Finding Nearest Neighbors in Road Networks: A Tree Decomposition Method

Fang Wei-Kleiner
Computer Science Department
Linköping University
Linköping, Sweden
fang.wei-kleiner@liu.se

ABSTRACT
Finding k Nearest Neighbors in one category of POIs (point of interests) belongs to the most frequently issued queries in the navigating systems or online maps. This problem can be formulated as given a graph $G(V,E)$, a vertex $u$ and $S \subseteq V$, finding $k$ nearest neighbors of $u$ in $S$. Classic Dijkstra’s algorithm offers an optimal solution if $S = V$ holds, but the performance deteriorates as $S$ is of smaller size. Other approaches such as pre-computing and storing all the shortest distances require too much storage, thus suffer from drawbacks of scalability.

To address these problems, we propose TlkNN (stands for Tree decomposition-based Indexing for kNN), an indexing and query processing scheme for kNN query answering. TlkNN is based on the tree decomposition methodology. The graph is first decomposed into a tree in which each node (a.k.a. bag) contains more than one vertex from graph. The shortest paths are stored in such bags and these local paths together with the tree are the components of the index of the graph. Based on this index, step-wise query processing over the tree can be executed to find the nearest neighbors.

Our experimental results show that TlkNN offers orders-of-magnitude performance improvement over Dijkstra’s algorithm on query answering, while the storage requirement for the index structure is relatively small.

Categories and Subject Descriptors
H.3.3 [Information Search and Retrieval]: Search process; H.3.1 [Content Analysis and Indexing]: Indexing methods

General Terms
Algorithms, Design

Keywords
graphs, indexing, k nearest neighbors, shortest path, tree decomposition

1. INTRODUCTION
As the online mapping and routing services such as Google Maps and other mobile apps become more and more popular, the need of the real time computation for these services is of vital importance. One of the most frequently used services is for instance to compute the shortest path between two locations. The typical setting of such road network problems is that we consider the map as an undirected graph where the locations are the nodes and the paths between the locations are modeled as edges. Another interesting problem is to find the nearest POIs (Point of Interest) given the user’s current location. POIs can be for instance petrol stations, restaurants, etc.

The above road network problem can be formulated as the $k$ Nearest Neighbor (kNN) graph problem, where $G = (V,E)$ is a weighted and undirected graph, and $u \in V$ is the vertex representing the users location, while $S \subseteq V$ representing the POIs. In essence, we aim at computing the shortest distances from the starting vertex $u$ to the vertices in $S$, and selecting $k$ nearest neighbors in $S$ with respect to $u$.

A straight forward solution by computing the shortest distance one by one has the drawback that the performance deteriorate as the size of $S$ increases. Moreover, the independent computation of the shortest distances could result in repeatedly calculating the same shortest distance several times. On the other hand, Dijkstra’s algorithm [2] generates the vertices with an increasing order with respect to the shortest distances to the starting vertex $u$. However, the generation of the vertices is independent to $S$. Hence, for a specific value of $k$, if $|S| \ll |V|$, the time consumption approaches the worst case complexity. Therefore, requiring that the result be obtained in real time (or almost real time) precludes the use of conventional algorithms such as INE and IER methods [6], which are basically improvements of Dijkstra’s algorithm.

One naive solution to achieve the optimal query answering time is to store the shortest paths for all the vertex pairs in the graph. However, the space requirement of $O(N^3)$ ($N = |V|$) is prohibitively high thus prevents the graph from being useful as long as the graph size is large. Samet et al. [7] proposed an approach by pre-computing the shortest paths between all possible vertices in the network and then making use of a data structure called shortest path quadtree to compress the index structure. The index size is reduced from $O(N^3)$ to $O(N^{1.5})$.

Note that the space requirement of $O(N^3)$ is an upper bound, where the average path length $L$ is estimated as $N$. In practice, however, $L$ is smaller than $N$.
1.1 Our Contributions

In this paper, we aim at building an index structure for a given graph with the tree decomposition methodology and solving the kNN problem based on this index structure. Briefly stated, we first decompose the graph \( G \) into a tree in which each node contains a set of vertices in \( G \). These tree nodes are called bags, in the sense that any of them contains more than one vertices. Different from other partitioning based methods, there are overlappings among the bags, i.e., for any vertex \( v \) in \( G \), there is more than one node in the tree which contains \( v \). However, it is required that all these related bags constitute a connected subtree (see Definition 2 for the formal definition). One important measurement of a tree decomposition is the treewidth, which is (roughly) the maximal bag cardinality of the underlying tree decomposition. Treewidth is a parameter associated with the tree-likeness property of a graph. Namely the more tree-like the graph, the smaller the treewidth.

