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 $\lambda$, a k-NK query seeks $k$ nodes which contain $\lambda$ and are nearest to $q$. Different from a large body of research on $k$-nearest neighbor (k-NN) search on spatial networks \cite{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\footnote{https://www.facebook.com/about/graphsearch}. 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 kindergarten 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 \cite{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 \cite{20} which is a distance estimation technique. Given a k-NK query $Q = (q, \lambda, k)$, it returns $k$ nodes containing keyword $\lambda$ 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 \cite{20}, where $V$ is the set of nodes in the
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 $k$, i.e., $k \leq k$. 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 $\lambda$, and index size $O(k \cdot \log |V|)$, where $|\text{doc}(V)|$ is the total number of keywords on all the nodes in the graph.

(2) Next we remove the $k$ 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 $O(\log V \cdot |\text{doc}(V)|)$ 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 $\text{weight}(u, v)$. A path $p = (v_1, v_2, \ldots, v_l)$ 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 $G$, $\text{dist}(u, v)$, 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 $\text{doc}(v)$. The union of keywords for all nodes in $G$ is denoted as $\text{doc}(V)$. Note that $\text{doc}(V)$ is a multiset and $|\text{doc}(V)| = \sum_{v \in V} |\text{doc}(v)|$. We use $V_\lambda \subseteq V$ to denote the set of nodes carrying keyword $\lambda$ in $V$.

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

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 $\lambda$ 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 $\text{dist}(f, b) = 2$, $\text{dist}(f, n) = 4$, and $\text{dist}(f, k) = 5$.

### 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 $\text{wit}_\mathcal{O}(v)$, which is called the witness node of $v$ w.r.t. $\mathcal{O}$. If $v$ is a center node, $\text{wit}_\mathcal{O}(v) = v$. For each node $v \in V(G)$, the shortest distance from $v$ to its witness node, i.e., $\text{dist}(v, \text{wit}_\mathcal{O}(v))$, 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., $\text{wit}_\mathcal{O}(u) = \text{wit}_\mathcal{O}(v)$, then $\text{dist}(u, v)$. To answer a k-NK query $Q = (q, \lambda, k)$ on $G$, we first find the witness nodes $\text{wit}_\mathcal{O}(q)$ and $\text{wit}_\mathcal{O}(q)$ of the query node $q$. Then we find the $k$ nearest keyword nodes to $\text{wit}_\mathcal{O}(q)$ and $\text{wit}_\mathcal{O}(q)$, respectively, using compact index structures.
result w.r.t. $O_1$. There is one center node for any two nodes $u$ and $v$ in $G$ can be estimated as an upper bound $\text{dist}(u, v) = \min_{i \in I} O_1(u, v)$.

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

**Example 2:** Fig. 2 shows two distance oracles $O_1$ and $O_2$ for the graph shown in Fig. 1. There is one center node $r$ in $O_1$, and four center nodes $r$, $n_{18}$, $c$, and $t$ in $O_2$. The distance of nodes $j$ and $s$ is estimated as $\text{dist}(j, s) = \min_{i \in I} O_1(j, s), \text{dist}(j, s) = \min_{i \in I} O_2(j, s)$ where $i$ is the first $|\lambda|$ elements in the inverted list for $\lambda$.

**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 $O_i$, an inverted list is constructed for each keyword in the partition. Specifically, for a partition with a center node $c$ and a keyword $\lambda$, the inverted list contains all nodes in the partition that contain keyword $\lambda$ ranked in nondecreasing order of their distances to $c$. Given a k-NK query $Q = (q, \lambda, k)$ and a distance oracle $O_i$, the algorithm first finds the partition that query $Q$ belongs to in $O_i$. The result w.r.t. $O_i$ is the first $k$ elements in the inverted list for $\lambda$ in the partition, denoted as $R_{O_i} = \{u_1 : \text{dist}(c, u_1) + \text{dist}(c, u_2) \cdot \text{dist}(c, u_3) \cdot \text{dist}(c, u_4) \cdot \text{dist}(c, u_5) \cdot \text{dist}(c, u_6) \}$. The final result $R$ is computed by merging the nodes in each $R_{O_i}$ 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 $O_1$ and $O_2$ in Fig. 2. For keyword $\lambda$, the inverted list for the partition centered at node $r$ in $O_2$ has 5 elements $\{b : 1, n : 3, k : 4, c : 5, t : 6\}$. The inverted list for the partition centered at node $o$ in $O_2$ has 1 element $\{k : 2\}$. Given a k-NK query $Q = (m, \lambda, k)$, from $O_1$, we can get a result $R_{O_1} = \{b : 1 + \text{dist}(r, m), n : 3 + \text{dist}(r, m) = \{b : 5, n : 7\}$, and from $O_2$, we can get a result $R_{O_2} = \{k : 2 + \text{dist}(o, m) = \{k : 3\}$. By merging $R_{O_1}$ and $R_{O_2}$, 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 worst distance used by [1] can be bounded by a factor of $2 \log_2 |V|$ 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 $O_1$ and $O_2$ in Fig. 2, for two nodes $s$ and $v$, the witness distance in $O_1$ is $\text{dist}(s, v) = \text{dist}(s, r) + \text{dist}(r, v) = 10$, and that in $O_2$ is $\text{dist}(s, v) = \text{dist}(s, n) + \text{dist}(n, v) = 6$. However, the exact distance is $\text{dist}(s, v) = 2$ in $G$, which is much smaller than both $\text{dist}(s, v)$ and $\text{dist}(s, r)$. 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.

