G-Tree: An Efficient Index for KNN Search on Road Networks

Ruicheng Zhong[†], Guoliang Li[†], Kian-Lee Tan[‡], Lizhu Zhou[†]
[†]Dept. of Computer Science, Tsinghua University, Beijing, China
[‡]Dept. of Computer Science, National University of Singapore, Singapore
zrc1101001@gmail.com, {liguoliang,dcszlz}@tsinghua.edu.cn,
tankl@comp.nus.edu.sg

ABSTRACT

In this paper we study the problem of kNN search on road networks. Given a query location and a set of candidate objects in a road network, the kNN search finds the k nearest objects to the query location. To address this problem, we propose a balanced search tree index, called `G-tree`. The `G-tree` of a road network is constructed by recursively partitioning the road network into sub-networks and each `G-tree` node corresponds to a sub-network. Inspired by classical kNN search on metric space, we introduce a best-first search algorithm on road networks, and propose an elaborately-designed assembly-based method to efficiently compute the minimum distance from a `G-tree` node to the query location. `G-tree` only takes $\mathcal{O}(|V|\log|V|)$ space, where $|V|$ is the number of vertices in a network, and thus can easily scale up to large road networks with more than 20 millions vertices. Experimental results on eight real-world datasets show that our method significantly outperforms state-of-the-art methods, even by 2-3 orders of magnitude.

Categories and Subject Descriptors

H.2.8 [**Database Applications**]: Spatial databases

Keywords

KNN search, Road network, Spatial databases

1. INTRODUCTION

Mobile devices (e.g., smartphones) have become more and more popular in our daily life. To provide users with location-based search experiences, location-based service(LBS) systems(e.g., Foursquare and Google Maps for Mobile) have been widely deployed and accepted by mobile users.

K nearest neighbor (kNN) search on road networks is a fundamental problem in LBS. Given a query location and a set of static objects (e.g., gas stations) on the road network, the kNN search problem finds k nearest objects to the query location. kNN search on road networks has many real-world applications. For example, a tourist looking for k nearest "`gas stations`" while driving in a city requires a kNN query. As another example, an ambulance searching for k nearest "`hospitals`" in an emergency case also requires a kNN query.

CIKM'13, Oct. 27–Nov. 1, 2013, San Francisco, CA, USA.

http://dx.doi.org/10.1145/2505515.2505749.

There are a large number of studies on kNN search on road networks [3, 8, 9, 14, 15, 16, 20, 23]. However, existing methods still cannot support very large road networks (e.g. the whole USA road network). The main limitation of these approaches is either high memory consumption or heavy search overhead. Consider the state-of-the-art approaches, *SILC* [23] and *ROAD* [15, 16], *SILC* requires $\mathcal{O}(|V|^{1.5})$ space to store all-pair shortest paths and *ROAD* employs Dijkstra-like algorithm for kNN finding, which has very poor scalability and efficiency on large road networks. For example, for the whole USA dataset(24M vertices), we estimate that *ROAD* needs over 105 days for pre-processing, and *SILC* consumes approximately 618GB memory!

In this paper, our goal is to design an elegant index which supports efficient kNN search on large road networks. Inspired by the classical `R-tree` on Euclidean space, we design our index on road networks by considering two core features. The first one is a balance tree structure, and we propose a balanced search tree index, called `G-tree`. The `G-tree` of a road network is constructed by recursively partitioning the road network into sub-networks and each `G-tree` node corresponds to a sub-network. The second one is to enable best-first search on such tree-based index, since the best-first algorithm has been widely applied and shown to be superior in performance [7]. However, it is non-trivial to devise the best-first search algorithm on `G-tree` since it is not easy to efficiently calculate the graph distance between the query location and a tree node, which is an essential operation in the best-first search algorithm. To address this issue, we define a shortest-path distance function which returns the minimum distance between a tree node and the query location, and propose an elaborately-designed *assembly-based method* to efficiently implement this function. The assembly-based method employs a dynamic-programming algorithm rather than the conventional network-expansion search algorithm, thus significantly reduces the overhead for calculating this function. `G-tree` only takes $\mathcal{O}(|V|\log|V|)$ space, hence, it can easily handle very large road networks, even with more than 20 million vertices. Experiments on eight real-world datasets show that our method significantly outperforms state-of-the-art methods, even by 2-3 orders of magnitude. To summarize, we make the following contributions.

- We propose a balanced search tree index, `G-tree`, which has high pruning power and a small index size.
- Based on the `G-tree`, we propose the dedicated assembly-based method to efficiently compute the minimum distance between the query location and a `G-tree` node.

Then, we devise a best-first search algorithm to retrieve the top-k answers on road networks.

- Our method has theoretical and practical superiority over existing methods. To our best knowledge, this is the first work challenging large data sets with more than 20M vertices for kNN search on road network.

The structure of this paper is organized as follows. We define the kNN search problem and review related works in Section 2. We present our `G-tree` index and and an efficient search algorithm in Section 3&4. We discuss the path recovery, `G-tree` construction, maintenance and extension issues in Section 5. Experiment results are reported in Section 6.

2. PRELIMINARIES

2.1 Problem Formulation

Data Model. We model a road network as an undirected weighted graph $\mathcal{G} = \langle \mathcal{V}, \mathcal{E} \rangle$, where $\mathcal{V}$ is a set of vertices and $\mathcal{E}$ is a set of edges. Each edge (u, v) in $\mathcal{E}$ has a weight (e.g., distance, travel time, etc.), which is a positive value. Given a path between vertex u and vertex v, the sum of weights of edges along the path is called the distance of the path. A path with the shortest distance is called a shortest path. Let $\texttt{SP}(u, v)$ denote a shortest path between u and v, and $\texttt{SPDist}(u, v)$ denote the shortest-path distance between u and v. We will discuss how to extend our method to support directed graphs in Section 5.4.