Upon the tree decomposition, we pre-compute the shortest distances between all the vertex pairs located in the same bag. As a result, the space requirement for the storage of the index structure is reduced to \( O(L \cdot |R|^2) \) where \( |R| \) is the parameter approximating the treewidth of the graph and \( L \) is the average path length of the stored paths. Experimental results show that the relationship \( |R| \ll N \) and \( L \ll N \) hold for road networks. For instance, the \( |R| \) parameter of the road network in our test with 1.2 million vertices is 562, and \( L \) is 55, which yield the space requirement of approximately 176MB. This is near to one magnitude less than the estimated space requirement in [7] (which amounts to more than 1.3GB).

Based on the index structure, we propose an efficient algorithm TlkNN solving the kNN query answering problem. The basic part of the TlkNN algorithm is akin to the shortest path query answering algorithm proposed by Wei [11]. The prominent feature of the algorithm is that the time complexity depends on the treewidth of the graph. Since the graphs models of road networks have small treewidth, the efficiency is improved. Moreover, we integrate two pruning strategies into the tree decomposition-based algorithm to speed up the query processing.

Finally we conduct experiments on a road network dataset which contains more than one million vertices and compare the results with Dijkstra’s algorithm. The experimental results confirm our theoretical analysis by demonstrating that our algorithm offers orders-of-magnitude performance improvement over Dijkstra’s algorithm.

The rest of the paper is organized as follows: in Section 3 we introduce the problem definition and describe how kNN query answering can be efficiently processed if the graph is a tree. This offers intuitions for the understanding of the tree decomposition-based algorithm, which is to some extent a generalization of tree-based algorithm. In Section 4 we introduce the formal definitions of tree decomposition and prove the theoretical results regarding to shortest path query answering. Section 5 contains the detailed algorithms and the complexity analysis. In Section 6 we present the experimental results. We conclude in Section 7.

2. RELATED WORK

There are numerous research papers tackling the kNN problem, while most of the work are based on the model of spatial databases, where the objects are identified with the coordinates and the Euclidean distance is used as the distance measurement. For instance, Jensen et al. [4] propose a data model and definition of abstract functionality required for NN queries in spatial network databases. Shekhar et al. [10] present four alternative techniques for finding the first nearest neighbor to a moving query object on a given path. Papadias et al. [6] presented a solution called INE for NN queries in spatial network databases by introducing an architecture that integrates network and Euclidean information and captures pragmatic constraints.

Another branch of related work proposes pre-computing and storing all the shortest distances of the graph \( G \) to speed up the process of shortest path computation (e.g., [3, 5, 13]). Unfortunately, the space requirement of \( (N^3) \) prevents these approaches from being attractive as long as large graph data is considered.

As mentioned in the Introduction section, the kNN algorithm we tackle is a special case of Dijkstra’s Algorithm, in the sense that the set \( S \subseteq V \) plays an important role for the efficiency issues. Clearly if \( S = V \), the problem is equivalent to the one point shortest path problem. Another related problem is the well-known P2P shortest path problem, where two points (source and target) are under consideration [12, 9, 8, 1]. However, our kNN algorithm aims at searching the entire graph for the k nearest neighbors in \( S \), while P2P algorithms follows in general the goal oriented guidelines. Another interesting observation is that both problems (kNN and P2P) can be solved by simply applying Dijkstra’s Algorithm: for the kNN problem, the Dijkstra’s algorithm starts from one point shortest path problem, while for P2P problem, the Dijkstra’s algorithm starts from the source vertex \( s \) and stops as soon as the target vertex \( t \) is met. Notwithstanding these approaches, we still aim at searching for novel solutions, since the naive ones are not optimal.

3. PRELIMINARIES

3.1 Problem Definition

In this paper, a road network is modeled as an undirected graph \( G = (V, E) \), \( n = |V| \) and \( m = |E| \), where \( V \) is the set of the vertices and \( E \) the set of the edges. Each edge \((u, v) \in E\) is associated with a nonnegative weight representing the time cost of traveling or simply the road distance between the two neighboring nodes, denoted as \( w(u, v) \). A path \( P \) from node \( u \) to \( v \) is a sequence of nodes \( P = \{u, u_1, \ldots, u_l, v\} \), where \((u, u_1), \ldots, (u_l, v) \in E \) and the length \(|P|\) is the sum of the weight of all edges on \( P \). We define the network distance \( sdist(u, v) \) between two vertices \( u \) and \( v \) as the length of the shortest path \( SP(u, v) \) of \( u \) and \( v \).

