is in $\mathsf{CT}(\lambda)$ because r has three direct subtrees with nodes carrying keyword λ. e is not in $\mathsf{CT}(\lambda)$ because e is not a keyword node and e has only one direct subtree rooted at m with nodes carrying keyword λ. The extended compact tree of λ, $\mathsf{ECT}(\lambda)$, is shown in the middle part of Fig. 4 with the

preorder label marked beside each node. Node e is in $\mathsf{ECT}(\lambda)$, because for its parent node h, $\mathsf{NN}(h,\lambda) = b \neq \mathsf{NN}(e,\lambda) = c$. The tree Voronoi partition of λ, $\mathsf{TVP}(\lambda)$, is shown on the right part of Fig. 4. For node s with preorder label 14, it is in the interval $[11, 16]$, thus $\mathsf{NN}(s,\lambda) = c$ as listed in $\mathsf{TVP}(\lambda)$. □

4. SOLUTION OVERVIEW

Answering k-NK on a Graph using Tree Distance: To address the drawback of witness distance, in this paper, we propose to use *tree distance* in processing a k-NK query. We observe that for a partition of a distance oracle, we can construct a shortest path tree rooted at the center node of the partition. Since a tree contains more structural information than a star, using tree distance will be more accurate than using witness distance for estimating the distance of two nodes. For a distance oracle $\mathcal{O}_i$, let the set of trees constructed in $\mathcal{O}_i$ be T_i. T_i can be considered as a tree by adding a virtual root and several virtual edges with weight $+\infty$ that connect the new virtual root to every root node in T_i respectively. Let the k-NK result on tree T be R_T. Suppose we have an algorithm to compute R_T on a tree T, we can solve the k-NK problem in a graph by merging R_{T_i} for each tree T_i, $1 \leq i \leq r$. Obviously, such a result will be more accurate than the result by [1]. The following example illustrates the k-NK query processing based on tree distance.

Example 5: For the distance oracles $\mathcal{O}_1$ and $\mathcal{O}_2$ shown in Fig. 2, the corresponding shortest path trees T_1 and T_2 are shown in Fig. 5. For T_1, there is only 1 tree rooted at r because there is only 1 partition in $\mathcal{O}_1$. For T_2, there are 4 trees rooted at nodes n, o, r, t respectively, because there are 4 partitions in $\mathcal{O}_2$. In each tree, the path from any node to the root node is a shortest path in the original graph. For two nodes s and v, their tree distance is 2 in both T_1 and T_2, the same as the exact distance $\mathsf{dist}(s,v)$ in G. For a k-NK query $Q = (m, \lambda, 2)$, we have $R_{T_1} = \{c:1, t:2\}$, and $R_{T_2} = \{k:1\}$. By merging R_{T_1} and R_{T_2}, we get $R = \{c:1, k:1\}$. Such a result is much better than the result in Example 3 computed using witness distance for the same query. □

With the tree distance formulation, the key operation in answering a k-NK query on a graph is to answer the k-NK query on a tree. Therefore, we start with processing a k-NK query on a tree.

Answering k-NK on a Tree: We show that it is nontrivial to answer a k-NK query on a tree efficiently even if k is bounded. Our first attempt is to extend the existing 1-NK solution on a tree $T(V,E)$ in [22]. Recall that in [22], for a certain keyword λ, the range $[1, |V|]$ is partitioned into several disjoint intervals, and nodes with the preorder label in an identical interval share the same 1-NK result. When $k \geq 2$, each interval needs to be further partitioned to ensure that all nodes with the preorder label in the same interval share an identical k-NK result. The number of intervals increases exponentially w.r.t. the number of keyword nodes on the tree until it reaches $|V|$ for a keyword λ. Clearly, using such an approach, the index size is too large in practice even for a small k. Our second attempt is that, for each node v on the tree $T(V,E)$ and each keyword λ, we precompute its $\overline{k}$ nearest nodes that contain λ. When processing a query $Q = (q, \lambda, k)$ with $k \leq \overline{k}$, we can simply retrieve the precomputed result on node q and output the first k nodes directly. Such an approach is impractical because for each keyword λ, we need $O(\overline{k} \cdot |V|)$ space to store the precomputed results.

In the following, we first introduce two algorithms for answering exact k-NK on a tree $T(V,E)$. Our first algorithm tree-boundk can only handle bounded k values with query processing time $O(k + \log |V_\lambda|)$ and index size $O(\overline{k} \cdot |\mathsf{doc}(V)|)$ for all keywords where $\overline{k}$ is an upper bound value of k. Our second algorithm tree-pivot can handle an arbitrary k with query processing time $O(k \cdot \log |V|)$ and index size $O(|\mathsf{doc}(V)| \cdot \log |V|)$ for all keywords which is independent of k. We then show our algorithm for approximate k-NK on a graph by merging results on a bounded number of trees. We propose a global storage technique to further reduce the index size and the query time on a graph. Finally we show how to extend our method to handle a query with multiple keywords.

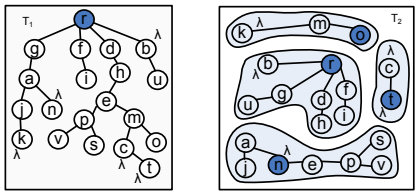


Figure 5: Shortest Path Trees T_1 and T_2

Algorithm 1: tree-boundk (Q,T)

Input: A k-NK query $Q = (q, \lambda, k)$, and a tree T.
Output: Answer for Q on T.
1 $R \leftarrow \emptyset$;
2 $(u, u') \leftarrow$ the entry edge of q on $\mathsf{CT}(\lambda)$;
3 $R \leftarrow R \otimes_k (\mathsf{cand}_\lambda(u) \oplus \mathsf{dist}(q,u))$;
4 $R \leftarrow R \otimes_k (\mathsf{cand}_\lambda(u') \oplus \mathsf{dist}(q,u'))$;
5 **return** R;

5. K-NK ON A TREE FOR A SMALL K

In this section, we study how to answer a k-NK query $Q = (q, \lambda, k)$ on a tree $T(V,E)$. We first consider a common scenario when users are interested in a small number of answer nodes bounded by a small constant $\overline{k}$, i.e., $k \leq \overline{k}$. Recall that for a keyword λ, its compact tree $\mathsf{CT}(\lambda)$ keeps all the structural information of λ on the tree T. Our idea is to precompute the top-$\overline{k}$ results for every keyword λ and every node on $\mathsf{CT}(\lambda)$. Since the total size of all compact trees is bounded by $O(|\mathsf{doc}(V)|)$, the total space to store the top-$\overline{k}$ results of nodes on all compact trees is bounded by $O(\overline{k} \cdot |\mathsf{doc}(V)|)$. Given a query $Q = (q, \lambda, k)$, if q is on $\mathsf{CT}(\lambda)$, we can simply report the precomputed answer on $\mathsf{CT}(\lambda)$. If q is not on $\mathsf{CT}(\lambda)$, we need to find a way to construct the answer using the precomputed results as well as the structure of $\mathsf{CT}(\lambda)$ and T. In the following, we first introduce how to answer a k-NK query using $\mathsf{CT}(\lambda)$, followed by discussions on the construction of the index.

5.1 Query Processing

For a keyword λ, and each node v in the compact tree $\mathsf{CT}(\lambda)$, we use a *candidate list* $\mathsf{cand}_\lambda(v)$ to denote the precomputed k-NK results for $k = \overline{k}$ on node v ranked in nondecreasing order of their distances to v, in the form of $\mathsf{cand}_\lambda(v) = \{v_1 : \mathsf{dist}(v,v_1), v_2 : \mathsf{dist}(v,v_2), \cdots, v_{\overline{k}} : \mathsf{dist}(v,v_{\overline{k}})\}$ where $\mathsf{dist}(v,v_1) \leq \mathsf{dist}(v,v_2) \leq \cdots \leq \mathsf{dist}(v,v_{\overline{k}})$. Given a query $Q = (q, \lambda, k)$ on a tree $T(V,E)$ where $k \leq \overline{k}$, if q is in $\mathsf{CT}(\lambda)$, we can simply report the first k elements in $\mathsf{cand}_\lambda(q)$ as the answer. The difficult case is when q is not in $\mathsf{CT}(\lambda)$. In order to answer such a query, we define an *entry edge* to be the edge in $\mathsf{CT}(\lambda)$ that is nearest to q. Intuitively, the entry edge plays a role of connecting the query node q to the compact tree $\mathsf{CT}(\lambda)$. The formal definition of entry edge is as follows.

DEFINITION 3. ***(Entry Node and Entry Edge)*** *Given a compact tree $\mathsf{CT}(\lambda)$, for each edge (u,u') on $\mathsf{CT}(\lambda)$ with u' being a child node of u, (u,u') represents a unique path from u to u' on the original tree T. For any node v on T, we say v* ***sticks to*** *$\mathsf{CT}(\lambda)$, denoted as $v \in_s \mathsf{CT}(\lambda)$, if and only if there exists an edge (u,u') on $\mathsf{CT}(\lambda)$ such that v is on the path from u to u' on T, otherwise v does not stick to $\mathsf{CT}(\lambda)$, denoted as $v \notin_s \mathsf{CT}(\lambda)$. For a node q on T, let v be the first node on the path from q to the root node of T such that $v \in_s \mathsf{CT}(\lambda)$. v is called the Entry Node of q w.r.t. λ,*

Algorithm 2: operator $R \oplus \delta$

Input: Candidate list $R = \{u_1 : d_{u_1}, u_2 : d_{u_2}, \cdots\}$, distance δ.
Output: A candidate list by adding δ to all distances in R.

```
1 R' ← ∅;
2 for i = 1 to |R| do
3     R' ← R' ⋃ {u_i : d_{u_i} + δ};
4 return R';
```

denoted as $\mathsf{EN}_\lambda(q)$. *The corresponding edge* (u, u') *on* $\mathsf{CT}(\lambda)$ *is called the Entry Edge of* q *w.r.t.* λ, *denoted as* $\mathsf{EE}_\lambda(q)$.

Note that for a node q and a keyword λ, $\mathsf{EE}_\lambda(q)$ is an edge on the compact tree $\mathsf{CT}(\lambda)$, and $\mathsf{EN}_\lambda(q)$ is a node on the original tree T. We use an example to illustrate the entry node and entry edge.