For example, Figure 1 shows a road network. The weight of edge (v_2, v_6) is 3. $\texttt{SP}(v_4, v_9) = v_4v_3v_2v_6v_7v_8v_9$ is a shortest path between v_4 and v_9 and $\texttt{SPDist}(v_4, v_9) = 15$.

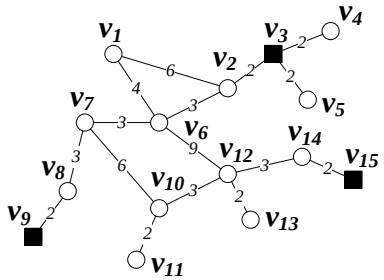


Figure 1: A road network.

Query Model. Given a graph $\mathcal{G}$, a query q is a triple $q = \langle v_q, \mathcal{C}, k \rangle$, where v_q is a query location, $\mathcal{C}$ is a subset of vertices in $\mathcal{G}$ (e.g., gas stations), and k is an integer. Each vertex in $\mathcal{C}$ is called an object.

Top-k Answers. Given a graph $\mathcal{G}$ and a query $q = \langle v_q, \mathcal{C}, k \rangle$, the answer, denoted by $\mathcal{R}$, is a set of k nearest objects to the query location such that,

(1) The size of $\mathcal{R}$ is k, i.e., $|\mathcal{R}| = k$;

(2) Each answer is an object, i.e., $\mathcal{R} \subseteq \mathcal{C}$;

(3) $\forall v \in \mathcal{R}, \forall u \in \mathcal{C} - \mathcal{R}$, $\texttt{SPDist}(v_q, v) \leq \texttt{SPDist}(v_q, u)$.

For simplicity, in the paper we assume that both the query location and the objects are at vertices. If not, we can use heuristic method, e.g. to place them on the nearest vertex.

For example, in Figure 1, consider $q = \langle v_4, \{v_3, v_9, v_{15}\}, 2 \rangle$. There are three objects (which are denoted by solid rectangles). We have $\texttt{SPDist}(v_4, v_3) = 2$, $\texttt{SPDist}(v_4, v_9) = 15$ and $\texttt{SPDist}(v_4, v_{15}) = 21$. The top-2 answers are $\mathcal{R} = \{v_3, v_9\}$.

2.2 Related Works

Existing studies [15, 16, 20, 23] addressed the same problem as ours. *INE* [20] extended the Dijkstra algorithm [4] by expanding neighbor vertices from the query location until kNN answers have been found. *IER* [20] improved *INE* by utilizing spatial pruning techniques, e.g, taking the Euclidean distance as a bound, to prune unpromising expansions. *IER* and *INE* are 'blind' algorithms since they can neither capture the global distance from objects to the query location nor prune unnecessary objects efficiently.

ROAD [15, 16] also extended the Dijkstra algorithm by using a hierarchical structure. *ROAD* recursively partitions a road network into sub-networks, pre-computes the shortest-path distances of "shortcuts" within a sub-network, and organizes them in a hierarchical manner. By using Dijkstra-like network expansion, *ROAD* can skip sub-networks which do not contain an object. However it cannot prune sub-networks with objects which are widely scattered. For example, if the objects are uniformly distributed (e.g., Mcdonald's or gas stations), *ROAD* will degenerate to the Dijkstra algorithm and have to traverse the whole network. Thus *ROAD* performs poorly, especially on large networks. We call *ROAD* a 'half-blind' algorithm as it partially captures global distance information.

Although *ROAD* uses a hierarchical structure, it is different from our method as follows. First, the tree structures are different. `G-tree` is a balanced search tree while *ROAD* is not. Second, the kNN finding paradigms are fundamentally different. *ROAD* employs an expansion-based method and cannot utilize the global distance information, e.g., the shortest-path distance from a query location to tree nodes, to do effective pruning. `G-tree` adopts a best-first search algorithm which only accesses tree branches containing objects and thus reduces the space space significantly. Thus our method significantly outperforms *ROAD* (see Section 6).

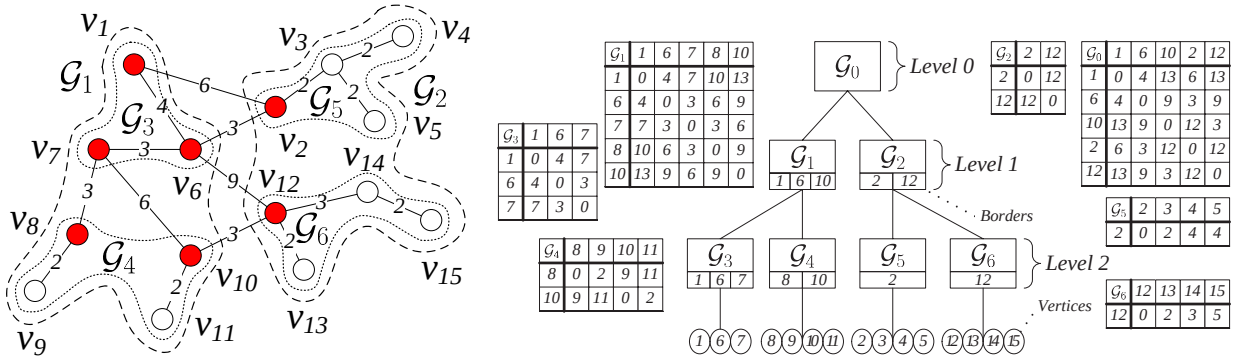
SILC [23] pre-computes the shortest paths between all vertex pairs and uses a quadtree-based encoding to store the shortest paths. It utilizes the materialized pairs to find k nearest neighbors by using Euclidean distance and stores the shortest-path distance as a bound. However if there are large numbers of objects clustered in a small region, *SILC* is inefficient. Moreover, *SILC* consumes $\mathcal{O}(|\mathcal{V}|^{1.5})$ storage space and incurs high pre-processing overhead, and thus it is impractical for large road networks.