**Definition 1** (k Nearest Neighbor (kNN) query). Let \( G = (V, E) \) be the graph of a road network and \( u \in V \). Let further \( S \subseteq V \). The k Nearest Neighbor (kNN) of \( u \) with respect to \( S \) is the set of vertices \( S^k = \{s_1, \ldots, s_k\} \), where \( S^k \subseteq S \), such that \( sdist(u, s_1) + \ldots + sdist(u, s_k) \) is minimum. That is, \( \forall x \in S \setminus S^k, \forall y \in S^k \): \( sdist(u, x) \geq sdist(u, y) \).

In the rest of paper, we will use the term graph representing a graph of a road network.
3.2 kNN over Trees

In this section we describe how the kNN query can be efficiently answered over a tree structure. Let $G = (V, E)$ be a tree, that is, there is a unique path between any two vertices $u, v \in V$. Therefore, given a vertex $u \in V$, the shortest paths from $u$ to all the vertices in $V$ can be retrieved by simply traversing over the tree, starting from $u$. Since each vertex is only accessed once, the worst case complexity is $O(n + m)$. As a matter of fact, more optimization approaches can be applied to achieve higher efficiency.

Consider the kNN query processing over the tree $G = (V, E)$. For a given vertex $u \in V$ and a vertex set $S$, we aim at finding $k$ vertices in $S$ whose shortest distance to $u$ is minimum. Without the loss of generality, we assume the root of the tree is $r$, as shown in Figure 1. Clearly, the vertices in $S$ form a borderline which cut the tree into two parts, so that the part below the borderline does not contain any vertex in $S$. This way, search space is restricted into the upper part of the tree.

The second optimization approach is to arrange the traversing sequence of the vertices during the query processing. Assume that the starting vertex is $u$, as depicted in Figure 1. Moreover, an ordered list is used to store the current $k$ nearest neighbors of $u$, and we denote the current $k$th nearest neighbor as upper bound. Of course at the beginning of the traversing, the list is set as empty. We follow the following traversing sequence:

1. starting from $u$, top-down traversing (either breath-first or depth-first) the subtree rooted at $u$;
2. moving to the parent $v$, then top-down traversing the subtree rooted at $v$,
3. recursively executing step 2 till the root $r$ is reached.

The above approach clearly shows that over a tree structure, the kNN query answering algorithm follows a traversal order regarding to $S$, (instead of $N$ as in Dijkstra algorithm), in the sense that the smaller the shortest distance of the vertex $v \in S$ is, it is more probable that $v$ is visited earlier.

With the effective pruning strategy, the number of visited vertices can be considerably reduced.

In the following sections, we propose an index structure of graphs named tree decomposition, to speed up the processing of kNN query answering. The structure of tree decomposition has the form of a tree while the nodes in the tree consist of a set of vertices in the original graph $G$. The query processing over the tree decomposition is thus a generalization of that over the tree.

4. TREE DECOMPOSITION

In this section we first present the definition of tree decomposition on graphs. The definitions on the tree decomposition as well as treewidth follows the standard definitions, which can be found in other literals. However, we present them here in order to improve the readability of the paper.

In the second part of this section, we show how shortest distance query answering is processed over the tree decomposition. Note that the proposed algorithms differs from the work presented in [11], in the sense that we consider the computation of the shortest distance from a source vertex $u$ to all the other vertices in the graph. While in [11], we concentrate on the shortest distance computation from a source vertex $u$ to another vertex $v$.

A tree decomposition of a graph $G$ is defined as follows:

**Definition 2 (Tree Decomposition).** A tree decomposition of $G = (V, E)$, denoted as $T_G$, is a pair $(\{X_i \mid i \in I\}, T)$, where $\{X_i \mid i \in I\}$ is a collection of subsets of $V$ and $T = (I, F)$ is a tree such that:

1. $\bigcup_{i \in I} X_i = V$.
2. for every $(u, v) \in E$, there is $i \in I$, s.t. $u, v \in X_i$.
3. for all $v \in V$, the set $\{i \mid v \in X_i\}$ induces a subtree of $T$.

Note that the tree structure is represented as $T = (I, F)$, where $I$ is the set of tree nodes ($i \in I$) and $F$ is the parent-child relationship of the tree nodes in $I$. Each node consists of a set of vertices in $V$. We call the sets $X_i$ bags.