Example 6: For the tree T shown in Fig. 3 and keyword λ, the compact tree $\mathsf{CT}(\lambda)$ is shown on the left part of Fig. 4. For ease of illustration, we also mark the nodes in $\mathsf{CT}(\lambda)$ dark on the tree T in Fig. 3. For edge (r, c) in $\mathsf{CT}(\lambda)$, $h \in_s \mathsf{CT}(\lambda)$ because h is on the path from r to c in T. $p \notin_s \mathsf{CT}(\lambda)$ since p is not on the tree path of any $\mathsf{CT}(\lambda)$ edge. For node v, its entry node is $\mathsf{EN}_\lambda(v) = e$, as e is the first node on the path (v, p, e, h, d, r) such that $e \in_s \mathsf{CT}(\lambda)$. The entry edge for v is $\mathsf{EE}_\lambda(v) = (r, c)$ since the entry node e for v is on the path from r to c in T. The entry nodes and entry edges for some other nodes in T are listed in the following table. □

Node	g	j	d	e	p	u
EN_λ	g	j	d	e	e	b
EE_λ	(r, a)	(a, k)	(r, c)	(r, c)	(r, c)	(r, b)

The Algorithm: Given a tree $T(V, E)$, for keyword λ, all keyword nodes are contained in $\mathsf{CT}(\lambda)$. For any node $q \in V$, the path from q to any keyword node will go through the entry node $\mathsf{EN}_\lambda(q)$. Based on such property, the result of a query $Q = (q, \lambda, k)$ is identical with the result of the query $Q' = (\mathsf{EN}_\lambda(q), \lambda, k)$. However, $\mathsf{EN}_\lambda(q)$ may not be on $\mathsf{CT}(\lambda)$, thus the result of Q' is not necessarily precomputed. Let $(u, u') = \mathsf{EE}_\lambda(q)$, since $\mathsf{EN}_\lambda(q)$ is on the path from u to u' on the tree T, the path from $\mathsf{EN}_\lambda(q)$ to any keyword node in T will go through either u or u'. Thus, the answer for Q' can be constructed by merging the precomputed candidate lists $\mathsf{cand}_\lambda(u)$ and $\mathsf{cand}_\lambda(u')$ on $\mathsf{CT}(\lambda)$.

Our algorithm for processing a query $Q = (q, \lambda, k)$ on a tree T is shown in Algorithm 1. We assume that the compact tree $\mathsf{CT}(\lambda)$ for each keyword λ and the list $\mathsf{cand}_\lambda(u)$ for every node u on $\mathsf{CT}(\lambda)$ have been computed. After initializing the result R in line 1, we find the entry edge (u, u') for q on $\mathsf{CT}(\lambda)$ (line 2). We add a distance $\mathsf{dist}(q, u)$ to every node in $\mathsf{cand}_\lambda(u)$ using the $\oplus$ operator, to reflect the distance from q to a keyword node via u. We then merge the new result into R using the $\otimes_k$ operator (line 3). Similarly we apply the two operators to $\mathsf{cand}_\lambda(u')$ with the distance $\mathsf{dist}(q, u')$ (line 4). We will describe the operators $\oplus$ and $\otimes_k$ later. We use the following example to illustrate the algorithm.

Example 7: Given the tree T shown in Fig. 3 and $\mathsf{CT}(\lambda)$ on the left part of Fig. 4, for a query $Q = (o, \lambda, 2)$, the entry edge $\mathsf{EE}_\lambda(o) = (r, c)$. Suppose the lists $\mathsf{cand}_\lambda(r) = \{b : 1, n : 3\}$ and $\mathsf{cand}_\lambda(c) = \{c : 0, t : 1\}$ are precomputed. By adding $\mathsf{dist}(o, r) = 5$ to $\mathsf{cand}_\lambda(r)$, and adding $\mathsf{dist}(o, c) = 2$ to $\mathsf{cand}_\lambda(c)$, we get the new lists $\{b : 6, n : 8\}$ for r and $\{c : 2, t : 3\}$ for c. We merge the two lists and get the final result $R = \{c : 2, t : 3\}$. □

The efficiency of Algorithm 1 depends on three operations. The first operation is to find the entry edge for any node on T (line 2). The second operation is to calculate the distance of any two nodes on T, e.g., $\mathsf{dist}(q, u)$ and $\mathsf{dist}(q, u')$ (line 3-4). The third operation is to merge two sorted lists into a new one using operators $\oplus$ and $\otimes_k$ (line 3-4). Next, we discuss the three operations separately.

Algorithm 3: operator $R_1 \otimes_k R_2$

Input: Two sorted candidate lists $R_1 = \{u_1 : d_{u_1}, u_2 : d_{u_2}, \cdots\}$
$R_2 = \{v_1 : d_{v_1}, v_2 : d_{v_2}, \cdots\}$, and result size k.
Output: The merged candidate list.

```
1 R ← ∅; i ← 1; j ← 1;
2 while (i < |R_1| or j < |R_2|) and |R| ≤ k do
3     if i < |R_1| and (d_{u_i} ≤ d_{v_j} or j ≥ |R_2|) then
4         if u_i ∉ R then R ← R ⋃ {u_i : d_{u_i}};
5         i ← i + 1;
6     else if j < |R_2| and (d_{v_j} ≤ d_{u_i} or i ≥ |R_1|) then
7         if v_j ∉ R then R ← R ⋃ {v_j : d_{v_j}};
8         j ← j + 1;
9 return R;
```

Finding the Entry Edge: Given a keyword λ, for any node v on a tree $T(V, E)$, our idea of finding the entry edge $\mathsf{EE}_\lambda(v)$ of v is similar to the idea of finding the 1-NK answer using the tree Voronoi partition $\mathsf{TVP}(\lambda)$ in [22]. For the range $[1, |V|]$, we partition it into several disjoint intervals, such that nodes with the preorder label in the same interval share an identical entry edge. We call such partition an *entry edge partition* for λ, denoted as $\mathsf{EEP}(\lambda)$. Given $\mathsf{EEP}(\lambda)$, $\mathsf{EE}_\lambda(v)$ can be computed easily using a binary search in $\mathsf{EEP}(\lambda)$ in $O(\log |V_\lambda|)$ time. In the next subsection, we show how to build $\mathsf{EEP}(\lambda)$ for all keywords efficiently and prove that the total size of $\mathsf{EEP}(\lambda)$ for all keywords in T is bounded by $O(\mathsf{doc}|V|)$.

Computing Tree Distance: Given a tree $T(V, E)$ with root r, suppose the distance from r to every node in T has been precomputed. For any two nodes u and v on T, we denote $\mathsf{LCA}(u, v)$ as their lowest common ancestor. The distance of u and v can be computed as $\mathsf{dist}(u, v) = \mathsf{dist}(r, u) + \mathsf{dist}(r, v) - 2\mathsf{dist}(r, \mathsf{LCA}(u, v))$. Using the techniques in [2], $\mathsf{LCA}(u, v)$ can be found in $O(1)$ time using $O(|V|)$ index space. Thus $\mathsf{dist}(u, v)$ for any two nodes u and v on T can be computed in $O(1)$ time using $O(|V|)$ index space.

Merging Results: The results are merged using two operators $\oplus$ and $\otimes_k$. Algorithm 2 shows the operator $\oplus$, which takes a candidate list R and a distance δ as input, and outputs a candidate list by adding δ to all distances in R. The time complexity for the $\oplus$ operator is $O(|R|)$. Algorithm 3 shows the operator $\otimes_k$, which takes two candidate lists R_1 and R_2 sorted in nondecreasing order of the distances, and a value k as input, and outputs the merged candidate list R. R contains at most k elements sorted in nondecreasing order of the distances. R can be constructed by visiting each element in R_1 and R_2 at most once. The time complexity for the $\otimes_k$ operator is $O(\min\{|R_1| + |R_2|, k\})$. The $\otimes_k$ and $\oplus$ operators satisfy the commutative, associative and distributive laws as follows.

(Commutative Law) $R_1 \otimes_k R_2 = R_2 \otimes_k R_1$.
(Associative Law) $(R_1 \otimes_k R_2) \otimes_k R_3 = R_1 \otimes_k (R_2 \otimes_k R_3)$.
(Distributive Law) $(R_1 \otimes_k R_2) \oplus d = (R_1 \oplus d) \otimes_k (R_2 \oplus d)$.

THEOREM 1. *Algorithm 1 computes the exact* k-NK *answer for a query* $Q = (q, \lambda, k)$ *on a tree* $T(V, E)$ *in* $O(k + \log |V_\lambda|)$ *time.*

Algorithm 1 uses the novel idea of entry edge, and elegantly extends the 1-NK method [22] to handle k-NK ($k > 1$) with the same query time complexity, except for an extra linear cost $O(k)$ indispensable for reporting the results.

Given the tree T, for every keyword λ in T, the compact tree $\mathsf{CT}(\lambda)$ can be constructed using the algorithm in [22], and in addition, two more indexes need to be constructed. The first index, the *entry edge partition* $\mathsf{EEP}(\lambda)$, is to find the entry edge for any node on T. The second index is the *candidate list* $\mathsf{cand}_\lambda(v)$ for every node on $\mathsf{CT}(\lambda)$. Below we show how to construct the two indexes.

5.2 Construction of Entry Edge Partition

Given a tree $T(V, E)$, for each keyword λ, sharing the similar idea with the tree Voronoi partition $\mathsf{TVP}(\lambda)$, we construct an entry

Algorithm 4: EEP-construct (T,CT(λ))

Input: A tree $T(V, E)$ and a labelled compact tree CT(λ).
Output: Entry edge partition EEP(λ).

```
1  r ← the original root of CT(λ);
2  EEP(λ) ← ∅;
3  partition(EEP(λ), [1, |V|], (φ, r), CT(λ));
4  return EEP(λ);
5  Procedure partition(EEP(λ), interval [s, t], edge (u, u'), CT(λ))
6  foreach subnode u'' of u' on CT(λ) in increasing preorder do
7      [s', t'] ← interval of (u', u'');
8      if s < s' then add ([s, s' − 1], (u, u')) to EEP(λ);
9      partition(EEP(λ), [s', t'], (u', u''), CT(λ));
10     s ← t' + 1;
11 if s ≤ t then add ([s, t], (u, u')) to EEP(λ);
```

edge partition EEP(λ), which divides $[1, |V|]$ into several disjoint intervals, such that nodes in V with preorder in the same interval share an identical entry edge on CT(λ). In order to construct the entry edge partition, for each edge (u, u') on CT(λ), we label (u, u') with an interval according to the following definition.