There are some studies which assume that the object set is given [3, 8, 9, 14]. They pre-compute and materialize results of potential queries on the graph. However these approaches highly depend on the given object set. In addition they involve high pre-computation cost and large memory overhead.

[21, 17, 28] studied spatial keyword search on road networks and in metric space. Their problem is different from ours. They focused on how to combine keyword information and distance information to compute top-k answers, while we emphasized on finding k nearest neighbors.

kNN Queries for Moving Objects on Road Networks: There are quite a number of studies on kNN queries for moving objects monitoring [10, 19, 26, 27]. These works studied the problem of finding nearest moving objects (e.g., taxies) to a location and focused on dealing with frequent updates of moving objects. In our case, we emphasize on the efficiency of the kNN queries on the static objects (e.g., gas stations), of which the ideas and implementations are totally different.

Single-Pair Shortest Path Queries: Many previous studies [1, 2, 6, 11, 12, 24, 25] addressed the problem of shortest



(a) Graph partition. (b) G-tree.

Figure 2: A G-tree (The solid vertices in the graph are borders. $f = 2$ and $\tau = 4$.).

path queries between two vertices on road networks. However, it is worth noting that kNN search is radically different from single-pair shortest path solution. Though kNN search involves shortest path computation, the key point of kNN search is to quickly find those promising top-k objects rather than to calculate the shortest path from the query location to all candidate objects then rank them. Therefore, it is not feasible to apply them to handle the kNN search on road networks effectively. Although HEPV [11] and HiTi [12] also organize the road network into a hierarchical structure, they still use 'half-blind' Dijkstra-like network expansion, while G-tree is fundamentally different which fully utilizes the best-first search and employs dedicated assembly-based method to calculate the distance between the query location to a tree node.

3. GRAPH TREE

In Euclidean space, tree structured indices, e.g., R-tree, have salient features to support kNN search. We would like to incorporate two of these features in our G-tree to support kNN search on road networks. The first feature is the balanced tree structure that can help to prune subtrees. We will discuss it in this section. The second feature is the efficient computation of the minimum distance from the query location to tree nodes which is used for best-first search, which will be discussed in Section 4.

3.1 G-tree

Before we present the G-tree structure, for ease of presentation, we first introduce some important concepts which will be used throughout the paper.

DEFINITION 1 (GRAPH PARTITION). *Given a graph $\mathcal{G} = \langle \mathcal{V}, \mathcal{E} \rangle$, where $\mathcal{V}$ is the vertex set and $\mathcal{E}$ is the edge set of $\mathcal{G}$, a partition of $\mathcal{G}$ is a set of subgraphs, i.e., $\mathcal{G}_1 = \langle \mathcal{V}_1, \mathcal{E}_1 \rangle, \mathcal{G}_2 = \langle \mathcal{V}_2, \mathcal{E}_2 \rangle, \ldots, \mathcal{G}_f = \langle \mathcal{V}_f, \mathcal{E}_f \rangle$ such that*

(1) $\bigcup_{1 \le i \le f} \mathcal{V}_i = \mathcal{V}$,

(2) For $i \neq j$, $\mathcal{V}_i \cap \mathcal{V}_j = \emptyset$, and

(3) $\forall u, v \in \mathcal{V}_i$, if $(u, v) \in \mathcal{E}$, then $(u, v) \in \mathcal{E}_i$.

Vertices in different subgraphs may be connected in the original graph $\mathcal{G}$ but be separated in different subgraphs after partitioning. To differentiate such vertices from others, we define a concept, called *borders*.

DEFINITION 2 (BORDERS). *Given a subgraph $\mathcal{G}_i$ of $\mathcal{G}$, a vertex $u \in \mathcal{V}_i$ is called a border if $\exists (u, v) \in \mathcal{E}$ and $v \notin \mathcal{V}_i$. We use $\mathcal{B}(\mathcal{G}_i)$ to denote the border set in graph $\mathcal{G}_i$.*

A subgraph $\mathcal{G}_i$ is called a super-graph of another subgraph $\mathcal{G}_j$ if $\mathcal{V}_i \supseteq \mathcal{V}_j$ and $\mathcal{E}_i \supseteq \mathcal{E}_j$. Based on these concepts, next we formally define the G-tree structure.

DEFINITION 3. *A G-tree is a balanced search tree that satisfies the following properties.*

(1) Each node represents a subgraph. The root node corresponds to the graph $\mathcal{G}$. The subgraph of a parent node is a super-graph of those of its child nodes.

(2) Each non-leaf node has $f (\ge 2)$ children.

(3) Each leaf node contains at most $\tau (\ge 1)$ vertices. All leaf nodes appear at the same level.

(4) Each node maintains its border set and a distance matrix. In the distance matrix of a non-leaf node, the columns/rows are all borders in its children and the value of each entry is the shortest-path distance between the two borders. In the distance matrix of a leaf node, the rows are all borders in the node, columns are all vertices in the node, and the value of each entry is the shortest-path distance between the border and the vertex.