### 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 $\lambda$, denoted as 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 $\lambda$, 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 NN$(v, \lambda)$. The partition is called tree Voronoi partition of $\lambda$, denoted as TVP$(\lambda)$. By precomputing TVP$(\lambda)$ for all keywords $\lambda$ on the tree, a query $Q = (q, \lambda, 1)$ can be answered in $O(|\log |V||)$ time using a binary search in TVP$(\lambda)$.

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

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

Using ECT$(\lambda)$, 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 $O(|\log |V||)$. The time to compute all compact trees and all extended compact trees for all keywords in the tree $T(V, E)$ is bounded by $O(|\log |V|| \cdot |\log |V||)$.

**Example 4:** Fig. 3 shows a tree with the preorder label from 1 to 20 on its nodes. For keyword $\lambda$, there are 5 keyword nodes $b, c, k, n, t$. The compact tree of $\lambda$, CT$(\lambda)$, is shown on the left part of Fig. 4. Node $r$ is in CT$(\lambda)$ because $r$ has three direct subtrees with nodes carrying keyword $\lambda$, $e$ is not in CT$(\lambda)$ because $e$ is not a keyword node and $e$ has only one direct subtree rooted at $m$ with nodes carrying keyword $\lambda$. The extended compact tree of $\lambda$, ECT$(\lambda)$, is shown in the middle part of Fig. 4.
preorder label marked beside each node. Node e is in ECT(λ), because for its parent node h, NN(h, λ) = b ≠ NN(e, λ) = c. The tree Voronoi partition of λ, TVP(λ), is shown on the right part of Fig. 4. For node s with preorder label 14, it is in the interval [11, 16], thus NN(s, λ) = c as listed in TVP(λ).

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 O_i, let the set of trees constructed in O_i be T_i. T_i can be considered as a tree by adding a virtual root and several virtual edges with weight +∞ 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 for each tree T_i, 1 ≤ i ≤ 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 O_1 and 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 O_1. For T_2, there are 4 trees rooted at nodes n, o, r, t respectively, because there are 4 partitions in 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 dist(s, v) in G. For a k-NK query Q = (m, λ, 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 ≥ 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 K_nearest nodes that contain λ. When processing a query Q = (q, λ, k) with k ≤ 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(K · |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|) and index size O(K · |doc(V)|) for all keywords where K is an upper bound value of k. Our second algorithm tree-pivot can handle an arbitrary k with query processing time O(k · log |V|) and index size O(|doc(V)| · 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.

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

In this section, we study how to answer a k-NK query Q = (q, λ, 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 K, i.e., k ≤ K. Recall that for a keyword λ, its compact tree CT(λ) keeps all the structural information of λ on the tree T. Our idea is to precompute the top-k results for every keyword λ and every node on CT(λ). Since the total size of all compact trees is bounded by O(|doc(V)|), the total space to store the top-k results of nodes on all compact trees is bounded by O(K · |doc(V)|). Given a query Q = (q, λ, k), if q is on CT(λ), we can simply report the precomputed answer on CT(λ). If q is not on CT(λ), we need to find a way to construct the answer using the precomputed results as well as the structure of CT(λ) and T. In the following, we first introduce how to answer a k-NK query using CT(λ), 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 CT(λ), we use a candidate list cand_v(v) to denote the precomputed k-NK results for k = K on node v ranked in nondecreasing order of their distances to v, in the form of cand_v(v) = {v_1 : dist(v, v_1), v_2 : dist(v, v_2), ..., v_q : dist(v, v_q)} where dist(v, v_q) ≤ dist(v, v_q-1) ≤ ... ≤ dist(v, v_1). Given a query Q = (q, λ, k) on a tree T(V, E) where k ≤ K, if q is in CT(λ), we can simply report the first k elements in cand_v(q) as the answer. The difficult case is when q is not in CT(λ). In order to answer such a query, we define an entry edge to be the edge in CT(λ) that is nearest to q. Intuitively, the entry edge plays a role of connecting the query node q to the compact tree CT(λ). The formal definition of entry edge is as follows.