In this paper, we distinguish the node in a graph $G$ and the tree decomposition $T$ by denoting the node in the graph $G$ as vertex, and the node in the tree decomposition as tree node or simply node. For each tree node $i$, there is a bag $X_i$ consisting of vertices. We use the term $X_i[j]$ to represent the vertex at position $j$ in the bag of $X_i$. To simplify the presentation, we will sometimes use the term node and its corresponding bag interchangeably.

**Definition 3 (Induced Subtree, RIS).** Let $G = (V, E)$ be a graph and $T_G = (\{X_i \mid i \in I\}, T)$ be the tree decomposition, where $T = (I, F)$. Let $v \in V$.

- The induced subtree of $v$, is a pair $(\{X_i \mid i \in I_v\}, T_v)$, where $T_v = (I_v, F_v)$, such that $I_v \subseteq I$ and $F_v \subseteq F$, and $\forall i \in I, i \in I_v$ iff $v \in X_i$.

Following this traversing sequence, if a new vertex $z$ is reached, then the shortest distance from $u$ to $z$ ($sdist(u, z)$) can be used as the lower bound, in the sense that if $sdist(u, z)$ is greater than the upper bound of the ranked list, pruning can be executed.

- If $z$ is located along the path from $u$ to $r$, then the traversal terminates. Because every unvisited vertex has greater shortest distance to $u$ than $sdist(u, z)$, therefore greater than the upper bound.

- Otherwise, the subtree rooted at $z$ can be pruned.

$\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{knn_tree.png}
\caption{kNN over a Tree}
\end{figure}$
• The Root of the Induced Subtree of \( v \) is referred to as \( RIS(v) \).

Intuitively, the induced subtree of \( v \) contains precisely the bags in \( T_G \) in which \( v \) is an element. Due to the third condition of the tree decomposition, these bags are connected.

Given any graph \( G \), there may exist many tree decompositions which fulfill all the conditions in Definition 2. However, we are interested in those tree decompositions with smaller bag sizes. We call the cardinality of a bag the width of the bag.

Definition 4 (Width, Treewidth). Let \( G = (V, E) \) be a graph.

• The width of a tree decomposition \( (\{X_i \mid i \in I\}, T) \) is defined as \( \max\{|X_i| \mid i \in I\} \).

• The treewidth of \( G \) is the minimal width of all tree decompositions of \( G \). It is denoted as \( \operatorname{tw}(G) \) or simply \( tw \).

Given a graph \( G \), deciding whether its treewidth is less than a constant is intractable. In [11], a linear time decomposition which fulfills all the conditions in Definition 2. However, the R-Algorithm ensures that \( RIS(v) = v \), which means, that every bag is precisely the root of the induced subtree of its first vertex element. This way, there is in fact an one-to-one mapping between the vertices in \( G \) (except for the vertices in the root of \( T \)) and the tree nodes in \( T \) (except for the root node \( X_0 \)). These properties can be formulated as follows:

Lemma 1. Let \( G = (V, E) \) be a graph and \( T_G \) be the tree decomposition computed by the R-Algorithm, where \( |R| \) is the size of the root and \( w \) is the upper bound of the bag width for the rest of the tree nodes. \( G \) has small width if \( |R| \ll N \).

In the rest of the paper, for a graph \( G \), we will deal with the tree decomposition \( T_G \) computed by applying the R-Algorithm. The following properties of \( T_G \) do not necessarily hold for tree decompositions computed by algorithms other than the R-Algorithm.

Let \( X_i \) be a bag other than the root \( (i \neq 0) \), and \( v \) be the first element of \( X_i \) \( (v = X_i[0]) \). R-Algorithm ensures that \( RIS(v) = v \), which means, that every bag is precisely the root of the induced subtree of its first vertex element.

The third property states that other than the first element, all the vertices in the child bag can be found in the parent bag. This property is also a consequence of the R-Algorithm and does not hold on general tree decompositions.