Conditions (1)-(3) ensure that the G-tree has a balanced search tree structure. It is worth noting that for each node we do not maintain the physical subgraph. Instead, we only maintain a dummy subgraph ID. As there is a one-to-one correspondence between a node and a subgraph, for simplicity, "nodes" and "subgraphs" are interchangeably used if the context is clear. In the paper "nodes" refer to G-tree nodes and "vertices" refer to vertices in the graph. Condition (4) is used to efficiently compute the shortest-path distance from a vertex u to a vertex v, i.e., $\texttt{SPDist}(u, v)$. We will use it to compute the minimal distance from a vertex u to a node n, i.e., $\texttt{SPDist}(u, n) = \min\{\texttt{SPDist}(u, v) | v \text{ is a vertex in } n\}$, which will be discussed in Section 4.

EXAMPLE 1. Figure 2(b) shows the G-tree of the road network in Figure 2(a). The borders of each node are shown in the rectangle box under the node. For example, $\mathcal{G}_1$ has three borders $\{v_1, v_6, v_{10}\}$. The distance matrix of each node is listed around the tree node. For $\mathcal{G}_1$, its children $\mathcal{G}_3$ and $\mathcal{G}_4$ contain five borders $\{v_1, v_6, v_7, v_8, v_{10}\}$, thus the rows/colmuns of $\mathcal{G}_1$'s distance matrix are the five borders. The set of vertices of each leaf node are shown in the circled numbers. For instance, $\mathcal{G}_4$ contains two borders $\{v_8, v_{10}\}$ and four vertices $\{v_8, v_9, v_{10}, v_{11}\}$. In $\mathcal{G}_4$'s distance matrix, the rows are borders $\{v_8, v_{10}\}$ and the columns are vertices $\{v_8, v_9, v_{10}, v_{11}\}$. The entry $(v_8, v_{11}) = 11$ since the shortest distance between border v_8 and v_{11} is 11.

3.2 G-tree Construction

We use a graph partition based method to build the G-tree. Initially, we take the graph $\mathcal{G}$ as the root. Then we partition $\mathcal{G}$ into f equal-sized subgraphs (i.e., $|\mathcal{V}_1| \approx \ldots \approx |\mathcal{V}_f|$) and take them as the root's children. Next we recursively partition the children and repeat this step until each

leaf-node's subgraph has no more than τ vertices. Notice that during the partitioning, for each subgraph, we will add its borders into the corresponding node. For example, in Figure 2(a), suppose $f = 2$ and $\tau = 4$, the original graph $\mathcal{G}_0$ is partitioned into two subgraphs $\mathcal{G}_1$ and $\mathcal{G}_2$. $\mathcal{G}_1$ is further partitioned into $\mathcal{G}_3$ and $\mathcal{G}_4$. $\mathcal{G}_2$ is partitioned into $\mathcal{G}_5$ and $\mathcal{G}_6$.

Graph partitioning is an important step in `G-tree` construction. The optimal one should not only generate approximately equal-sized subgraphs, but also minimize the number of borders. However, it has been proven that the optimal graph partitioning is NP-Hard [5]. In this paper, we adopt a famous heuristics algorithm, called the multilevel partitioning algorithm [13]. The multilevel partition algorithm can guarantee that each subgraph nearly has the same size and thus `G-tree` is a balanced search tree.

For distance matrices of `G-tree`, we need to compute the shortest-path distance between a border and a border/vertex(non-leaf/leaf). We can use a single source shortest-path algorithm, e.g. Dijkstra algorithm, to compute the graph distance. It starts from each border/vertex within one `G-tree` node, expands the edges until if all borders of such node have been reached. In Section 5.2, we will introduce an efficient bottom-up algorithm to speed up this procedure.

Notice that we focus on the memory-based index in this paper and leave the disk-based index as our future work.

3.3 Space Complexity of The `G-tree`

Height: The height of the `G-tree` is $\mathcal{H} = log_f \frac{|\mathcal{V}|}{\tau} + 1$.

Number of Nodes: At level 0, there is one node (the root). In level i, there are f^i nodes. There are $\frac{|\mathcal{V}|}{\tau}$ leaf nodes. Thus the total number of nodes is $\mathcal{O}(\frac{f}{f-1}\frac{|\mathcal{V}|}{\tau}) = \mathcal{O}(\frac{|\mathcal{V}|}{\tau})$.

Number of Borders: A road network is usually modeled as a planar graph [23]. We also consider the planar graph in the space analysis. Consider a node on the i-th level. Its borders are generated by its parent which has $|\mathcal{V}|/f^{i-1}$ vertices. According to the Planar Separator Theorem [18], the f-partition on the parent totally involves $\mathcal{O}(\log_2 f \cdot \sqrt{|\mathcal{V}|/f^{i-1}})$ borders. Hence, the total number of borders in the `G-tree` is $\mathcal{O}(\sum_{1\le i\le \mathcal{H}} \log_2 f \cdot \sqrt{|\mathcal{V}|f^{i-1}}) = \mathcal{O}(\frac{\log_2 f}{\sqrt{\tau}}|\mathcal{V}|)$.