Definition 3. (Entry Node and Entry Edge) Given a compact tree CT(λ), for each edge (u, u') on CT(λ) 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 CT(λ), denoted as v ∈_s CT(λ), if and only if there exists an edge (u, u') on CT(λ) such that v is on the path from u to u' on T, otherwise v does not stick to CT(λ), denoted as v /∈_s CT(λ). 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 ∈_s CT(λ), v is called the Entry Node of q w.r.t. λ.
denoted as \( E_{\lambda}(q) \). The corresponding edge \((u, u')\) on \( CT(\lambda)\) is called the Entry Edge of \( q \) w.r.t. \( \lambda \), denoted as \( EE_{\lambda}(q) \).

Note that for a node \( q \) and a keyword \( \lambda \), \( EE_{\lambda}(q) \) is an edge on the compact tree \( CT(\lambda) \), and \( E_{\lambda}(q) \) 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 \( \lambda \), the compact tree \( CT(\lambda) \) is shown on the left part of Fig. 4. For ease of illustration, we also mark the nodes in \( CT(\lambda) \) dark on the tree \( T \) in Fig. 3. For edge \((r, c)\) in \( CT(\lambda) \), \( h \in CT(\lambda) \) because \( h \) is on the path from \( r \) to \( e \) in \( T \). \( p \notin_{\lambda} CT(\lambda) \) since \( p \) is not on the path of any \( CT(\lambda) \) edge. For node \( v \), its entry node is \( EN_{\lambda}(v) = e \), as \( e \) is the first node on the path \((u, p, e, d, r)\) such that \( e \in_{\lambda} CT(\lambda) \). The entry edge for \( v \) is \( EE_{\lambda}(v) = (r, e) \) since the entry node \( e \) on \( v \) is on the path from \( r \) to \( e \) in \( T \). The entry nodes and entry edges for some other nodes in \( T \) are listed in the following table.

<table>
<thead>
<tr>
<th>Node ( g )</th>
<th>( j )</th>
<th>( d )</th>
<th>( e )</th>
<th>( p )</th>
<th>( u )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( EN_{\lambda} )</td>
<td>( g )</td>
<td>( j )</td>
<td>( d )</td>
<td>( e )</td>
<td>( p )</td>
</tr>
<tr>
<td>( EE_{\lambda} )</td>
<td>((r, e))</td>
<td>((r, e))</td>
<td>((r, e))</td>
<td>((r, e))</td>
<td>((r, e))</td>
</tr>
</tbody>
</table>

**The Algorithm:** Given a tree \( T(V, E) \), for keyword \( \lambda \), all keyword nodes are contained in \( CT(\lambda) \). For any node \( q \in V \), the path from \( q \) to any keyword node will go through the entry node \( E_{\lambda}(q) \). Based on such property, the result of a query \( Q = (q, \lambda, k) \) is identical with the result of the query \( Q' = (E_{\lambda}(q), \lambda, k) \). However, \( E_{\lambda}(q) \) may not be on \( CT(\lambda) \), thus the result of \( Q' \) is not necessarily precomputed. Let \((u, u') = EE_{\lambda}(q)\), since \( E_{\lambda}(q) \) is on the path from \( u \) to \( u' \) on the tree \( T \), the path from \( E_{\lambda}(q) \) 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 \( cand_{\lambda}(u) \) and \( cand_{\lambda}(u') \) on \( CT(\lambda) \).