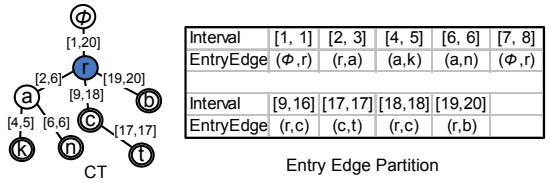
DEFINITION 4. **(Labeled Compact Tree)** *Given a tree T, a node v on T has an interval $[s_v, t_v]$ where s_v is the preorder label of v on T and t_v is the maximum preorder label for all nodes in the subtree rooted at v. Given a compact tree* CT(λ)*, for any edge (u, u') on* CT(λ)*, let the branching node of (u, u') be the first node along the path from u to u' on T, and denote it as u_b. We label edge (u, u') with the interval of u_b.*

The label of every edge on a compact tree CT(λ) can be computed easily when constructing CT(λ). Given any node v on a tree T and an edge (u, u') on a compact tree CT(λ), denote the branching node of (u, u') as u_b, then v is in the subtree rooted at u_b if and only if the preorder label of v on T is in the interval of u_b, which is identical with the label of edge (u, u'). For ease of presentation, for each labeled compact tree CT(λ), we add a virtual root ϕ and an edge from ϕ to the original root of CT(λ). We use the following example to illustrate the labeled compact tree.

Example 8: For the tree T shown in Fig. 3, we mark the preorder and the interval of each node on the tree. For the node h, its interval is $[10, 18]$ because the preorder of h on T is 10 and the maximum preorder for all nodes on the subtree rooted at h is 18. The labeled compact tree CT(λ) for keyword λ is shown on the left part of Fig. 6. For the edge (r, c) on CT(λ), its branching node is d because d is the first node along the path (r, d, h, e, m, c) on T. The label of edge (r, c) is the interval of node d, which is $[9, 18]$. □

For a compact tree CT(λ) of tree T and a keyword λ, suppose (u, u') on CT(λ) is an entry edge of a node v on tree T, i.e., $\mathsf{EE}_\lambda(v) = (u, u')$. The preorder of v is in the interval of (u, u'), because the interval of (u, u') contains all nodes under the subtree rooted at the branching node of (u, u'). Based on such an observation, by excluding the intervals of all edges under the subtree rooted at u' in CT(λ) from the interval of (u, u'), nodes with preorder in the remaining intervals will use (u, u') as the entry edge. For example, in the compact tree CT(λ) shown in Fig. 6, the edge (ϕ, r) has an interval $[1, 20]$. r has three branches with intervals $[2, 6]$, $[9, 18]$ and $[19, 20]$ respectively. By excluding the three intervals from $[1, 20]$, two intervals $[1, 1]$ and $[7, 8]$ are left. Thus nodes with preorder in either of the two intervals $[1, 1]$ and $[7, 8]$ share the same entry edge (ϕ, r). For edge (r, c) with interval $[9, 18]$, by excluding interval $[17, 17]$ of the only branch of c, nodes with preorder in either of the two intervals $[9, 16]$ and $[18, 18]$ share the same entry edge (r, c).

Algorithm 4 shows the construction of the entry edge partition EEP(λ) on CT(λ) for a keyword λ. After initializing EEP(λ) (line 2), the main operation is a recursive procedure partition (line 3),



Interval	[1, 1]	[2, 3]	[4, 5]	[6, 6]	[7, 8]
EntryEdge	(φ,r)	(r,a)	(a,k)	(a,n)	(φ,r)

Interval	[9,16]	[17,17]	[18,18]	[19,20]	
EntryEdge	(r,c)	(c,t)	(r,c)	(r,b)	

Entry Edge Partition

Figure 6: Labeled Compact Tree and Entry Edge Partition

to partition the interval $[1, |V|]$ to several disjoint intervals. Each entry in EEP(λ) is in the form of $([s, t], (u, u'))$ denoting that nodes with the preorder label in the interval $[s, t]$ share the same entry edge (u, u'). For an edge (u, u') with interval $[s, t]$, the procedure processes every child node u'' of u' on CT(λ) in increasing preorder of u'' (line 6). For each edge (u', u'') with interval $[s', t']$, the interval $[s, t]$ is partitioned into three parts: $[s, s' - 1]$, $[s', t']$ and $[t' + 1, t]$. The first part is added to EEP(λ) with the entry edge (u, u') if it is not empty (line 8). The second part is processed recursively for edge (u', u'') (line 9), and the third part is left to be further partitioned by other child nodes of u' by simply setting s to be $t' + 1$ (line 10). After processing all child nodes of u', if $[s, t]$ is still not empty, we add $[s, t]$ to EEP(λ) with the entry edge (u, u') (line 11).

The time complexity of Algorithm 4 is $O(|V(\mathsf{CT}(\lambda))|)$ since every node on CT(λ) is visited once. For each edge (u, u') on CT(λ), at most two intervals are added into EEP(λ). One is added before invoking partition for edge (u, u') (line 8) and the other is added at the end of partition for (u, u') (line 11). Thus the total number of intervals in EEP(λ) is no more than $2 \times |V(\mathsf{CT}(\lambda))|$.

Example 9: For the labeled compact tree CT(λ) shown in Fig. 6, when invoking partition(EEP(λ), $[1, 20]$, (ϕ, r), CT(λ)), we process the three child nodes a, c, b of r in order. We first process edge (r, a) with interval $[2, 6]$, which divides the interval $[1, 20]$ into three parts: $[1, 1]$, $[2, 6]$, and $[7, 20]$. $[1, 1]$ is added into EEP(λ) with the entry edge (ϕ, r). $[2, 6]$ is processed recursively by invoking partition(EEP(λ), $[2, 6]$, (r, a), CT(λ)), and $[7, 20]$ is processed by the other two child nodes c and b similarly. EEP(λ) is shown on the right part of Fig. 6. □

THEOREM 2. *For a tree $T(V, E)$ with the compact trees for all keywords constructed, the entry edge partition* EEP(λ) *for all keywords can be constructed in $O(|\mathsf{doc}(V)|)$ time and stored in $O(|\mathsf{doc}(V)|)$ space.*

5.3 Construction of Candidate List

Given a compact tree CT(λ) for a tree T and a keyword λ, we need to compute the candidate list $\mathsf{cand}_\lambda(v)$ for every node v on CT(λ). Since CT(λ) keeps the structural information of all keyword nodes in T, it is sufficient to search only on CT(λ) to calculate $\mathsf{cand}_\lambda(v)$. A simple solution is to compute each $\mathsf{cand}_\lambda(v)$ separately on CT(λ). This approach may take $O(|V(\mathsf{CT}(\lambda))|)$ time to calculate $\mathsf{cand}_\lambda(v)$ for a node v, thus $O(|V(\mathsf{CT}(\lambda))|^2)$ time to compute all candidate lists in CT(λ) for one keyword λ, which is too slow.

In order to save the computational cost, we design a novel method to update the candidate list of a node using those of its nearby nodes on the tree CT(λ). Note that in CT(λ), the path between two nodes u, v is unique: from node u to the lowest common ancestor of u and v, LCA(u, v), and then from LCA(u, v) to v. Based on this observation, we can follow the path to propagate the candidate list on u to v. Using this idea, we just need to traverse the tree CT(λ) twice to build the candidate lists for all nodes on CT(λ). The first traversal on CT(λ) is a bottom-up one, such that the candidate list on each node is propagated to all its ancestors on CT(λ). The second traversal on CT(λ) is a top-down one, such that the candidate list on each node is further propagated to all its descendants.

Algorithm 5: cand-construct (T,$\mathsf{CT}(\lambda)$, $\overline{k}$)

Input: A tree T, a compact tree $\mathsf{CT}(\lambda)$, and the upper bound of k, $\overline{k}$.
Output: $\mathsf{cand}_\lambda(v)$ for each v on $\mathsf{CT}(\lambda)$.

1 $\mathsf{cand}_\lambda(v) \leftarrow \emptyset$ for each node v on $\mathsf{CT}(\lambda)$;
2 $\mathsf{cand}_\lambda(v) \leftarrow \{v : 0\}$ for each node v on $\mathsf{CT}(\lambda)$ that contains λ;
3 **foreach** v *on* $\mathsf{CT}(\lambda)$ *in a bottom-up fashion* **do**
4 $u \leftarrow$ the parent node of v on $\mathsf{CT}(\lambda)$;
5 $\mathsf{cand}_\lambda(u) \leftarrow \mathsf{cand}_\lambda(u) \otimes_{\overline{k}} (\mathsf{cand}_\lambda(v) \oplus \mathsf{dist}(u, v))$;
6 **foreach** v *on* $\mathsf{CT}(\lambda)$ *in a top-down fashion* **do**
7 $u \leftarrow$ the parent node of v on $\mathsf{CT}(\lambda)$;
8 $\mathsf{cand}_\lambda(v) \leftarrow \mathsf{cand}_\lambda(v) \otimes_{\overline{k}} (\mathsf{cand}_\lambda(u) \oplus \mathsf{dist}(u, v))$;

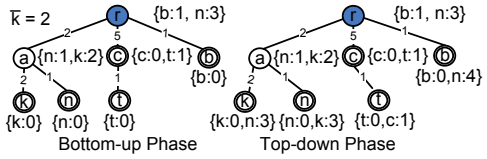


Figure 7: Constructing Candidate Lists

Algorithm 5 shows the construction of the candidate lists on $\mathsf{CT}(\lambda)$. We first initialize the candidate list for each keyword node to be the node itself and initialize the candidate list for each non-keyword node to be $\emptyset$ (line 1-2). We then traverse $\mathsf{CT}(\lambda)$ in a bottom-up fashion, e.g., using postorder traversal. For each node v traversed, we merge $\mathsf{cand}_\lambda(v)$ into that of its parent node u by adding a distance $\mathsf{dist}(u, v)$ to the list $\mathsf{cand}_\lambda(v)$ (line 3-5). At last, we traverse $\mathsf{CT}(\lambda)$ in a top-down fashion, e.g., using preorder traversal. For each node v traversed, we merge the list of v's parent node u, $\mathsf{cand}_\lambda(u)$, into that of v by adding a distance $\mathsf{dist}(u, v)$ to the list $\mathsf{cand}_\lambda(u)$ (line 6-8). Since the $\otimes_{\overline{k}}$ operator takes $O(\overline{k})$ time, the time complexity of Algorithm 5 is $O(\overline{k} \cdot |V(\mathsf{CT}(\lambda))|)$ using $O(\overline{k} \cdot |V(\mathsf{CT}(\lambda))|)$ space.