Distance Matrix: The average number of borders in a leaf node is $\mathcal{O}(\log_2 f \cdot \sqrt{|\mathcal{V}|/f^{\mathcal{H}+1}}) = \mathcal{O}(\log_2 f \cdot \sqrt{\tau})$. Thus, the total distance-matrix size of all leaf nodes is $\mathcal{O}(\log_2 f \cdot \tau^{1.5}\frac{|\mathcal{V}|}{\tau}) = \mathcal{O}(\log_2 f \cdot \sqrt{\tau}|\mathcal{V}|)$. For each non-leaf node, the rows/columns of its distance matrix are the union of borders in its children. Each node on level i generates $\mathcal{O}(\log_2 f \cdot \sqrt{|\mathcal{V}|/f^i})$ borders. Thus the matrix size of each node at level i is $\mathcal{O}(\log_2^2 f \cdot |\mathcal{V}|/f^i)$. As there are f^i nodes at level i, the distance-matrix size at level i is $\mathcal{O}(\log_2^2 f \cdot |\mathcal{V}|)$. Hence the total matrix size of non-leaf nodes is $\mathcal{O}(\mathcal{H}\log_2^2 f \cdot |\mathcal{V}|) = \mathcal{O}(\log_2^2 f \cdot \log_f \frac{|\mathcal{V}|}{\tau} \cdot |\mathcal{V}|)$.

Overall Space: The overall size of the `G-tree` is $\mathcal{O}(\frac{|\mathcal{V}|}{\tau} + \frac{\log_2 f}{\sqrt{\tau}}|\mathcal{V}| + \log_2 f \cdot \sqrt{\tau}|\mathcal{V}| + \log_2^2 f \cdot \log_f \frac{|\mathcal{V}|}{\tau} \cdot |\mathcal{V}|) = \mathcal{O}(\log_2 f \cdot \sqrt{\tau}|\mathcal{V}| + \log_2^2 f \cdot \log_f \frac{|\mathcal{V}|}{\tau} \cdot |\mathcal{V}|)$. It is worth noting that $\log_2^2 f$, $\sqrt{\tau}$ and $log_f \frac{|\mathcal{V}|}{\tau}$ are small, thus `G-tree` is scalable.

4. SEARCH ALGORITHM

In this section, using the `G-tree` index, we propose a best-first kNN search algorithm for road networks.

4.1 Algorithm Overview

The basic idea of our algorithm is as follows. Suppose we can use function $\texttt{SPDist}(v_q, n)$ to compute the minimum distance between the query location v_q and a tree node n. Given a query $q = \langle v_q, \mathcal{C}, k\rangle$, we first locate the leaf nodes of query location and objects using a hash table which maps a vertex to the corresponding leaf node. For each leaf node n found, we construct an occurrence list $\mathcal{L}(n)$ which is composed of IDs of objects that appear in the leaf node (line 1). Then for each ancestor n_a of such a leaf node, we also construct an occurrence list $\mathcal{L}(n_a)$, which is composed of IDs of n_a's children that contain objects. Thus from the root, we can easily figure out which nodes contain objects based on the occurrence list. Figure 7(a) illustrates an example for $\mathcal{C} = \{v_3, v_9, v_{15}\}$. For example, for $\mathcal{G}_5$, its occurrence list is $\{v_3\}$ as vertex v_3 is an object. For $\mathcal{G}_2$, its occurrence list is $\{\mathcal{G}_5, \mathcal{G}_6\}$ as nodes $\mathcal{G}_5$ and $\mathcal{G}_6$ contain objects.

Then we initialize a priority queue $\mathcal{Q}$ and a result set $\mathcal{R}$ (line 2). Initially we put $\langle root, \texttt{SPDist}(v_q, root) = 0\rangle$ into $\mathcal{Q}$ (line 3). We iteratively dequeue the first element e of $\mathcal{Q}$ and handle it separately according to whether e is an object, a leaf node or a non-leaf node(line 4 to line 18). Figure 3 shows the pseudo-code of our algorithm.

It is worth noting that if $\mathcal{R}$ has k answers, the algorithm can safely terminate. This is because the distance from the query location v_q to the kth answer is currently the best. Thus our algorithm can correctly find the top-k answers.

EXAMPLE 2. *Consider the query $q = \{v_4, \{v_3, v_9, v_{15}\}, 2\}$ on the graph in Figure 1. We first construct the occurrent list and then compute the top-2 answers as follows.*

Step 1: *Enqueue the root node, i.e. $\langle \mathcal{G}_0, 0\rangle$.*
Queue: $\boxed{\langle \mathcal{G}_0, 0\rangle}$

Step 2: *Dequeue $\langle \mathcal{G}_0, 0\rangle$. Find two child nodes in the occurrence list of $\mathcal{G}_0$, i.e. $\mathcal{G}_1$ and $\mathcal{G}_2$. Get $\texttt{SPDist}(v_4, \mathcal{G}_1) = 7$ and $\texttt{SPDist}(v_4, \mathcal{G}_2) = 0$. Enqueue $\mathcal{G}_2, \mathcal{G}_1$.*
Queue: $\boxed{\langle \mathcal{G}_2, 0\rangle \mid \langle \mathcal{G}_1, 7\rangle}$

Step 3: *Dequeue $\langle \mathcal{G}_2, 0\rangle$. Find two child nodes in the occurrence list of $\mathcal{G}_2$, i.e. $\mathcal{G}_5$ and $\mathcal{G}_6$. Get $\texttt{SPDist}(v_4, \mathcal{G}_5) = 0$ and $\texttt{SPDist}(v_4, \mathcal{G}_6) = 16$. Enqueue $\mathcal{G}_5, \mathcal{G}_6$.*
Queue: $\boxed{\langle \mathcal{G}_5, 0\rangle \mid \langle \mathcal{G}_1, 7\rangle \mid \langle \mathcal{G}_6, 16\rangle}$