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 \( CT(\lambda) \) for each keyword \( \lambda \) and the list \( cand_{\lambda}(u) \) for every node \( u \) on \( CT(\lambda) \) have been computed. After initializing the result \( R \) in line 1, we find the entry edge \((u, u')\) for \( q \) on \( CT(\lambda) \) (line 2). We add a distance \( dist(q, u) \) to every node in \( cand_{\lambda}(u) \) 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 \( cand_{\lambda}(u') \) with the distance \( dist(q, u') \) (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 \( CT(\lambda) \) on the left part of Fig. 4, for a query \( Q = (o, \lambda, 2) \), the entry edge \( EE_{\lambda}(o) = (r, c) \). Suppose the lists \( cand_{\lambda}(r) = \{b : 1, n : 3\} \) and \( cand_{\lambda}(c) = \{c : 0, t : 1\} \) are precomputed. By adding \( dist(o, r) = 5 \) to \( cand_{\lambda}(r) \), and adding \( dist(o, c) = 2 \) to \( cand_{\lambda}(c) \), 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., \( dist(q, u) \) and \( dist(q, u') \) (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.

**Finding the Entry Edge:** Given a keyword \( \lambda \) for any node \( v \) on a tree \( T(V, E) \), our idea of finding the entry edge \( EE_{\lambda}(v) \) is similar to the idea of finding the 1-NK answer using the tree Voronoi partition \( TVP(\lambda) \) 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 \( \lambda \), denoted as \( EE(\lambda) \). Given \( EE(\lambda) \), \( EE_{\lambda}(v) \) can be computed easily using a binary search in \( EE(\lambda) \) in \( O(\log |V|) \) time. In the next subsection, we show how to build \( EE(\lambda) \) for all keywords efficiently and prove that the total size of \( EE(\lambda) \) for all keywords in \( T \) is bounded by \( O(\text{doc}(V)) \).

**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 \( LCA(u, v) \) as their lowest common ancestor. The distance of \( u \) and \( v \) can be computed as \( dist(u, v) = dist(r, u) + dist(r, v) - 2 \cdot dist(r, LCA(u, v)) \). Using the techniques in [2], \( LCA(u, v) \) can be found in \( O(1) \) time using \( O(|V|) \) index space. Thus \( dist(u, v) \) 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 destination \( \delta \) as input, and outputs a candidate list by adding \( \delta \) 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 \( O(\text{min}(|R_{1}| + |R_{2}|, k)) \). The \( \otimes_{k} \) and \( \oplus \) operators satisfy the commutative, associative and distributive laws as follows.

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

Given the tree \( T \), for every keyword \( \lambda \) in \( T \), the compact tree \( CT(\lambda) \) 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 \( EE(\lambda) \), is to find the entry edge for any node on \( T \). The second index is the candidate list \( cand_{\lambda}(v) \) for every node on \( CT(\lambda) \). 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 \( \lambda \), sharing the similar idea with the tree Voronoi partition \( TVP(\lambda) \), we construct an entry...
Algorithm 4: EEP-construct\((T, CT(\lambda))\)

```
Input: A tree \(T(V, E)\) and a labelled compact tree \(CT(\lambda)\).
Output: Entry edge partition \(\lambda\).
1 \(r \leftarrow \) the original root of \(CT(\lambda)\);
2 \(\lambda(\lambda) \leftarrow \emptyset\);
3 partition(EEP(\lambda), \(\{1, V\}\), \((r, r), CT(\lambda))\);
4 return EEP(\lambda);

Procedure partition(EEP(\lambda), interval \([s, t]\), edge \((u, u')\), \(CT(\lambda))\)

1. foreach subnode \(u''\) of \(u\) on \(CT(\lambda)\) in increasing preorder do
2. \([s', t'] \leftarrow \) interval of \((u', u'')\);
3. if \(s < s'\) then add \(((s', s - 1), (u, u'))\) to EEP(\lambda);
4. partition(EEP(\lambda), \([s', t']\), \((u', u'')\), \(CT(\lambda))\);
5. \(s \leftarrow t' + 1\);
6. if \(s \leq t\) then add \(((s, t), (u, u'))\) to EEP(\lambda);
```

to partition the interval \([1, |V|]\) into several disjoint intervals. Each entry in EEP(\lambda) 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 \(CT(\lambda)\) 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(\lambda) 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(\lambda) with the entry edge \((u, u')\) (line 11).

The time complexity of Algorithm 4 is \(O(|V(CT(\lambda))|)\) since every node on \(CT(\lambda)\) is visited once. For each edge \((u, u')\) on \(CT(\lambda)\), at most two intervals are added into EEP(\lambda). 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(\lambda) is no more than \(2 \times |V(CT(\lambda))|\).