Example 10: Fig. 7 shows the candidate lists after the bottom-up phase and the top-down phase for the compact tree $\mathsf{CT}(\lambda)$ shown on the left part of Fig. 4. Initially, the candidate list for t is $\{t : 0\}$ and the candidate list for c is $\{c : 0\}$. Since c is a parent node of t, in the bottom-up phase, the list of t is propagated and merged into that of c by adding a distance $\mathsf{dist}(c, t) = 1$, thus $\mathsf{cand}_\lambda(c) = \{c : 0, t : 1\}$ after the bottom-up phase. In the top-down phase, the list of c is propagated and merged into that of t, thus $\mathsf{cand}_\lambda(t) = \{t : 0, c : 1\}$ after the top-down phase. □

THEOREM 3. *Given a tree T, an upper bound of k, $\overline{k}$, and $\mathsf{CT}(\lambda)$ for all keywords λ, the candidate lists $\mathsf{cand}_\lambda(v)$ for all keywords λ and all nodes v on $\mathsf{CT}(\lambda)$ can be constructed in $O(\overline{k} \cdot |\mathsf{doc}(V)|)$ time and stored in $O(\overline{k} \cdot |\mathsf{doc}(V)|)$ space.*

6. K-NK ON A TREE FOR A LARGE K

Algorithm 1 can only process a k-NK query $Q = (q, \lambda, k)$ with a bounded k, i.e., $k \leq \overline{k}$, on a tree T. If k can be arbitrarily large, the index size cannot be bounded. In this section, we will remove the restriction on k and introduce an algorithm to handle a k-NK query for an arbitrary k, with an index size independent of k.

6.1 A Basic Pivot Approach

Recall that for a node u that contains keyword λ and an arbitrary node v in a tree T, the path from v to u is unique on T, and can be divided into two segments: the first segment is from v to their lowest common ancestor $\mathsf{LCA}(u, v)$, and the second segment is from $\mathsf{LCA}(u, v)$ to u. Our basic idea is to compute the first segment online and precompute the results regarding the second segment offline. Thus, in the precomputing phase, instead of propagating a keyword node u to all nodes in T to update their candidate lists, we just need to propagate u to its ancestors in T. In the query processing phase, we do not search the whole tree to get the answer for a query, but instead, we just need to merge the precomputed candidates along the path from the query node to the root node of the tree T. Using this method, the size of the index to keep the candidate nodes can be largely reduced at the expense of longer query time.

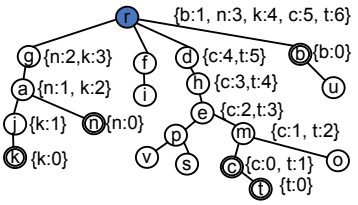


Figure 8: Basic Pivot Approach

We use $\mathsf{depth}(T)$ to denote the depth of tree T, and $\mathsf{depth}(u, T)$ to denote the depth of node u on tree T. For any two nodes u and v on T, u is a pivot of v if and only if u is an ancestor of v on T. For each node v, we denote the set of pivots of v on T as $\mathsf{PV}(v, T)$. We have $|\mathsf{PV}(v, T)| = \mathsf{depth}(v, T)$. Given a keyword λ, for each node u on tree T, we use the candidate list $\mathsf{cand}_\lambda(u)$ to denote the set of nodes that contain keyword λ on the subtree rooted at u on tree T, sorted in nondecreasing order of their distances to u. The candidate list is in the form of $\mathsf{cand}_\lambda(u) = \{u_1 : \mathsf{dist}_T(u, u_1), u_2 : \mathsf{dist}_T(u, u_2), \cdots\}$ where $\mathsf{dist}_T(u, u_1) \leq \mathsf{dist}_T(u, u_2) \leq \cdots$. In order to handle an arbitrary k, the size of $\mathsf{cand}_\lambda(u)$ is not bounded by any predefined $\overline{k}$. Clearly, a node $v \in \mathsf{cand}_\lambda(u)$ if and only if v contains keyword λ and $u \in \mathsf{PV}(v, T)$. In other words, a keyword node v only appears in the candidate lists of its pivots. As a result, for any keyword λ, the total size of all candidate lists for λ is $\sum_{v \in V_\lambda} |\mathsf{PV}(v, T)| = \sum_{v \in V_\lambda} \mathsf{depth}(v, T)$. We use the following example to illustrate the pivot based approach.

Example 11: Fig. 8 shows a tree T with $\mathsf{depth}(T) = 6$. For keyword λ, the nodes that contain λ are marked with bold circles. For every node v, we create a candidate list $\mathsf{cand}_\lambda(v)$ that contains all keyword nodes in its subtree, sorted in nondecreasing distances to v. For example, $\mathsf{cand}_\lambda(g) = \{n : 2, k : 3\}$ means there are two keyword nodes n and k in the subtree rooted at g with distances 2 and 3 to g respectively. For node p, $\mathsf{PV}(p, T) = \{r, d, h, e\}$. For a k-NK query $Q = (d, \lambda, 3)$, the path from d to the root r contains two nodes d and r. We merge the lists $\mathsf{cand}_\lambda(d)$ and $\mathsf{cand}_\lambda(r)$ by adding a distance $\mathsf{dist}(r, d) = 1$ to all elements in $\mathsf{cand}_\lambda(r)$. The final answer for Q is $\{b : 2, c : 4, n : 4\}$. □

6.2 Pivot Approach with Tree Balancing

The problem is not perfectly solved using the basic pivot approach above. The reasons are twofold. First, in the precomputing phase, the index size for each keyword λ is $\sum_{v \in V_\lambda} \mathsf{depth}(v, T)$, which can be large if $\mathsf{depth}(v, T)$ is large. Second, when processing a query $Q = (q, \lambda, k)$, we need to traverse all nodes from the query node q to the root of T. This is also costly if $\mathsf{depth}(q, T)$ is large. Thus the key to optimizing both index space and query time is to reduce the average depth of nodes on the tree. A simple solution is to rotate the tree T to find a proper root such that the average depth of nodes is minimized. However, such an approach cannot essentially solve the problem, as illustrated by the following example. Let $T(V, E)$ be a chain of $2n+1$ nodes where every node contains keyword λ. The best way is to select the middle node on the chain as the root to minimize the average depth of nodes. The total index size is $\sum_{v \in V_\lambda} \mathsf{depth}(v, T) = \sum_{v \in V(T)} \mathsf{depth}(v, T) = n(n-1)$, which is $O(n^2)$. Furthermore, we need to traverse n nodes to answer a query when the query node q is at one end of the chain, leading to $O(n)$ query time. This example shows that both the index space and query processing can still be very costly, even though we rotate the tree.

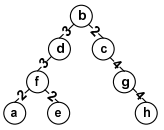
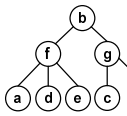
Original Tree T	$\mathsf{DT}(T)$	v	$\mathsf{PV}(v, \mathsf{DT}(T))$
		a	$\{b:8, f:2\}$
		d	$\{b:3, f:3\}$
		e	$\{b:8, f:2\}$
		c	$\{b:2, g:4\}$
		h	$\{b:10, g:4\}$
		f	$\{b:6\}$
		g	$\{b:6\}$

Figure 9: Distance Preserving Balanced Tree

In order to reduce the average depth of nodes to optimize both index space and query processing time, we introduce a new structure called distance preserving balanced tree for $T(V, E)$, denoted as $\mathsf{DT}(T)$. Generally speaking, $\mathsf{DT}(T)$ preserves all distance information for any node pair on T and the height of $\mathsf{DT}(T)$ is at most $\log_2 |V|$. The formal definition of $\mathsf{DT}(T)$ is as follows.

DEFINITION 5. **(Distance Preserving Balanced Tree)** *Given a tree $T(V, E)$ with a positive weight on each edge, a Distance Preserving Balanced Tree of T, denoted as $\mathsf{DT}(T)$, is an unweighted tree with the following three properties.*

P_1: $V(\mathsf{DT}(T)) = V(T)$.

P_2: $\mathsf{depth}(\mathsf{DT}(T)) \leq \log_2 |V|$.

P_3: For any two nodes u and v, let the lowest common ancestor of u and v on $\mathsf{DT}(T)$ be $o = \mathsf{LCA}_{\mathsf{DT}(T)}(u, v)$. The following equation always holds: $\mathsf{dist}_T(u, v) = \mathsf{dist}_T(u, o) + \mathsf{dist}_T(v, o)$.

Note that $\mathsf{DT}(T)$ is unweighted and the distances $\mathsf{dist}_T(u, v)$, $\mathsf{dist}_T(u, o)$ and $\mathsf{dist}_T(v, o)$ in P_3 are calculated on the original tree T, but not $\mathsf{DT}(T)$. The lowest common ancestor $\mathsf{LCA}_{\mathsf{DT}(T)}(u, v)$ is not necessarily the ancestor of u or v on the original tree T. Based on P_3, we can also divide our algorithm into two phases using $\mathsf{DT}(T)$. In the preprocessing phase, for each keyword λ, and each node v that contains keyword λ, we propagate v into the candidate lists of its pivots on $\mathsf{DT}(T)$. In the query processing phase, we traverse from the query node q to the root node on $\mathsf{DT}(T)$. Using the balanced tree $\mathsf{DT}(T)$, the total size of the candidate lists for a keyword λ is bounded by $\sum_{v \in V_\lambda} \mathsf{depth}(v, \mathsf{DT}(T)) \leq \sum_{v \in V_\lambda} \log_2 |V|$, and the total size for all keywords is bounded by $O(|\mathsf{doc}(V)| \cdot \log |V|)$. For processing a query, we need to traverse at most $\log_2 |V| + 1$ nodes on the path from the query node to the root of $\mathsf{DT}(T)$.