Step 4: *Dequeue $\langle \mathcal{G}_5, 0\rangle$. Find a vertex $v_3 \in \mathcal{G}_5$. Get $\texttt{SPDist}(v_4, v_3) = 2$. Enqueue v_3.*
Queue: $\boxed{\langle v_3, 2\rangle \mid \langle \mathcal{G}_1, 7\rangle \mid \langle \mathcal{G}_6, 16\rangle}$

Step 5: *Dequeue $\langle v_3, 2\rangle$. $\mathcal{R} = \{v_3\}$.*
Queue: $\boxed{\langle \mathcal{G}_1, 7\rangle \mid \langle \mathcal{G}_6, 16\rangle}$

Step 6: *Dequeue $\langle \mathcal{G}_1, 7\rangle$. Find a child in the occurrence list of $\mathcal{G}_1$, i.e., $\mathcal{G}_4$. Get $\texttt{SPDist}(v_4, \mathcal{G}_4) = 13$ and enqueue $\mathcal{G}_4$.*
Queue: $\boxed{\langle \mathcal{G}_4, 13\rangle \mid \langle \mathcal{G}_6, 16\rangle}$

Step 7: *Dequeue $\langle \mathcal{G}_4, 13\rangle$. Find a vertex $v_9 \in \mathcal{G}_4$. Get $\texttt{SPDist}(v_4, v_9) = 15$. Enqueue v_9.*
Queue: $\boxed{\langle v_9, 15\rangle \mid \langle \mathcal{G}_6, 16\rangle}$

Step 8: *Dequeue v_9. $\mathcal{R} = \{v_3, v_9\}$. Top-2 answers have been generated. Algorithm terminates.*

The biggest challenge in the algorithm is to efficiently implement $\texttt{SPDist}(v_q, e)$ under three circumstances: MINDIST-INSIDE-LEAF, MINDIST-OUTSIDE-LEAF, and MINDIST-OUTSIDE-NONLEAF. To address this problem, we will present our dedicated assembly-based method in next section.

4.2 Implementing `SPDist` Function

One significant issue remained in Algorithm 1 is to calculate the $\texttt{SPDist}(v_q, e)$ where e is a vertex or node. This is the most important part in the framework of `G-tree`, since it will greatly affect the efficiency of kNN search. In this section, we discuss how to efficiently implement the function

Algorithm 1: KNNSearch ($q = \langle v_q, \mathcal{C}, k\rangle$, $\mathcal{G}$)

Input: $q = \langle v_q, \mathcal{C}, k\rangle$: A query; $\mathcal{G}$: A `G-tree`
Output: $\mathcal{R}$: The top-k result list;

```
1  Compute the occurrence list L based on C ;
2  Initialize priority queue Q = φ and result set R = φ ;
3  Q.Enqueue(⟨G.root, 0⟩) ;
4  while R.Size() < k & Q is not empty do
5      e ← Q.Dequeue() ;
6      if e is an object then Insert e into R;
7      else if e is a leaf node then
8          if v_q ∈ e then MinDist-Inside-Leaf (v_q, e);
9          else MinDist-Outside-Leaf (v_q, e) ;
10         foreach v ∈ L(e) do
11             Q.Enqueue(⟨v, SPDist(v_q, v)⟩) ;
12     else if e is a non-leaf node then
13         foreach child node c ∈ L(e) do
14             if v_q is in c then
15                 Q.Enqueue(⟨c, SPDist(v_q, c) = 0⟩) ;
16             else
17                 MinDist-Outside-NonLeaf (v_q, c) ;
18                 Q.Enqueue(⟨c, SPDist(v_q, c)⟩) ;
```

Figure 3: KNNSearch Algorithm

$\texttt{SPDist}(v_q, e)$. First, we introduce two naive methods, and then we present our dedicated *assembly-based method*.

Naive Method 1: A naive solution is to pre-calculate the minimum distances between all vertices and nodes/vertices. In this case, we can use $\mathcal{O}(1)$ time to implement the `SPDist` function. Obviously the space complexity is $\mathcal{O}(|\mathcal{V}|^2 + \frac{|\mathcal{V}|}{\tau}|\mathcal{V}|) = \mathcal{O}(|\mathcal{V}|^2)$. Apparently when $|\mathcal{V}|$ becomes large, this method may incur an unacceptable memory cost and is not scalable.

Naive Method 2: Another naive method is to utilize borders based on the following "*closure*" property.

LEMMA 1 (CLOSURE). *Given a subgraph $\mathcal{G}_i = \langle \mathcal{V}_i, \mathcal{E}_i\rangle$, for any vertex $u \notin \mathcal{V}_i$ and $v \in \mathcal{V}_i$, any shortest path between u and v must contain a border in $\mathcal{B}(\mathcal{G}_i)$, i.e., for any shortest path $\texttt{SP}(u, v)$, $\exists w \in \mathcal{B}(\mathcal{G}_i), w \in \texttt{SP}(u, v)$.*

PROOF. We omit the proof due to space constraints. □

As shown in Figure 2(a), consider $v_9 \in \mathcal{G}_1$ and $v_4 \in \mathcal{G}_5$. As v_4 and v_9 are not within the same subgraph, any path from v_4 to v_9 must contain a border in $\mathcal{G}_1$, e.g., v_6. Similarly, any path must contain a border in $\mathcal{G}_4$, e.g., v_8.