4.1 Shortest Distance Computation over Tree Decomposition

Given a graph \( G = (V, E) \) and \( u \in V \), the shortest distance from \( u \) to the other vertices in \( V \) can be computed based on the tree decomposition \( T_G \). The basic idea is for each bag \( X \), the local shortest distances between any two vertices \( u, v \in X \) have to be pre-computed and stored as the index structure. Based on this structure, the computation can be conducted as follows. First \( RIS(u) \) is located in \( T_G \), and we denote it as \( X \). Note that if \( X \) is a non-root node in \( T_G \), then \( u = X[0] \) holds. Based on the assumption that the local shortest distances inside each bag are pre-computed and stored, the shortest distance between \( u \) and \( X[1], \ldots, X[n] \) is retrieved from the index structure. Then, the shortest distance from \( u \) to the other vertices are computed by traversing from \( X \) to all its neighbor nodes, which include its child nodes and its parent node. This traversing process is executed till all the nodes of the entire tree are visited. Note that the traversal sequence and pruning strategies are presented in the next section. In this section we introduce two basic traversing steps which lay the ground stone of the kNN algorithm, namely bottom-up and top-down traversals.

• Bottom-up Let \( X_i \) be the child bag of \( X_p \). Assume \( RIS(u) \) is located in the subtree rooted at \( X_c \), and \( \operatorname{sdist}(u, X_c[i]) \) are known. We first obtain the vertex set \( X = X_c \cap X_p \), which is the set of vertices occurring in both bags.

Following the analogous argument we can compute \( \operatorname{sdist}(u, y) \) as follows:

\[
\operatorname{sdist}(u, y) = \min(\operatorname{sdist}(u, z) + \operatorname{sdist}(z, y), z \in X)
\]

The pseudo code of the bottom-up algorithm is given in Algorithm 1.

• Top-down The setting for top-down processing is similar to that of bottom-up: \( X_c \) be the child bag of \( X_p \). The difference is that we require that \( RIS(u) \) is not in the the subtree rooted at \( X_c \). Now we aim at computing \( \operatorname{sdist}(u, y) \) for every vertex \( y \) in \( X_c \). Following the analogous argument we can compute \( \operatorname{sdist}(u, y) \) as

\[
\operatorname{sdist}(u, y) = \min(\operatorname{sdist}(u, z) + \operatorname{sdist}(z, y), z \in X)
\]

where \( X = X_c \cap X_p \). The pseudo code of the top-down algorithm is given in Algorithm 2.

5. TIKNN ALGORITHMS OVER TREE DECOMPOSITION

In this section, we present the detailed algorithms for the kNN query answering algorithms.
Algorithm 1 Bottomup($X_c, X_p, u$)

Input: $G = (V, E)$, $X_c$ is the child node of $X_p$ in $T_G$; $sdist(u, X_p[i])_{0 \leq i < |X_p|}$
Output: $sdist(u, X_p[i])_{0 \leq i < |X_p|}$
1: $X = X_c \cap X_p$
2: copy $sdist(u, x)(x \in X)$ from $X_c$ to $X_p$
3: for $y \in X_p \setminus X$ do
4: $sdist(u, y) = \min(sdist(u, z) + sdist(z, y)_{z \in X})$
5: end for

Algorithm 2 Topdown($X_c, X_p, u$)

Input: $G = (V, E)$, $X_c$ is the child node of $X_p$ in $T_G$, $sdist(u, X_p[i])_{0 \leq i < |X_p|}$
Output: $sdist(u, X_p[i])_{0 \leq i < |X_p|}$
1: $X = X_c \cap X_p$
2: copy $sdist(u, x)(x \in X)$ from $X_p$ to $X_c$
3: for $y \in X_c \setminus X$ do
4: $sdist(u, y) = \min(sdist(u, z) + sdist(z, y)_{z \in X})$
5: end for

5.1 TiKNN Algorithms in a Nutshell

Given the graph $G = (V, E)$ and the corresponding tree decomposition $T_G$, as well as the pre-computed local shortest distances in the local bags, we can compute the shortest distance from $u$ to all the other vertices in $G$ by starting from $RIS(u)$ in $T_G$ and traversing the entire tree decomposition using the Bottom-up and Top-down algorithms from the last section.

This solution is however not optimal. Recall from the algorithm of computing shortest distance over a tree structure introduced in Section 3.2 certain pruning techniques are applied during the traversal through the tree nodes. The first pruning strategy, namely first removing the branches in the tree which do not contain any vertex in $S$, can be easily adapted here. Recall that there is a one-one mapping between a vertex $v$ in $G$ and a tree node in $T_G$, namely $RIS(v)$. Therefore, given the set $S$, we can identify the corresponding tree node set in $T_G$. We call this process $S$-pruning (see Figure 2(a)).

![Figure 2: S-pruning (a) and traversal sequence (b)](image)

Analogously, we can also follow the traversal sequence proposed in Section 3.2. That is, the sequence of top-down $\rightarrow$ one level bottom-up $\rightarrow$ top down..., until the root is reached. Figure 2(b) illustrates the sequence.