Example 12: A tree T with $\mathsf{depth}(T) = 3$ and a distance preserving balanced tree of T, $\mathsf{DT}(T)$ with $\mathsf{depth}(\mathsf{DT}(T)) = 2$ are shown in Fig. 9. The weight of each edge is marked on T. Edge (b, d) is on T but not on $\mathsf{DT}(T)$, and edge (b, f) is on $\mathsf{DT}(T)$ but not on T. For two nodes a and d, $\mathsf{LCA}_{\mathsf{DT}(T)}(a, d) = f$, thus $\mathsf{dist}_T(a, d) = \mathsf{dist}_T(a, f) + \mathsf{dist}_T(d, f) = 2 + 3 = 5$. Note that f is not an ancestor of d on the original tree T. $\mathsf{PV}(v, \mathsf{DT}(T))$ for each node v in $\mathsf{DT}(T)$ is listed on the right part of Fig. 9. □

Here we introduce our algorithm of processing a k-NK query on a tree T using $\mathsf{DT}(T)$, and in the next subsection, we will show that $\mathsf{DT}(T)$ always exists for any tree T. We will also describe how to construct $\mathsf{DT}(T)$ for a tree T and how to compute all candidate lists $\mathsf{cand}_\lambda(v)$ for all keywords λ and all nodes v on the tree $\mathsf{DT}(T)$.

Query Processing: Given a tree T and $\mathsf{DT}(T)$, Algorithm 6 shows how to process a query $Q = (q, \lambda, k)$. We traverse all nodes on the path from q to the root of $\mathsf{DT}(T)$, which is $\mathsf{PV}(q, \mathsf{DT}(T)) \bigcup \{q\}$ (line 2). For each traversed node v, we add $\mathsf{dist}_T(q, v)$ to all elements in $\mathsf{cand}_\lambda(v)$ and then merge the list into the current result R, since we need to first go from node q to node v (the first segment), and then go from v to the keyword nodes in $\mathsf{cand}_\lambda(v)$ (the second segment). Note that the time complexity of the $\oplus$ operator in line 3 is $O(|\mathsf{cand}_\lambda(v)|)$. However, by combining $\oplus$ with $\otimes_k$, it is easy to reduce the time complexity of line 3 to $O(k)$.

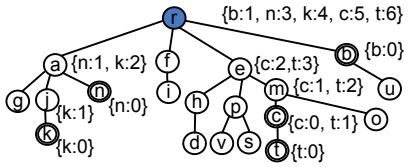


Figure 10: Pivot Approach with Tree Balancing

Algorithm 6: tree-pivot (Q,T)

Input: A k-NK query $Q = (q, \lambda, k)$, and a tree T.
Output: Answer for Q on T.
1 $R \leftarrow \emptyset$;
2 **foreach** $v \in \mathsf{PV}(q, \mathsf{DT}(T)) \bigcup \{q\}$ **do**
3 $\quad R \leftarrow R \otimes_k (\mathsf{cand}_\lambda(v) \oplus \mathsf{dist}_T(q, v))$;
4 **return** R;

Example 13: Fig. 10 shows a distance preserving balanced tree $\mathsf{DT}(T)$ for the tree T shown in Fig. 8, with depth 4. For keyword λ, the nodes that contain λ are marked with bold circles in Fig. 10. For a query $Q = (e, \lambda, 3)$, we just need to merge 2 candidate lists $\mathsf{cand}_\lambda(e)$ and $\mathsf{cand}_\lambda(r)$ by adding a distance $\mathsf{dist}_T(e, r) = 3$ to all elements in $\mathsf{cand}_\lambda(r)$. However, if we use the basic pivot approach on the original tree T without tree balancing, we need to merge 4 candidate lists for nodes e, h, d and r respectively. The answer for Q is $\{c:2, t:3, b:4\}$. □

THEOREM 4. *The time complexity for answering a k-NK query on a tree $T(V, E)$ using Algorithm 6 is $O(k \cdot \log |V|)$.*

6.3 Index Construction

Given a tree T, in order to answer a query $Q = (q, \lambda, k)$ using Algorithm 6, we need to build two indexes. The first index is the distance preserving balanced tree $\mathsf{DT}(T)$ for T and the second index is the candidate list $\mathsf{cand}_\lambda(v)$ for each keyword λ and each node v on $\mathsf{DT}(T)$. We introduce them separately in the following.

Constructing $\mathsf{DT}(T)$: Before introducing how to construct a tree $\mathsf{DT}(T)$ to satisfy the three properties in Definition 5, we first present an approach to constructing a tree T' from T, which satisfies properties P_1 and P_3. In other words, T' is distance preserving but not necessarily balanced. Let the initial T' be T. We change T' by performing the following steps.

(1) Randomly select a node r on T' as the new root and rotate T' accordingly.

(2) For each direct subtree T'_c of r on T', perform steps (1) and (2) on T'_c recursively.

Clearly, after steps (1) and (2), T' may not be isomorphic to T. We have the following two observations on T'. O_1: After performing step (1) on T', two nodes u and v are in different direct subtrees of r if and only if $\mathsf{LCA}_{T'}(u, v) = r$. Such a property also holds after performing step (2) on T' because step (2) only changes the structure within a subtree of r. O_2: Since the structure of T' is not changed after step (1), we have $\mathsf{dist}_T(u, v) = \mathsf{dist}_T(u, r) + \mathsf{dist}_T(v, r)$ on the original tree T. From O_1 and O_2, we have $\mathsf{dist}_T(u, v) = \mathsf{dist}_T(u, \mathsf{LCA}_{T'}(u, v)) + \mathsf{dist}_T(v, \mathsf{LCA}_{T'}(u, v))$ after step (2) on T'. Such a property also holds for any subtree of T' because it is processed using steps (1) and (2) recursively. As a result, T' satisfies property P_3.

Our $\mathsf{DT}(T)$ is constructed in a similar way as T'. In order to construct a balanced tree, in step (1), the root node r should be selected more carefully, instead of random selection. In our method, we select a *median node* to be the root node in step (1), which is defined as follows.

DEFINITION 6. **(Median Node)** *Given a tree T, the Median Node of T is a node r on T such that when using r as the root of T, for each direct subtree T_c of r on T, $|V(T_c)| \leq \frac{|V(T)|}{2}$ holds.*

Algorithm 7: DT-construct (T)

Input: A tree T.
Output: A distance preserving balanced tree $\mathsf{DT}(T)$.
1 $r \leftarrow$ the median node of T ;
2 rotate T with r as the root;
3 $\mathsf{DT}(T) \leftarrow$ a tree with a single node r;
4 **foreach** *direct subtree* T_i *of* r *in* T **do**
5 $\quad \mathsf{DT}(T_i) \leftarrow$ DT-construct(T_i);
6 $\quad$ add $\mathsf{DT}(T_i)$ as a subtree of r in $\mathsf{DT}(T)$;
7 **return** $\mathsf{DT}(T)$;

The median node r is used to balance the size of each direct subtree of T when using r as the root of T, as a direct subtree of r in T contains at most half of the nodes in T. Clearly, if a median node always exists for any tree, we can select a median node of tree T as the root and recursively do this for each direct subtree of the root. In this way we can construct a tree T' with $\mathsf{depth}(T') \leq \log_2 |V(T)|$. The following lemma shows that the median node always exists on any tree T, and also gives a method to find the median node of T.

LEMMA 1. *Given a tree T, the median node of T is the node r, such that the subtree rooted at r contains more than $\frac{|V(T)|}{2}$ nodes and $\mathsf{depth}(r, T)$ is the maximum.*

According to Lemma 1, the median node r is unique on T. Otherwise if there are two such nodes with the same maximum depth, the size of the tree will be larger than $|V(T)|$. Given a tree T, we can easily find the median node of T using time $O(|V(T)|)$ by traversing each node in T only once.

Algorithm 7 shows how to construct $\mathsf{DT}(T)$ for a tree T. Specifically, given a tree T, we first find the median node r of T as the new root and then rotate T accordingly (line 1-2). The median node r is also the root of $\mathsf{DT}(T)$ (line 3). For each direct subtree T_i of r in T, we create $\mathsf{DT}(T_i)$ recursively and add $\mathsf{DT}(T_i)$ as a subtree of $\mathsf{DT}(T)$ (line 4-6).

Example 14: For the tree T shown in Fig. 8, $\mathsf{DT}(T)$ is shown in Fig. 10. $\mathsf{DT}(T)$ is constructed as follows. Since r is the median node of T, the root of $\mathsf{DT}(T)$ is r. For the first subtree under r in T, its median node is a, thus the first subtree under r in $\mathsf{DT}(T)$ is rooted at a. All other nodes in $\mathsf{DT}(T)$ are constructed similarly. We have $\mathsf{depth}(\mathsf{DT}(T)) = 4 \leq \log_2 |V(T)| = \log_2 20$. □

THEOREM 5. *Given a tree $T(V, E)$, Algorithm 7 constructs a distance preserving balanced tree $\mathsf{DT}(T)$ for T using $O(|V| \cdot \log |V|)$ time and $O(|V|)$ space.*

Constructing $\mathsf{cand}_\lambda(v)$: For a tree $T(V, E)$, given $\mathsf{DT}(T)$, the algorithm for constructing the candidate list $\mathsf{cand}_\lambda(v)$ for each node v and each keyword λ is quite simple. For each node v, we propagate its keyword information to all its pivots in $\mathsf{DT}(T)$. Our algorithm is shown in Algorithm 8. We first initialize every candidate list to be $\emptyset$ (line 1). Then we traverse each node v in $\mathsf{DT}(T)$ and each keyword λ that is contained in node v (line 2-3). For each pivot p of v as well as v itself, we calculate $\mathsf{dist}_T(p, v)$ on the original tree T, and add the element $v : \mathsf{dist}_T(p, v)$ to the candidate list $\mathsf{cand}_\lambda(p)$ (line 4-5). After all candidate lists are created, we sort the elements in every candidate list in nondecreasing order of the distances. The time complexity for line 2-5 is $O(|\mathsf{doc}(V)| \cdot \log |V|)$ since each keyword is propagated into at most $\log |V|$ candidate lists in $\mathsf{DT}(T)$. For line 6-7, we need $O(|\mathsf{doc}(V)| \cdot \log^2 |V|)$ time to sort all candidate lists in $\mathsf{DT}(T)$.