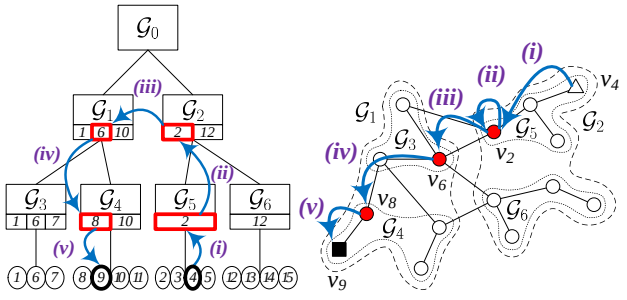
Consider a query location v_q, a vertex v, and $v_q \notin \texttt{leaf}(v)$, where $\texttt{leaf}(v)$ denotes the leaf node of v. Based on Lemma 1, we can decompose $\texttt{SPDist}(v_q, v)$ into two sub-paths. The first one is from v_q to $\mathcal{B}(\texttt{leaf}(v))$ and the second one is from $\mathcal{B}(\texttt{leaf}(v))$ to v. Thus we have

$$\texttt{SPDist}(v_q, v) = \min_{w \in \mathcal{B}(\texttt{leaf}(v))} \big(\texttt{SPDist}(v_q, w) + \texttt{SPDist}(w, v)\big). \tag{1}$$

Similarly, given a query location v_q and a tree node n such that $v_q \notin n$, we have

$$\texttt{SPDist}(v_q, n) = \min_{w \in \mathcal{B}(n)} \texttt{SPDist}(v_q, w). \tag{2}$$

Based on this property, we propose another naive method. We pre-compute and store the shortest-path distances of all vertex-border pairs. Thus we can efficiently implement MINDIST-OUTSIDE-LEAF and MINDIST-OUTSIDE-NONLEAF. The space complexity is $\mathcal{O}(|\mathcal{V}|\log_2 f \cdot \frac{|\mathcal{V}|}{\sqrt{\tau}} = \frac{\log_2 f}{\sqrt{\tau}}|\mathcal{V}|^2)$. To support MINDIST-INSIDE-LEAF, we need to pre-compute and store shortest-path distance between all vertexes in the same leaf node. The space complexity is $\mathcal{O}(\tau|\mathcal{V}|)$. Thus the total space complexity is $\mathcal{O}(\frac{\log_2 f}{\sqrt{\tau}}|\mathcal{V}|^2 + \tau|\mathcal{V}|)$. Although this method reduces the storage space, it is still not scalable to large graphs.



(a) Assembling on `G-tree`. (b) Assembling on Graph.

Figure 4: An Assembly-based Method.

Assembly-based Method: We have an observation that many shortest paths share common sub-paths, and we do not need to store shortest-path distances for all pairs between vertices and borders. Instead we only materialize some pairs and assemble these pairs to implement the `SPDist` function. We use an example to show our idea. For example, in Figure 2(a), consider two vertices v_4, v_5 and a border v_8. The shortest path from v_4 to v_8 is $v_4v_3v_2v_6v_7v_8$ and the shortest path from v_5 to v_8 is $v_5v_3v_2v_6v_7v_8$. The two paths share one of the common sub-path $v_2v_6v_7v_8$ which is the shortest path from border v_2 to border v_8. This common path can be used to compute the shortest path from v_4 to v_8 and the shortest path from v_5 to v_8. This implies us to only store the shortest-path distances of pairs between borders (e.g., (v_2, v_8)) within one `G-tree` node.

Thus, we pre-compute and store the following pairs. (1) For a leaf node, we maintain the vertex-border pairs in the same leaf node, e.g., (v_4, v_2); (2) For a non-leaf node, we maintain the border-border pair, where the borders are from its children, e.g., (v_2, v_6) in node $\mathcal{G}_0$, (v_6, v_8) in node $\mathcal{G}_1$. For each node, we use a distance matrix to maintain the shortest-path distances between such pairs on the `G-tree` (Section 3.1). Based on the distance matrix, we can assemble these pairs and compute the shortest-path distance from a vertex to a node/vertex. For example, to compute the shortest path from v_4 to v_9, we can assemble (v_4, v_2), (v_2, v_6), (v_6, v_8), and (v_8, v_9) (as shown in Figure 4). Next we formally introduce our method based on three cases.

MinDist-Outside-Leaf: Consider two vertices u, v in two different leaf nodes. Let $\texttt{LCA}(u, v)$ denote the least common ancestor of nodes $\texttt{leaf}(u)$ and $\texttt{leaf}(v)$. Let $\texttt{LCA}(u, v), \mathcal{G}_1(u), \mathcal{G}_2(u), \cdots, \mathcal{G}_x(u) = \texttt{leaf}(u)$ denote the ancestors of $\texttt{leaf}(u)$ from $\texttt{LCA}(u, v)$ to $\texttt{leaf}(u)$ as illustrated in Figure 5. Let $\texttt{LCA}(u, v), \mathcal{G}_1(v), \mathcal{G}_2(v), \cdots, \mathcal{G}_y(v) = \texttt{leaf}(v)$ denote the ancestors of $\texttt{leaf}(v)$ from $\texttt{LCA}(u, v)$ to $\texttt{leaf}(v)$.