Now let us consider the pruning strategy during the traversal. In contrast to the tree structure, the tree nodes in the tree decomposition contain a set of vertices. Thus at each step of the traversal, assume the current bag is $X$, we obtain a set of shortest distance $\{sdist(u, v) | v \in X\}$. Assume further that we maintain an ordered list of $\{list(i) | 1 \leq i \leq k\}$ which stores the shortest $k$ distances from $u$ to the vertices in $S$, that we have visited so far. Now consider the case where a vertex $v \in X$ such that $sdist(u, v) > list(k)$. That is, the shortest distance from $u$ to $v$ is greater than the distance of the current $k$th neighbor. If this is the case, we can remove $v$ as well as $sdist(u, v)$ from $X$ for the further processing. This is because if $sdist(u, v)$ serves for computing shortest distance for any vertex $w$ which is not yet visited, then the value of $sdist(u, w)$ will be greater than list($k$) as well. Since list($k$) is the upper bound of the current list of $k$ nearest neighbors, $w$ will never be a member of $k$ nearest neighbors.

Algorithm 3 Bottomup with Pruning($X_c, X_p, u$)

Input: $G = (V, E)$, $X_c$ is the child node of $X_p$ in $T_G$, $X_p \subseteq X_c$, $sdist(u, X_p[i])_{0 \leq i < |X_p|}$
Output: $sdist(u, X_p[i])_{0 \leq i < |X_p|}$
1: $X = X_c \cap X_p$
2: copy $sdist(u, x)(x \in X)$ from $X_p$ to $X_c$
3: for $y \in X_c \setminus X$ do
4: $sdist(u, y) = \min(sdist(u, z) + sdist(z, y)_{z \in X})$
5: end for
6: return $X_c$

Algorithm 4 Topdown with Pruning($X_c, X_p, u$)

Input: $G = (V, E)$, $X_c$ is the child node of $X_p$ in $T_G$, $X_p \subseteq X_c$, $sdist(u, X_p[i])_{0 \leq i < |X_p|}$
Output: $X_p \subseteq X_c$, $sdist(u, X_p[i])_{0 \leq i < |X_p|}$
1: $X = X_c \cap X_p$
2: copy $sdist(u, x)(x \in X)$ from $X_p$ to $X_c$
3: for $y \in X_c \setminus X$ do
4: $sdist(u, y) = \min(sdist(u, z) + sdist(z, y)_{z \in X})$
5: end for
6: return $X_p$

The above observation enables us to narrow down the search space during the traversal in the sense that the cardinality of the underlying bag is decreased according to the upper bound of the ordered list. Since there is overlapping between the bags, it is possible that one vertex occurs in both the parent and the child bags. Therefore, for instance, in a bottom-up transformation, if one vertex $v$ in the child bag $X_c$ has been removed, and $v$ occurs also in the parent bag $X_p$, then $v$ should be removed in $X_p$ as well. To achieve this, we introduce the modified version of Bottomup and Topdown, namely Bottomup with Pruning (see Algorithm 3) and Topdown with Pruning (see Algorithm 4). In Algorithm 3 we add $X'_c$ which is the subset of $X_c$ as the input parameter. Then in the parent bag $X_p$, we generate a subset $X'_p$ to ensure that the vertices that not in $X_c$ does not occur in $X'_p$ as well. Algorithm 4 is implemented analogously.
5.2 TIKNN Algorithm

In this section we explain the major steps of the TIKNN algorithm. The main algorithm is presented in Algorithm 7.

**S_Pruning.** The first step of the algorithm is to prune the unnecessary tree nodes in $T_G$. Given the graph $G = (V, E)$, $T_G$, and $S$, we start from each leaf node in $T_G$ in a bottom-up manner, until a vertex $v = X[0]$ is found in bag $X$. Then all the bags below $X$ will be pruned. The algorithm is shown in Algorithm 5.

**Algorithm 5 S_Pruning($T_G, S$)**

**Input:** $G = (V, E)$, $T_G$, $S$

**Output:** $T_G$ with useless bags pruned

1: for each leaf bag $X$ of $T_G$ do
2: while $X[0] \notin S$ and $X \neq R$ do
3: $X = X.parent$; remove $X$
4: end while
5: end for

We maintain a ranked list $\text{list}[i]_{0 \leq i \leq k}$ to store the current $k$ shortest distance. At the beginning of the algorithm the list is initialized with a maximal integer.