THEOREM 6. *For a tree T, Algorithm 8 computes the candidate lists $\mathsf{cand}_\lambda(v)$ for all nodes v and all keywords λ on $\mathsf{DT}(T)$ using $O(|\mathsf{doc}(V)| \cdot \log^2 |V|)$ time and $O(|\mathsf{doc}(V)| \cdot \log |V|)$ space.*

Algorithm 8: cand-construct (T,$\mathsf{DT}(T)$)

Input: A tree T, a distance preserving balanced tree $\mathsf{DT}(T)$.
Output: $\mathsf{cand}_\lambda(v)$ for each v on $\mathsf{DT}(T)$ and each keyword λ.
1 $\mathsf{cand}_\lambda(v) \leftarrow \emptyset$ for each node v on $\mathsf{DT}(T)$ and each keyword λ;
2 **foreach** $v \in V(\mathsf{DT}(T))$ **do**
3 $\quad$ **foreach** $\lambda \in \mathsf{doc}(v)$ **do**
4 $\quad\quad$ **foreach** $p \in \mathsf{PV}(v, \mathsf{DT}(T)) \bigcup \{v\}$ **do**
5 $\quad\quad\quad \mathsf{cand}_\lambda(p) \leftarrow \mathsf{cand}_\lambda(p) \bigcup \{v : \mathsf{dist}_T(p, v)\}$;
6 **foreach** $v \in V(\mathsf{DT}(T))$ *and keyword* λ **do**
7 $\quad$ sort elements in $\mathsf{cand}_\lambda(v)$ in nondecreasing order of distances;

Algorithm 9: graph-knk (G,Q)

Input: A graph $G(V, E)$ and a k-NK query $Q = (q, \lambda, k)$.
Output: The answer for Q on G.
1 $R \leftarrow \emptyset$;
2 **foreach** *Distance Oracle* $\mathcal{O}_i$ **do**
3 $\quad T_i \leftarrow$ shortest path tree for $\mathcal{O}_i$;
4 $\quad R \leftarrow R \otimes_k$ tree-knk(T_i, Q);
5 **return** R;

7. APPROXIMATE K-NK ON A GRAPH

In this section, we discuss how to answer a k-NK query $Q = (q, \lambda, k)$ on a graph G. We introduce two algorithms graph-boundk and graph-pivot for a bounded k and an arbitrary k respectively. We then propose a global storage technique to reduce the index size and query processing time. We also show how our approach can be extended to handle multiple keywords. Finally, we summarize the complexities of all algorithms introduced in this paper.

Query Processing: Our general idea for query processing on a graph is introduced in Section 4. Suppose we have computed $r = O(\log |V|)$ distance oracles $\mathcal{O}_1, \mathcal{O}_2, \cdots, \mathcal{O}_r$ using the algorithm in [20]. Let the shortest path trees for the oracles be $T_1, T_2, \cdots, T_r$ respectively. Algorithm 9 shows our framework for answering Q on G. The algorithm simply enumerates all shortest path trees and answers the k-NK query using a tree based approach, denoted as tree-knk, on each shortest path tree T_i, and merges all the results using the $\otimes_k$ operator (line 4). Since we have two tree based solutions, namely, tree-boundk and tree-pivot, we have two corresponding algorithms on graphs, denoted as graph-boundk and graph-pivot, by instantiating tree-knk (line 4) to tree-boundk and tree-pivot respectively.

Global Storage: As discussed above, we have r shortest path trees $T_1, T_2, \cdots, T_r$. For a keyword λ and a node v, let $\mathsf{cand}^i_{v,\lambda}$ be the candidate list of v on tree T_i, $1 \leq i \leq r$. To answer a k-NK query $Q = (q, \lambda, k)$ on a graph, consider a case when the candidate lists of node v on two different trees T_i and T_j are both merged into the result, in the form of

$$R \leftarrow R \otimes_k (\mathsf{cand}^i_{v,\lambda} \oplus \mathsf{dist}_{T_i}(q, v)) \otimes_k (\mathsf{cand}^j_{v,\lambda} \oplus \mathsf{dist}_{T_j}(q, v)).$$

This expression can be generalized to the case of merging the candidate lists of node v on more than two trees. Instead of keeping a candidate list $\mathsf{cand}^i_{v,\lambda}$ for each tree T_i ($1 \leq i \leq r$) separately, we propose a technique called *global storage* which keeps a global candidate list of node v and keyword λ for all trees $T_1, T_2, \cdots, T_r$. Denote the global candidate list of node v and keyword λ as $\mathsf{cand}_{v,\lambda}$. It is computed by

$$\mathsf{cand}_{v,\lambda} = \mathsf{cand}^1_{v,\lambda} \otimes \mathsf{cand}^2_{v,\lambda} \otimes \cdots \otimes \mathsf{cand}^r_{v,\lambda}.$$

For a node v, a node $v' \in \mathsf{cand}_{v,\lambda}$ may appear in the candidate list $\mathsf{cand}^i_{v,\lambda}$ of multiple trees T_i, but will be stored at most once in the global candidate list $\mathsf{cand}_{v,\lambda}$. Therefore, the global storage technique can effectively reduce the index size, but it adds difficulty to query processing due to two reasons: (1) we need to add $\mathsf{dist}_{T_i}(q, v)$ to $\mathsf{cand}^i_{v,\lambda}$ using the $\oplus$ operator, i.e., $\mathsf{cand}^i_{v,\lambda} \oplus$

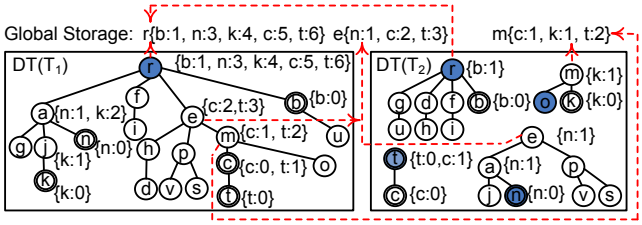


Figure 11: Global Storage Example for graph-pivot

$\mathsf{dist}_{T_i}(q,v)$, but $\mathsf{dist}_{T_i}(q,v)$ is query dependent, thus cannot be precomputed; (2) the global candidate list may provide a different result list from the one computed by Algorithm 9 without using global storage. In the following, we will show that the global candidate list can be used to answer k-NK queries without sacrificing the result quality. We first define the domination relationship between two candidate lists.

DEFINITION 7. *For two candidate lists* $R_1 = \{u_1 : d_{u_1}, u_2 : d_{u_2}, \cdots\}$ *and* $R_2 = \{v_1 : d_{v_1}, v_2 : d_{v_2}, \cdots\}$ *sorted in nondecreasing order of distances,* R_1 *is dominated by* R_2*, denoted as* $R_1 \geq R_2$*, if and only if* $|R_1| \leq |R_2|$ *and* $d_{u_i} \geq d_{v_i}$ *for all* $1 \leq i \leq |R_1|$*. Clearly, the domination relationship is transitive, i.e., if* $R_1 \geq R_2$ *and* $R_2 \geq R_3$*, then* $R_1 \geq R_3$*.*

To solve the first problem, we need to find a merge method that is independent of $\mathsf{dist}_{T_i}(q,v)$ and at the same time, can generate an answer that is no worse than the answer computed without global storage. The solution is expressed in Equ. 1. For any two candidate lists $\mathsf{cand}^i_{v,\lambda} \oplus \mathsf{dist}_{T_i}(q,v)$ and $\mathsf{cand}^j_{v,\lambda} \oplus \mathsf{dist}_{T_j}(q,v)$, using Equ. 1, we can generate a better result by merging $\mathsf{cand}^i_{v,\lambda}$ and $\mathsf{cand}^j_{v,\lambda}$ using $\otimes_k$ first, then taking distances $\mathsf{dist}_{T_i}(q,v)$ and $\mathsf{dist}_{T_j}(q,v)$ out and applying the minimum value of them. Clearly, $(\mathsf{cand}^i_{v,\lambda} \otimes_k \mathsf{cand}^j_{v,\lambda}) \oplus \min\{\mathsf{dist}_{T_i}(q,v), \mathsf{dist}_{T_j}(q,v)\}$ is a valid candidate list for query Q, because $\mathsf{cand}^i_{v,\lambda} \otimes_k \mathsf{cand}^j_{v,\lambda}$ is a candidate list for node v and $\min\{\mathsf{dist}_{T_i}(q,v), \mathsf{dist}_{T_j}(q,v)\}$ suggests a path from q to v in G.

$$(\mathsf{cand}^i_{v,\lambda} \oplus \mathsf{dist}_{T_i}(q,v)) \otimes_k (\mathsf{cand}^j_{v,\lambda} \oplus \mathsf{dist}_{T_j}(q,v)) \geq (\mathsf{cand}^i_{v,\lambda} \otimes_k \mathsf{cand}^j_{v,\lambda}) \oplus \min\{\mathsf{dist}_{T_i}(q,v), \mathsf{dist}_{T_j}(q,v)\} \quad (1)$$

The second problem can be solved if we prove that by merging more candidate lists using the $\otimes$ operator, the answer will not get worse. Consider a node $v' \in \mathsf{cand}_{v,\lambda}$, the merging operation finds the minimum distance between v' and v over multiple trees, which is a refined estimation of their distance on graph. We formulate such a situation using Equ. 2.

$$\mathsf{cand}^i_{v,\lambda} \geq \mathsf{cand}^i_{v,\lambda} \otimes \mathsf{cand}^j_{v,\lambda} \quad (2)$$

Equ. 1 and Equ. 2 also hold for multiple candidate lists. Therefore, we show that using global storage will not sacrifice the result quality. More importantly, global storage can effectively reduce the index size and query processing time. It applies to both graph algorithms graph-boundk and graph-pivot. We use the following example to illustrate global storage.