Example 9: For the labeled compact tree \(CT(\lambda)\) shown in Fig. 6, when invoking partition(EEP(\lambda), \([1, 20]\), \((\phi, r), CT(\lambda))\), we process the three child nodes \(a, b, c\) 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(\lambda) with the entry edge \((\phi, r)\). \([2, 6]\) is processed recursively by invoking partition(EEP(\lambda), \([2, 6]\), \((r, a), CT(\lambda))\), and \([7, 20]\) is processed by the other two child nodes \(c\) and \(b\) similarly. EEP(\lambda) 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(\lambda) for all keywords can be constructed in \(O(|\text{doc}(V)|)\) time and stored in \(O(|\text{doc}(V)|)\) space.

5.3 Construction of Candidate List

Given a compact tree \(CT(\lambda)\) for a tree \(T\) and a keyword \(\lambda\), we need to compute the candidate list \(\text{cand}_x(v)\) for every node \(v\) on \(CT(\lambda)\). Since \(CT(\lambda)\) keeps the structural information of all keyword nodes in \(T\), it is sufficient to search only on \(CT(\lambda)\) to calculate \(\text{cand}_x(v)\). A simple solution is to compute each \(\text{cand}_x(v)\) separately on \(CT(\lambda)\). This approach may take \(O(|\text{doc}(CT(\lambda))|)\) time to calculate \(\text{cand}_x(v)\) for a node \(v\), thus \(O(|\text{doc}(CT(\lambda))|)\) time to compute all candidate lists in \(CT(\lambda)\) for one keyword \(\lambda\), 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 \(CT(\lambda)\). Note that in \(CT(\lambda)\), the path between two nodes \(u, v\) is unique. From node \(u\) to the lowest common ancestor of \(u\) and \(v\), \(\text{LCA}(u, v)\), and then from \(\text{LCA}(u, v)\) 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 \(CT(\lambda)\) twice to build the candidate lists for all nodes on \(CT(\lambda)\). The first traversal on \(CT(\lambda)\) is a bottom-up one, such that the candidate list on each node is propagated to all its ancestors on \(CT(\lambda)\). The second traversal on \(CT(\lambda)\) 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, CT (λ), k)

Input: A tree T, a compact tree CT (λ), and the upper bound of k, k.
Output: cand_d(v) for each v on CT (λ).
1. cand_d(0) ← ∅ for each leaf of CT (λ);
2. cand_d(v) ← {v : 0} for each node v on CT (λ);
3. for each v on CT (λ) in a bottom-up fashion do
   4. u ← the parent node of v on CT (λ);
   5. cand_d(u) ← cand_d(u) ⊕ cand_d(v) ⊕ dist(u, v);
4. for each v on CT (λ) in a top-down fashion do
   7. u ← the parent node of v on CT (λ);
   8. cand_d(v) ← cand_d(u) ⊕ cand_d(v) ⊕ dist(u, v);

Figure 7: Constructing Candidate Lists

Algorithm 5 shows the construction of the candidate lists on CT (λ). 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 ∅ (line 1-2). We then traverse CT (λ) in a bottom-up fashion, e.g., using postorder traversal. For each node v traversed, we merge cand_d(v) into that of its parent node u by adding a distance dist(u, v) to list cand_d(u) (line 3-5). At last, we traverse CT (λ) 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, cand_d(u), into that of v by adding a distance dist(u, v) to the list cand_d(u) (line 6-8). Since the ⊕ operator takes O(1) time, the time complexity of Algorithm 5 is O(κ · |V (CT (λ))|) using O(κ · |V (CT (λ))|) space.

Example 10: Fig. 7 shows the candidate lists after the bottom-up phase and the top-down phase for the compact tree CT (λ) 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 dist(c, t) = 1, thus cand_d(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 cand_d(t) = {t : 0, c : 1} after the top-down phase.

Theorem 3. Given a tree T, an upper bound of k, κ, and CT (λ) for all keywords λ, the candidate lists cand_d(v) for all keywords λ and all nodes v on CT (λ) can be constructed in O(κ · |V (T)|) time and stored in O(κ · |V (T)|) space.

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

Algorithm 1 can only process a k-NK query Q = (q, λ, k) with a bounded k, i.e., k ≤ κ, 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 LCA(u, v), and the second segment is from 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.