**Algorithm 6 Recur($X, X'$)**

1: for each non-visited child of $X$: $X_c$ do
2: $X'_c = \text{Topdown_with_Pruning}(X_c, X, X', u)$;
3: if $v = X_c[0] \in S$ and $\text{sdist}(u, v) < \text{list}[k]$ then
4: insert $\text{sdist}(u, v)$ to list;
5: end if
6: Recur($X_c, X'_c$);
7: /* update $\text{sdist}$ at $X'$ */
8: $v_i = X'[i]_{0 \leq i \leq |X'|};$
9: if $\text{sdist}(u, v_i) > \text{list}[k]$ then $X' = X' \setminus v_i$
10: end for

**Traversal.** The traversal process starts at $\text{RIS}(u)$ in $T_G$, which is assigned as $X$ in line 3 of Algorithm 4. First the the nodes in the subtree rooted at $\text{RIS}(u)$ are visited in a depth-first manner (see Algorithm 4). During the recursive call of the function of $\text{Topdown_with_Pruning}$, $X$ is pruned each time one child subtree is visited. After all the child nodes are visited, $X'$ (which is the subset of $X$) is returned. Then we traverse to the parent of $X$ using the function $\text{Bottomup_with_Pruning}$. At the parent node, the same top-down procedure is executed. The process continues until the root of the tree decomposition is reached.

5.3 Complexity

**Index size.** It is required that shortest distances be pre-computed and stored for every pair of vertices in the bags of the tree decomposition. However, there exist redundancies. Due to the property of the tree decomposition, there are vertex overlappings between the tree nodes. Note that this overlapping property is inherent thus can not be avoided. This storage redundancy can be avoided by storing the shortest distances into a hash table.

According to the decomposition algorithm proposed in [1], there are two parameters related to the index size, namely the root node size $|R|$, and $w$, which is the upper bound of the bag size other than the root $R$. Thus the overall space for the index structure is $O(L \cdot |R|^2)$, where $L$ is the average path length of the shortest paths in the tree decomposition.

**Query.** Both the bottom-up and top-down query processing between the bags takes time in the worst case $O(w^2)$, where $w$ is the upper bound of the bag size. Thus traversing the entire tree takes in worst case $O(Nw^2)$, where $N$ is the vertex size of the graph. However by applying the vertex pruning according to the value of $\text{list}(k)$ can reduce the size of $w$. The effectiveness of $S_Pruning$ decides the actual tree size after the pruning. In theory the smaller the value of $S$, the more effective the tree is pruned.

6. EXPERIMENTAL RESULTS

In this section we evaluate the TIKNN algorithm on one large scale road network graph [4] Northwest USA (denoted as USA-NW), which contains around 1.2 million vertices and 2.8 million edges.

All tests are run on an Intel(R) Core i5 1.7 GHz CPU, and 4 GB of main memory. All algorithms are implemented in C++ with the Standard Template Library (STL).

We are interested in the following parameters:

- Index size,
- Query time.

The space cost of the index structure consists of two parts:

(1) the size of the tree decomposition. This includes the tree structure and the vertices stored in the bags of the tree decomposition. (2) the local shortest paths stored in the hash table, which is dominant comparing to part (1).

Besides the standard measurements, we are also interested in the structure of the tree decomposition, which may influence the performance of the algorithm. These are:

- the number of tree nodes ($\#\text{TreeN}$),
- the number of all the vertices stored in the bags ($\#\text{SumV}$),
- the height of the tree ($h$),
- the number of vertex reductions ($w$).

• the root size of the tree after the tree decomposition \((|R|)\),
• the number of shortest distance pairs \((#P)\), and
• the average path length \((L)\).

Table 1 shows the parameters of the decomposed tree of the graph USA-NW. We applied the linear approximation algorithm introduced in [11]. Due to the property of the algorithm the relationship of \(N = #\text{TreeN} + |R|\) holds. That is, the number of the bags is always less than the size of the vertices in \(G\). However, this relationship does not hold in general. As mentioned in the last section, the relationship of \(w < |R|\) holds, where \(w\) is the maximal cardinality of the bags other than the root.

| #TreeN | #SumV | h | w | |R| | #P | L |
|--------|-------|---|---|-----|-----|-----|-----|
| 1 207 383 | 4 285 201 | 424 | 80 | 562 | 3 248 245 | 55 |

Table 1: Statistics of USA-NW and properties of the tree decomposition based index structure

The number of the shortest distance pairs \((#P)\) and the average path length \((L)\) are the factors deciding the index size. In fact, the number of overall stored shortest paths is \((#P \cdot L)\), which is approximately 176MB, as shown in Table 2. In contrast, the approach proposed in [7] requires \(N^{1.5}\) for storing the index, which is more than 1.3GB regarding our example graph USA-NW. Clearly index size of our approach is approximately ten times smaller than [7].