Example 15: We take the graph-pivot algorithm as an example. Fig. 11 shows two trees $\mathsf{DT}(T_1)$ and $\mathsf{DT}(T_2)$ for the shortest path tree T_1 and T_2 shown in Fig. 5, with candidate list marked beside each node for keyword λ. Using global storage, for the same node on different trees, we merge all its candidate lists using $\otimes$ and only keep one global candidate list. The global candidate lists for nodes r, e and m are marked on the top of Fig. 11. For query $Q_1 = (p, \lambda, 2)$, without global storage, we need to merge three candidate lists, $\mathsf{cand}^1_{e,\lambda} \oplus \mathsf{dist}_{T_1}(p,e)$, $\mathsf{cand}^1_{r,\lambda} \oplus \mathsf{dist}_{T_1}(p,r)$ and $\mathsf{cand}^2_{r,\lambda} \oplus \mathsf{dist}_{T_2}(p,e)$. Using global storage, only two candidate lists $\mathsf{cand}_{e,\lambda} \oplus \min\{\mathsf{dist}_{T_1}(p,e), \mathsf{dist}_{T_2}(p,e)\}$, $\mathsf{cand}_{r,\lambda} \oplus \mathsf{dist}_{T_1}(p,r)$ need to be merged. For query $Q_2 = (h, \lambda, 2)$, without global storage, we get the result $R = \{c:3, t:4\}$. Using global storage, we can get a result $R' = \{n:2, c:3\}$ with $R \geq R'$. □

Table 1: Algorithm Complexities on Trees (T) and Graphs (G)

	boundk	pivot
Query Time (T)	$O(\log \lvert V_\lambda \rvert + k)$	$O(k \cdot \log \lvert V \rvert)$
Index Time (T)	$O(\overline{k} \cdot \lvert \mathsf{doc}(V) \rvert)$	$O(\lvert \mathsf{doc}(V) \rvert \cdot \log^2 \lvert V \rvert)$
Index Size (T)	$O(\overline{k} \cdot \lvert \mathsf{doc}(V) \rvert)$	$O(\lvert \mathsf{doc}(V) \rvert \cdot \log \lvert V \rvert)$
Query Time (G)	$O((\log \lvert V_\lambda \rvert + k) \cdot \log \lvert V \rvert)$	$O(k \cdot \log^2 \lvert V \rvert)$
Index Time (G)	$O(\overline{k} \cdot \lvert \mathsf{doc}(V) \rvert \cdot \log \lvert V \rvert)$	$O(\lvert \mathsf{doc}(V) \rvert \cdot \log^3 \lvert V \rvert)$
Index Size (G)	$O(\overline{k} \cdot \lvert \mathsf{doc}(V) \rvert \cdot \log \lvert V \rvert)$	$O(\lvert \mathsf{doc}(V) \rvert \cdot \log^2 \lvert V \rvert)$

Table 2: Dataset Statistics

	$\lvert V \rvert$	$\lvert E \rvert$	$\lvert \mathsf{doc}(V) \rvert$	keywords
DBLP	1,695,469	4,726,801	12,842,501	331,301
FLARN	1,070,376	1,356,399	6,966,665	2,730

Handling Multiple Keywords: We discuss how to extend our approach to handle a k-NK query of multiple keywords with AND (denoted as $\wedge$) and OR (denoted as $\vee$) semantics. Without loss of generality, we assume the format of a keyword expression is $(\lambda_{1,1} \wedge \lambda_{1,2} \cdots) \vee (\lambda_{2,1} \wedge \lambda_{2,2} \cdots) \vee \cdots$. It is easy to handle $\vee$, by answering each $\lambda_{i,1} \wedge \lambda_{i,2} \cdots$ separately and merging the results using the $\otimes$ operator. For handling $\lambda_{i,1} \wedge \lambda_{i,2} \cdots$, we select a keyword $\lambda_{i,j}$ from $\{\lambda_{i,1}, \lambda_{i,2}, \cdots\}$ with the least frequency $|V_{\lambda_{i,j}}|$ as the primary keyword and consider other keywords as filter keywords. We answer the query for the single keyword $\lambda_{i,j}$. Before merging each candidate list using the $\otimes$ operator, we remove the candidate nodes that do not contain one or more of the filter keywords from the candidate list. In this way, each element in the final answer satisfies the predicate specified in the keyword expression.

Comparison: Table 1 summarizes and compares the query time, index time and index size for boundk and pivot on trees and graphs. Here, the listed complexities of index time and index size are for all keywords in the tree/graph. boundk is faster than pivot in query processing on both trees and graphs. When k is small, the index time and index space for boundk are smaller than pivot on both trees and graphs. However, when k is large, the index time and index space for boundk are large, while the index time and index space of pivot are independent of k on both trees and graphs.

8. EXPERIMENTS

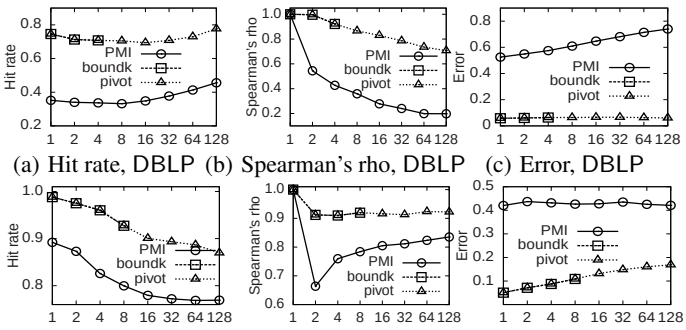
In this section, we report the performance of our methods boundk, pivot, and their global storage implementations boundk-gs and pivot-gs, with two baseline solutions BFS and PMI. BFS is a brute-force search that uses Dijkstra's algorithm to identify the nearest k keyword nodes, and PMI (Partitioned Multi-Indexing) [1] is the state-of-the-art approximate algorithm based on distance oracle [20]. For all the distance oracles involved we set the parameter $r = \log_2 |V|$. We implemented all methods in GNU C++, and conducted all experiments on a Windows machine with an Intel Xeon 2.7GHz CPU and 128GB memory. All methods run in main memory. A 32GB memory limit is set for index size.

Datasets and Queries. We use two real graphs, DBLP[3], and Florida road network FLARN[4], with statistics listed in Table 2.

DBLP includes 1,060,763 articles, 631,589 authors and 3,117 conferences/journals, all of which are treated as nodes. There is an edge between nodes u and v, if u is an author of article v, or u is an article published in conference/journal v. The keywords of an author node include first name and last name, the keywords of an article node include title words, editor, year, publisher, isbn, etc., and the keywords of a conference/journal node include association and name. A weight $(\log_2 \mathsf{deg}(u) + \log_2 \mathsf{deg}(v))$ is assigned to

[3]http://www.informatik.uni-trier.de/~ley/db

[4]http://www.dis.uniroma1.it/challenge9/download.shtml



(a) Hit rate, DBLP (b) Spearman's rho, DBLP (c) Error, DBLP

(d) Hit rate, FLARN (e) Spearman's rho,FLARN (f) Error, FLARN

Figure 12: Hit rate, Spearman's rho and Error by Varying k

edge (u, v), where $\deg(u)$ denotes the degree of node u. Compared with the unit edge weight setting, the numerical edge weights can effectively differentiate the weights of all edges in a graph. Thus for any k-NK query, this helps produce a ranking of top-k answer nodes with less ties in their distances as the ground truth, which is important for fair and unambiguous ranking quality evaluation.

In FLARN, a node represents an intersection or endpoint, an edge denotes a road segment, and the edge weight is the distance of the road segment. We obtained the keywords of nodes from the OpenStreetMap project[5] with a bounding box. However, only $7,172$ nodes out of $1,070,376$ have keywords. To address the keyword sparseness issue and better discriminate different methods, we assign a random number (between 0 and 4) of keywords to the nodes with no keyword. After this step, there are still $213,081$ nodes without any keyword in FLARN.

We remove stop words in DBLP and FLARN. For each dataset, we generate 500 k-NK queries in the form of $Q = (q, \lambda, k)$, where $q \in V$ is a randomly selected query node, and λ is a keyword randomly selected by following the keyword frequency distribution in the document collection. We test $k = 1, 2, \ldots, 128$.

Evaluation Metrics. We use six metrics for evaluation: *hit rate*, *Spearman's rho* [21], *error*, *query time*, *index time*, and *index size*. *Spearman's rho* measures the rank correlation between an approximate rank result and the ground truth. *Hit rate* and *error*, defined as follows, measure the quality of an approximate result. For a query $Q = (q, \lambda, k)$, denote the exact result as $R = \{u_1 : d_1, \ldots, u_k : d_k\}$ in nondecreasing order of their distances, and $\overline{d} = d_k$ as the upper bound distance of the result R. Denote an approximate result set as $R' = \{u'_1 : d'_1, \ldots, u'_k : d'_k\}$ in nondecreasing order of their distances. The hit rate is defined as:

$$\mathsf{hit}(R') = |\{i \in [1,k] | \mathsf{dist}(u'_i, q) \leq \overline{d}\}|/k$$

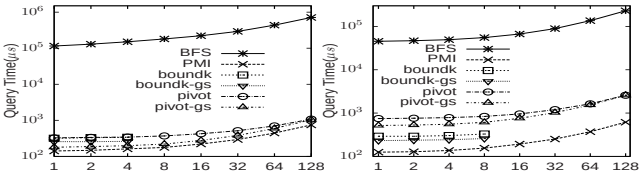
and the error is the average relative error of the estimated distances w.r.t. the ground truth:

$$\mathsf{err}(\mathrm{R}') = \sum_{1 \leq i \leq k} |d'_i/d_i - 1|/k$$

Hit rate, Spearman's rho and Error. Figures 12(a)–(c) show the hit rate, Spearman's rho, and error on DBLP respectively when we vary k. Our method pivot improves the hit rate of PMI by 96%, and improves Spearman's rho by 111% on average. The error of pivot is within 0.066 for all k values, demonstrating that the distance estimated by pivot is very close to the exact distance. Notably, pivot reduces the error of PMI by an order of magnitude, i.e., from 0.630 to 0.063 on average. Furthermore, the error of pivot does not increase with k, while that of PMI increases by 40% with the increase of k. Note when $k = 1$, Spearman's rho is constantly 1.

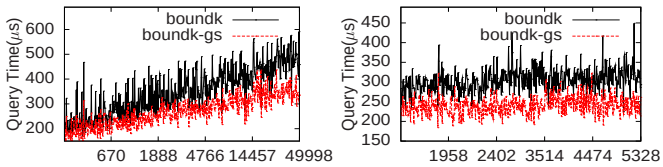
Figures 12(d)–(f) show the hit rate, Spearman's rho, and error on FLARN respectively. pivot improves both the average hit rate and

[5]http://wiki.openstreetmap.org/wiki/Main_Page



(a) DBLP (b) FLARN

Figure 13: Query Time in Microseconds by Varying k



(a) DBLP (b) FLARN

Figure 14: Query Time of boundk Varying Keyword Frequency

Spearman's rho of PMI by 14%. The error of pivot is below 0.168 for all k values and is 4 times smaller than that of PMI on average.