We use depth(T) to denote the depth of tree T, and 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 PV(v, T). We have |PV(v, T)| = depth(v, T). Given a keyword λ, for each node u on tree T, we use the candidate list cand_d(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 cand_d(u) = {u1 : distT(u, u1), u2 : distT(u, u2), ...} where distT(u, ui) ≤ distT(u, uj) ≤ ... ≤ distT(u, u1). In order to handle an arbitrary k, the size of cand_d(u) is not bounded by any predefined κ. Clearly, a node v ∈ cand_d(u) if and only if v contains keyword λ and u ∈ 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 ∑v∈V (λ) |PV(v, T)| = |V (T)| · depth(v, T). We use the following example to illustrate the pivot based approach.

Example 11: Fig. 8 shows a tree T with 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 cand_d(v) that contains all keyword nodes in its subtree, sorted in nondecreasing distances to v. For example, cand_d(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 n, PV(n, T) = {r, d}, for a k-NK query Q = (d, λ, 3), the path from d to the root r contains two nodes d and r. We merge the lists cand_d(d) and cand_d(r) by adding a distance dist(r, d) = 1 to all elements in cand_d(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 ∑v∈V (λ) depth(v, T), which can be large if depth(v, T) is large. Second, when processing a query Q = (q, λ, k), we need to traverse all nodes from the query node q to the root of T. This is also costly if 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 ∑v∈V (T) depth(v, T) = |V (T)| · depth(v, T) = n(n − 1), which is O(n²). 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.
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 \(DT(T)\) for \(T\) and the second index is the candidate list \(\text{cand}_\lambda(v)\) for each keyword \(\lambda\) and each node \(v\) on \(DT(T)\). We introduce them separately in the following.

Constructing \(DT(T)\): Before introducing how to construct a tree \(DT(T)\) to satisfy the three properties in Definition 5, we first present an approach to constructing a tree \(T'\) from \(T\), which satisfies properties \(P_3\) and \(P_4\). 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 \(T'_e\) of \(r\) on \(T'\), perform steps (1) and (2) on \(T'_e\) 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 \(LCA_{T'}(u, v) = r\). 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 \(dist_{T'}(u, v) = dist_{T'}(u, r) + dist_{T'}(v, r)\) on the original tree \(T\). From \(O_1\) and \(O_2\), we have \(dist_{T}(u, v) = dist_{T}(u, LCA_{T'}(u, v)) + dist_{T}(v, LCA_{T'}(u, v))\) 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 \(DT(T)\) 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 \(T_e\) of \(r\) on \(T\), \(|V(T_e)| \leq \frac{|V(T)|}{2}\) holds.
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 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 of the size of the tree will be larger than depth$T$ and such that the subtree rooted at $T$ node of $T$. Traversing each node in $T$ recursively (line 1-2), the median node is $v$ on the original $T$, and also gives a method to find the median node of $T$.

**Algorithm 7:** DT-construct $(T)$

Input: A tree $T$.
Output: A distance preserving balanced tree $DT(T)$.

1. $r \leftarrow$ the median node of $T$;
2. rotate $T$ with $r$ as the root;
3. $DT(T) \leftarrow$ a tree with a single node $r$;
4. foreach direct subtree $T_i$ of $r$ in $T$ do
5. DT$(T_i) \leftarrow$ DT-construct($T_i$);
6. add DT$(T_i)$ as a subtree of $r$ in DT$(T)$;
7. return DT$(T)$;

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

**Constructing cand$_j(v)$:** For a tree $T(V, E)$, given DT$(T)$, the algorithm for constructing the candidate list cand$_j(v)$ for each node $v$ and each keyword $\lambda$ is quite simple. For each node $v$, we propagate its keyword information to all its pivots in 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 DT$(T)$ and each keyword $\lambda$ that is contained in node $v$ (line 2-3). For each pivot $p$ of $v$ as well as $v$ itself, we calculate dist$_r(p, v)$ on the original tree $T$ and add the element $v \cdot \text{dist}_r(p, v)$ to the candidate list cand$_j(v)$ (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 $O(|\text{doc}(V)| \cdot |\log |V||)$, thus each keyword is propagated into at most $\log_2|V|$ candidate lists in DT$(T)$. For line 6-7, we need $O(|\text{doc}(V)| \cdot \log^2 |V|)$ time to sort all candidate lists in DT$(T)$.