<table>
<thead>
<tr>
<th>Index Size (MB)</th>
<th>Index Time (min)</th>
</tr>
</thead>
<tbody>
<tr>
<td>paths tree kNN</td>
<td>(t_{\text{tree}}) (t_{\text{paths}}) kNN</td>
</tr>
<tr>
<td>175.6 7.8 183.4 9.5 260.7 270.2</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Index construction: size and time

Next we conduct the comparison on the performance of our algorithm and Dijkstra’s algorithm. Figure 3 shows the execution time \((a)\) and the number of visited nodes \((b)\) on the graph of USA-NW, with \(S\) to be fixed as \((2/10000)N\), which contains in this case about 240 vertices. The number of neighbors \(k\) varies from 10 to 90. First of all, as \(k\) increases, both the time consumption and the number of visited vertices tend to increase on both algorithms. Note that for Dijkstra’s algorithm, both the time consumption and the number of visited nodes increase linearly. Notably TIKNN is two magnitude faster than Dijkstra’s algorithm. This is because as \(|S|\) is of small value, the density of \(S\) values in \(G\) is low. Thus in average, Dijkstra’s algorithm would meet one \(S\) vertex after processing approximately 5000 vertices. Hence as \(k\) grows, the number of vertices that have to be processed increases linearly. One the other hand, the smaller the value of \(|S|\), the more effective \(S\_Pruning\) of the TIKNN algorithm. This test setting shows that for the kNN problems with small \(S\) cardinality, TIKNN gains the most performance improvement in comparison to Dijkstra’s algorithm.

Figure 4 shows the execution time \((a)\) and the number of visited nodes \((b)\) on the graph of USA-NW, with \(S\) to be fixed as \((50/10000)N\), which contains in this case about 6000 vertices. This configuration is in favor of Dijkstra’s algorithm, since the density of \(S\) values in \(G\) is much higher than the previous case. In fact, approximately after every 200 vertices are processed, an \(S\) vertex would be met. Therefore Dijkstra’s algorithm is more efficient on both the time consumption and the number of visited nodes comparing to its performance in Figure 3. Nevertheless, our algorithm is one magnitude faster.

In the last set of experiments, we fix the number of neighbors \(k\) as 10, and vary the cardinality of \(S\) from \((2/10000)N\) to \((50/10000)N\). Analogously, we measure the time consumption and the number of visited vertices. As \(|S|\) is increasing, the performance improves on both algorithms. The improvement on Dijkstra’s algorithm is straightforward. As far as TIKNN algorithm is concerned, we can explain the performance improvement as follows. As the cardinality of \(S\) increases, the density of the corresponding tree nodes in \(T_G\) increases as well. Thus during the traversal over the tree decomposition, the number of processed tree nodes between two \(S\) vertices decreases. As a consequence, if \(k\) is fixed, the upper bound (that is, the value \(\text{list}(k)\)) decreases faster. This results in a more effective pruning on the tree nodes during the traversal. Our algorithm is approximately one magnitude faster than Dijkstra’s algorithm.

7. CONCLUSIONS

A novel algorithm for computing \(k\) nearest neighbor over a weighted, undirected graph is proposed in this paper. One property that makes this problem interesting is that we predefine a vertex set \(S \subseteq V\) such that all the \(kNN\) vertices are retrieved from \(S\), instead of the entire \(V\). One observation is that if \(S = V\), then Dijkstra’s algorithm is optimal. Therefore, our algorithm is most suitable for queries with \(S\) of small size. Experiments shows that our algorithm offers at least one magnitude improvement to Dijkstra’s algorithm, while the pre-computed index size is moderate.

8. REFERENCES


Figure 3: Comparison of TIkNN and Dijkstra algorithm for $S = (2/10000)N$ and varying sizes of $k$ (a) query time in ms (b) number of visited nodes

Figure 4: Comparison of TIkNN and Dijkstra algorithm for $S = (50/10000)N$ and varying sizes of $k$ (a) query time in ms (b) number of visited nodes

Figure 5: Comparison of TIkNN and Dijkstra algorithm for $k = 10$ and varying sizes of $S$ (a) query time in ms (b) number of visited nodes