Note that the performance of BFS is omitted in Figure 12, as it returns the exact result. Furthermore, the result quality between using and not using global storage does not differ substantially, for the sake of clarity, the global storage methods boundk-gs and pivot-gs are also omitted in Figure 12. But we do observe that global storage technique improves the hit rate of boundk/pivot by 1.3% on DBLP and 0.7% on FLARN, and reduces the error by 6.7% on DBLP and 16.9% on FLARN on average. Given the memory limit of 32GB for index size, boundk can only support $k \leq 4$ on DBLP and $k \leq 8$ on FLARN in Figure 12 as its index size increases linearly with $\overline{k}$.

Query Time. Figure 13 shows the query time of different methods in log scale when we vary k. The query time of BFS is 10^5–10^6 microseconds, which is two to three orders of magnitude slower than the other methods.

Figure 13(a) shows the query time on DBLP. The query time of all methods increases with the increase of k. PMI is the most efficient. The query time of boundk, boundk-gs, pivot and pivot-gs is less than 2 times that of PMI, which is quite close. Global storage reduces the query time of boundk by 22% and that of pivot by 25%. Remarkably, each of our proposed approaches can report a result within 1 millisecond for all k values.

Figure 13(b) shows the query time on FLARN. We can observe that PMI is the fastest, closely followed by boundk and boundk-gs, whose query time is less than two times that of PMI and one third that of pivot for all k values. pivot and pivot-gs take a little longer as their query time depends on the tree depth which is large on FLARN. But their query time is within 3 milliseconds for $k = 128$, which is still quite efficient. Global storage helps reduce the query time of boundk by 20% and that of pivot by 15%.

Figure 14 further plots the query time of boundk and boundk-gs on the 500 k-NK queries in ascending order of the query keyword frequency in the graph. We set $k = 4$ in this experiment. For illustration, we also label a few query keyword frequencies on the x axis. The query time shows a sharper increasing trend on DBLP than FLARN, as the frequency difference between DBLP keywords is larger. These empirical results are consistent with the theoretical result, i.e., the query time complexity of boundk depends on $\log |V_\lambda|$, where $|V_\lambda|$ is the frequency of keyword λ.

Index Time and Index Size. Figure 15 shows the total index time (IT) and index size (IS) for indexing all keywords by different methods. We observe that the index time of pivot is 2.6 times that of PMI on DBLP, and 8.2 times on FLARN. The index construction time of pivot is longer on FLARN than on DBLP. This is because the complexity of pivot grows linearly with the tree depth,

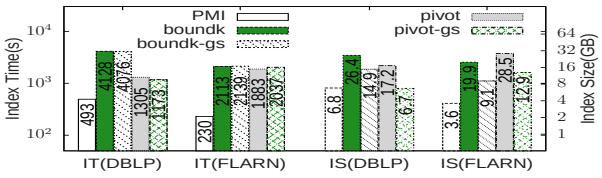


Figure 15: Index Time and Index Size

and the larger diameter of FLARN leads to a larger tree depth. All methods can finish the index construction for all keywords in a graph within 1.15 hours.

Given the memory limit of 32GB for index size, boundk can only support $k \leq 4$ on DBLP and $k \leq 8$ on FLARN, as its index size increases linearly with k. In contrast, pivot/pivot-gs have no such limitation. The index size of pivot is 2.5 times that of PMI on DBLP and 7.9 times on FLARN, due to the larger diameter of FLARN. By keeping a global candidate list and removing duplicate index items, global storage reduces the index size of pivot by 61% on DBLP and 55% on FLARN. It also reduces the index size of boundk by 44% on DBLP and 54% on FLARN. Remarkably, the index size of pivot-gs is 6.7GB on DBLP, which is even *smaller* than that of PMI (6.8GB). This result proves the superiority of global storage.

9. RELATED WORK

The most related work to our study include nearest keyword search on XML documents [22] and top-k nearest keyword search on graphs [1], both of which have been introduced in details in Section 3. In the sequel, we review the existing work on other topics related to our study.

Keyword search in a graph finds a substructure of the graph containing the query keywords. The answer substructure can be a tree [12, 3, 13, 8, 10, 9], a subgraph [16, 17] or a r-clique [14]. A survey on keyword search in databases and graphs can be found in [25]. Keyword search has substantial differences from the k-NK query studied in this paper. In terms of problem definition, keyword search looks for a network structure, the nodes in which jointly contain all the query keywords, whereas a k-NK query looks for k nearest answer nodes, each one of which contains all the query keywords. In terms of solution, keyword search performs BFS or Dijkstra's algorithm to find the answer networks, whereas our proposed solutions build an index structure based on distance oracles and compact trees for keywords. Therefore, our query time efficiency is much higher than BFS and Dijkstra's algorithm, which has also been confirmed in our experiments. [24] and [4] study *keyword routing* on a road network. Given a keyword set, a source and a target locations, the goal is to find the shortest path that passes through at least one matching object for each keyword.

Distance oracle is an approximate distance estimation technique. [23] is a seminal work on distance oracle that estimates distance with $2k-1$ stretch using an $O(|V|^{1+\frac{1}{k}})$ sized index. Hermelin et al. [11] adapt the distance oracle [23] to answer 1-NK queries with $4k-5$ stretch in $O(k)$ time using an $O(k|V|^{1+\frac{1}{k}})$ sized index. Our methods build on the distance oracle by Das Sarma et al. [20].

K nearest neighbor (k-NN) search has been extensively studied in spatial networks [15, 5, 6, 18, 19, 7]. [15] uses network Voronoi polygons to divide a graph into disjointed subsets for k-NN search. [5, 6] use R-tree to embed textual information on nodes, and augment a tree node with inverted index for spatial document within the MBR. [18] answers k-NN queries with a shortest path quadtree. [19] answers k-NN queries based on ε-approximated distance estimated by an index termed path-distance oracle of size $O(n \cdot \max(s^d, \frac{1}{\epsilon}^d))$ where d is the network dimension. [7] performs Dijkstra-like expansion from the query node. However the above approaches designed for spatial networks cannot apply to graphs without coordinates.

10. CONCLUSIONS

In this paper, we study top-k nearest keyword (k-NK) search on large graphs. We propose two exact k-NK algorithms on trees to handle a bounded k and an arbitrary k respectively. We extend tree based algorithms to graphs and propose a global storage technique to further reduce the index size and query time. We conducted extensive performance studies on real large graphs to demonstrate the effectiveness and efficiency of our algorithms.

Acknowledgments

This work is supported by the Hong Kong Research Grants Council (RGC) General Research Fund (GRF) Project No. CUHK 411211, 411310, 418512, and the Chinese University of Hong Kong Direct Grant No. 4055015.

11. REFERENCES

[1] B. Bahmani and A. Goel. Partitioned multi-indexing: Bringing order to social search. In *WWW*, pages 399–408, 2012.

[2] M. A. Bender and M. Farach-colton. The lca problem revisited. In *In Latin American Theoretical Informatics*, pages 88–94. Springer, 2000.

[3] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using banks. In *ICDE*, pages 431–440, 2002.

[4] X. Cao, L. Chen, G. Cong, and X. Xiao. Keyword-aware optimal route search. *PVLDB*, 5(11):1136–1147, 2012.

[5] Y.-Y. Chen, T. Suel, and A. Markowetz. Efficient query processing in geographic web search engines. In *SIGMOD*, pages 277–288, 2006.

[6] M. Christoforaki, J. He, C. Dimopoulos, A. Markowetz, and T. Suel. Text vs. space: Efficient geo-search query processing. In *CIKM*, pages 423–432, 2011.

[7] K. Deng, X. Zhou, H. T. Shen, S. W. Sadiq, and X. Li. Instance optimal query processing in spatial networks. *VLDB J.*, 18(3):675–693, 2009.

[8] B. Ding, J. X. Yu, S. Wang, L. Qin, X. Zhang, and X. Lin. Finding top-k min-cost connected trees in databases. In *ICDE*, pages 836–845, 2007.

[9] K. Golenberg, B. Kimelfeld, and Y. Sagiv. Keyword proximity search in complex data graphs. In *SIGMOD*, pages 927–940, 2008.

[10] H. He, H. Wang, J. Yang, and P. S. Yu. Blinks: Ranked keyword searches on graphs. In *SIGMOD*, pages 305–316, 2007.

[11] D. Hermelin, A. Levy, O. Weimann, and R. Yuster. Distance oracles for vertex-labeled graphs. In *ICALP (2)*, pages 490–501, 2011.

[12] V. Hristidis and Y. Papakonstantinou. Discover: Keyword search in relational databases. In *VLDB*, pages 670–681, 2002.

[13] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar. Bidirectional expansion for keyword search on graph databases. In *VLDB*, pages 505–516, 2005.

[14] M. Kargar and A. An. Keyword search in graphs: Finding r-cliques. *PVLDB*, 4(10):681–692, 2011.

[15] M. R. Kolahdouzan and C. Shahabi. Voronoi-based k nearest neighbor search for spatial network databases. In *VLDB*, pages 840–851, 2004.

[16] G. Li, B. C. Ooi, J. Feng, J. Wang, and L. Zhou. Ease: Efficient and adaptive keyword search on unstructured, semi-structured and structured data. In *SIGMOD*, pages 903–914, 2008.

[17] L. Qin, J. X. Yu, L. Chang, and Y. Tao. Querying communities in relational databases. In *ICDE*, pages 724–735, 2009.

[18] H. Samet, J. Sankaranarayanan, and H. Alborzi. Scalable network distance browsing in spatial databases. In *SIGMOD*, pages 43–54, 2008.

[19] J. Sankaranarayanan and H. Samet. Query processing using distance oracles for spatial networks. *IEEE Trans. Knowl. Data Eng.*, 22(8):1158–1175, 2010.

[20] A. D. Sarma, S. Gollapudi, M. Najork, and R. Panigrahy. A sketch-based distance oracle for web-scale graphs. In *WSDM*, pages 401–410, 2010.

[21] C. Spearman. The proof and measurement of association between two things. *Amer. J. Psychol.*, 15(1):72–101, 1904.

[22] Y. Tao, S. Papadopoulos, C. Sheng, and K. Stefanidis. Nearest keyword search in xml documents. In *SIGMOD*, pages 589–600, 2011.

[23] M. Thorup and U. Zwick. Approximate distance oracles. In *STOC*, pages 183–192, 2001.

[24] B. Yao, M. Tang, and F. Li. Multi-approximate-keyword routing in gis data. In *GIS*, pages 201–210, 2011.

[25] J. X. Yu, L. Qin, and L. Chang. Keyword search in relational databases: A survey. *IEEE Data Eng. Bull.*, 33(1):67–78, 2010.