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

**Algorithm 8:** cand-construct $(T, DT(T))$

Input: A tree $T$, a distance preserving balanced tree DT$(T)$.
Output: cand$_j(v)$ for each $v$ on DT$(T)$ and each keyword $\lambda$.
1. cand$_j(v) \leftarrow \emptyset$ for each node $v$ on DT$(T)$ and each keyword $\lambda$;
2. foreach $v \in V(\text{DT}(T))$ do
3. foreach $\lambda \in \text{doc}(v)$ do
4. foreach $p \in \text{PT}(v, \text{DT}(T)) \cup \{v\}$ do
5. cand$_j(p) \leftarrow \text{cand}_j(p \cup \{v \cdot \text{dist}_r(p, v)\});
6. output $v \in V(\text{DT}(T))$ and keyword $\lambda$ do
7. sort elements in cand$_j(v)$ 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. $T_i \leftarrow$ shortest path tree for $\mathcal{O}_i$;
4. $R \leftarrow R \oplus_k \text{tree-knk}(T_i, Q)$;
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 $\mathcal{O}_1, \mathcal{O}_2, \ldots, \mathcal{O}_k$ using the algorithm in [20]. Let the shortest path trees for the oracles be $T_1, T_2, \ldots, T_k$ respectively. Algorithm 9 gives 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 $\oplus_k$ 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 $T_1, T_2, \ldots, T_k$. For a keyword set $\lambda$ and a node $v$, let cand$_{\lambda}(v)$ 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

$$R \leftarrow R \oplus_k (\text{cand}_{\lambda}(v) \oplus \text{dist}_{T_i}(q, v)) \oplus_k (\text{cand}_{\lambda}(v) \oplus \text{dist}_{T_j}(q, v)).$$

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 cand$_{\lambda}$ 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 $\lambda$ for all trees $T_1, T_2, \ldots, T_k$. Denote the global candidate list of node $v$ and keyword $\lambda$ as cand$_{\lambda}(v)$. It is computed by

$$\text{cand}_{\lambda}(v) = \text{cand}_{\lambda}(v) \oplus \text{cand}_{\lambda}(v) \oplus \cdots \oplus \text{cand}_{\lambda}(v).$$

For a node $v$, a node $v' \in \text{cand}_{\lambda}$ may appear in the candidate list cand$_{\lambda}(v)$ of multiple trees $T_i$, but will be stored at most once in the global candidate list cand$_{\lambda}(v)$. 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 dist$_{T_i}(q, v)$ to cand$_{\lambda}(v)$ using the $\oplus$ operator, i.e., cand$_{\lambda}(v) \oplus$
ent 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 $R_1 = \{v_1 : d_{v_1}, u_2 : d_{v_2}, \cdots\}$ and $R_2 = \{v_1 : d_{v_1}, v_2 : d_{v_2}, \cdots\}$ sorted in non-decreasing 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 $d_{v_i} \leq d_{v_i}$ 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 $\text{dist}_i(q,v)$ 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 Eq. 1. For any two candidate lists $\text{cand}^a_{\lambda} \oplus \text{dist}_i(q,v)$ and $\text{cand}^b_{\lambda} \oplus \text{dist}_j(q,v)$, using Eq. 1, we can generate a better result by merging $\text{cand}^c_{\lambda} \oplus \text{dist}_k(q,v)$ using $\ominus$, first, then taking distances $\text{dist}_i(q,v)$ and $\text{dist}_j(q,v)$ out and applying the minimum value of them. Clearly, $(\text{cand}^c_{\lambda} \ominus_k \text{cand}^a_{\lambda}) \oplus \min\{\text{dist}_i(q,v), \text{dist}_j(q,v)\}$ is a valid candidate list for query $Q$ because $\text{cand}^c_{\lambda} \ominus_k \text{cand}^a_{\lambda}$ is a candidate list for node $v$ and $\min\{\text{dist}_i(q,v), \text{dist}_j(q,v)\}$ suggests a path from $q$ to $v$ in $G$.

\[
(\text{cand}^c_{\lambda} \ominus_k \text{dist}_i(q,v)) \ominus_k (\text{cand}^c_{\lambda} \ominus_k \text{dist}_j(q,v)) = (\text{cand}^c_{\lambda} \ominus_k \text{cand}^a_{\lambda}) \oplus \min\{\text{dist}_i(q,v), \text{dist}_j(q,v)\}
\]

The second problem can be solved if we prove that by merging more candidate lists using the $\ominus$ operator, the answer will not get worse. Consider a node $v' \in \text{cand}^a_{\lambda}$, 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 Eq. 2.

\[
\text{cand}^c_{\lambda} \ominus_k \text{cand}^a_{\lambda} \ominus \text{dist}_k(q,v)
\]

Eq. 1 and Eq. 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 $\text{DT}(T_1)$ and $\text{DT}(T_2)$ for the shortest path tree $T_1$ and $T_2$ shown in Fig. 5. with candidate list marked beside each node for keyword $\lambda$. Using global storage, for the same node on different trees, we merge all its candidate lists using $\ominus$ 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, $\text{cand}^a_{\lambda} \oplus \text{dist}_1(p,e)$, $\text{cand}^b_{\lambda} \oplus \text{dist}_2(p,e)$, and $\text{cand}^c_{\lambda} \oplus \text{dist}_3(p,e)$. Using global storage, only two candidate lists $\text{cand}^a_{\lambda} \oplus \min\{\text{dist}_1(p,e), \text{dist}_2(p,e)\}$, $\text{cand}^b_{\lambda} \oplus \text{dist}_3(p,e)$ need to be merged. For query $Q_2 = (h, \lambda, 2)$, without global storage, we get the result $R = \{e : 3, t : 4\}$. Using global storage, we can get a result $R' = \{n : 2, e : 3\} \geq R'$. □

**Handling Multiple Keywords:** We discuss how to extend our approach to handle a k-NK query of multiple keywords with AND (denoted as $\land$) and OR (denoted as $\lor$) semantics. Without loss of generality, we assume the format of a keyword expression is $(\lambda_1 \land \lambda_1 \land \lambda_2 \cdots) \lor (\lambda_2 \land \lambda_2 \land \cdots) \cdots$. It is easy to handle $\land$, by answering each $\lambda_1 \land \lambda_2 \cdots$ separately and merging the results using the $\otimes$ operator. For handling $\lambda_1 \land \lambda_2 \cdots$, we select a keyword $\lambda_{ij}$ from $\{\lambda_1, \lambda_2, \cdots\}$ with the least frequency $|\lambda_{ij}|$, as the primary keyword and consider other keywords as filter keywords. We answer the query for the single keyword $\lambda_{ij}$. 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 list of 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.

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

<table>
<thead>
<tr>
<th>Boundk</th>
<th>Pivot</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query Time ($T$)</td>
<td>$O(\log</td>
</tr>
<tr>
<td>Index Time ($T$)</td>
<td>$O(</td>
</tr>
<tr>
<td>Index Size ($T$)</td>
<td>$O(</td>
</tr>
<tr>
<td>Query Time ($G$)</td>
<td>$O((</td>
</tr>
<tr>
<td>Index Time ($G$)</td>
<td>$O(</td>
</tr>
<tr>
<td>Index Size ($G$)</td>
<td>$O(</td>
</tr>
</tbody>
</table>

### Table 2: Dataset Statistics

<table>
<thead>
<tr>
<th>Dataset</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DBLP</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$</td>
<td>V</td>
</tr>
<tr>
<td></td>
<td>695</td>
<td>469</td>
</tr>
<tr>
<td>FLARN</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>376</td>
<td>301</td>
</tr>
</tbody>
</table>

### 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 $\log_2 \deg(u) + \log_2 \deg(v)$ is assigned to

1. http://www.informatik.uni-trier.de/~ley/db
Figure 12: Hit rate, Spearman’s rho and Error by Varying k

edge \((u, v)\), where \(\text{deg}(u)\) 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 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 \(\lambda\) 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 \(R = \{u_1 : d_1, \ldots, u_m : d_m\}\) in nondecreasing order of their distances, and \(\mathcal{R} = d_k\) as the upper bound distance of the result \(R\). Denote an approximate result set as \(\mathcal{R}' = \{u_1' : d_1', \ldots, u_m' : d_m'\}\) in nondecreasing order of their distances. The hit rate is defined as:

\[
\text{hit}(\mathcal{R}') = \frac{|\{i \in [1, k] \mid \text{dist}(u_i', q) \leq d_i\}|}{k}
\]

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

\[
\text{err}(\mathcal{R}') = \sum_{1 < i < k} |d_i'/d_i - 1| / k
\]

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 Spearman’s rho.

Figure 13: Query Time in Microseconds by Varying k

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 techniques 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 \(k\).

Query Time. Figure 13 shows the query time of different methods in log scale when we vary \(k\). The query time of BFS is \(10^{-2}-10^{-6}\) 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 \(|V_\lambda|\), where \(|V_\lambda|\) is the frequency of keyword \(\lambda\).

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,
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 keyword search on XML documents [22] and top-k nearest keyword search on unstructured, semi-structured and structured data. In SIGMOD, pages 927–940, 2008.

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