We consider two general cases. The first one is within one branch, i.e., $\mathcal{G}_x(u)$ to $\mathcal{G}_1(i)$ or $\mathcal{G}_y(v)$ to $\mathcal{G}_1(v)$. Given two adjacent levels of nodes, e.g., $\mathcal{G}_i(u)$ and $\mathcal{G}_{i-1}(u)$, $1 < i \leq x$, the shortest path from u to $\mathcal{G}_{i-1}(u)$ must contain a border in $\mathcal{G}_i(u)$ based on the closure property, thus

$$\texttt{SPDist}(u, \mathcal{G}_{i-1}(u)) = \min_{u_i \in \mathcal{B}(\mathcal{G}_i(u))} \big(\texttt{SPDist}(u, u_i) + \texttt{SPDist}(u_i, \mathcal{G}_{i-1}(u))\big). \tag{3}$$

The second case is between two branches, i.e., from $\mathcal{G}_1(u)$ to $\mathcal{G}_1(v)$. Similarly, we have

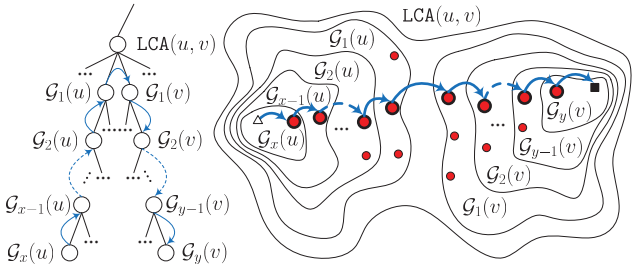


Figure 5: MinDist-Outside-Leaf.

$$\texttt{SPDist}(u, \mathcal{G}_1(v)) = \min_{u_1 \in \mathcal{B}(\mathcal{G}_1(u))} \big(\texttt{SPDist}(u, u_1) + \texttt{SPDist}(u_1, v_1)\big). \tag{4}$$

Equations 3 and 4 indicate that calculating $\texttt{SPDist}(u, u_{x-1})$ is only related to those of adjacent level, i.e. $\texttt{SPDist}(u, u_x)$. Thus, we can implement a *dynamic-programming algorithm* to efficiently calculate $\texttt{SPDist}(u, v)$. Consider the nodes $\mathcal{G}_x(u) = \texttt{leaf}(u), \mathcal{G}_{x-1}(u), \cdots, \mathcal{G}_1(u), \mathcal{G}_1(v), \mathcal{G}_2(v), \cdots, \mathcal{G}_y(v) = \texttt{leaf}(v)$. We first compute $\texttt{SPDist}(u, u_x)$ for $u_x \in \mathcal{G}_x(u)$. Then based on these results, we move forward to the next level to compute $\texttt{SPDist}(u, u_{x-1})$ for $u_{x-1} \in \mathcal{G}_{x-1}(u)$. Then we cross from $\mathcal{G}_1(u)$ to $\mathcal{G}_1(v)$, and move to the other branch. Iteratively, we can finally get $\texttt{SPDist}(u, v)$. Figure 5 shows the entire procedure.

LEMMA 2. *Consider a border b of node n. The border has the following properties. (1) For any child of n, e.g., c, if b is in node c, b must be a border of node c. (2) b must be a border of one of n's children.*

Note that, for any border-border pair in SPDist of Equations 3 and 4, e.g., $(u_2 \in \mathcal{G}_2(u), u_1 \in \mathcal{G}_1(u))$, they must appear in the distance matrix of a node (e.g., $\mathcal{G}_1(u)$, since u_1 must be a border of a child of $\mathcal{G}_1(u)$ based on Lemma 2 and u_2 is a border of $\mathcal{G}_1(u)$'s child $\mathcal{G}_2(u)$). Thus we can efficiently get the shortest-path distance of each pair from a distance matrix on the G-tree.

To summarize, given a query location v_q and a vertex v where $v_q \notin \texttt{leaf}(v)$, to compute $\texttt{SPDist}(v_q, v)$, we first compute their least common ancestor and the nodes on the paths from $\texttt{LCA}(v_q, v)$ to $\texttt{leaf}(v_q)$ and $\texttt{leaf}(v)$. Then we use the dynamic programming to compute $\texttt{SPDist}(v_q, v)$.

EXAMPLE 3. *Figure 6 and Figure 7(b) illustrate how to compute the shortest-path distance from v_4 to v_9. Initially we locate leaf nodes $\mathcal{G}_5$ (for v_4) and $\mathcal{G}_4$ (for v_9) by the hash table we mentioned at Section 4.1. Their LCA is $\mathcal{G}_0$. We use $\mathcal{G}_5, \mathcal{G}_2, \mathcal{G}_1, \mathcal{G}_4$ to compute the minimum distance. Each element in Figure 6 represents $\langle v_i, \texttt{SPDist}(v_q, v_i)\rangle$. By dynamic programming, we can finally get $\texttt{SPDist}(v_4, v_9) = 15$. The shortest-path contains vertices v_4, v_2, v_6, v_8 and v_9.*

MinDist-Outside-NonLeaf: Given a query location v_q, a node n, and $v_q \notin n$, we compute $\texttt{SPDist}(v_q, n)$ based on $\texttt{SPDist}(v_q, n) = \min_{w \in \mathcal{B}(n)} \texttt{SPDist}(v_q, w)$. Since each $\texttt{SPDist}(v_q, w \in \mathcal{B}(n))$ can be computed using the MINDIST-OUTSIDE-LEAF function, we can easily compute $\texttt{SPDist}(v_q, n)$.

MinDist-Inside-Leaf: Given a query location v_q, a vertex v, and $v_q \in \texttt{leaf}(v)$, consider a shortest path $\texttt{SP}(v_q, v)$ between v_q and v. There are two cases: (1) $\texttt{SP}(v_q, v)$ does not contain a vertex outside node $\texttt{leaf}(v_q)$. In this case, we use the Dijkstra algorithm to compute the shortest path in node $\texttt{leaf}(v_q)$. Let $\texttt{DijkDist}(v_q, v)$ denote the distance. Since the subgraph w.r.t. the leaf node is not large, the Dijkstra algorithm is efficient enough. (2) $\texttt{SP}(v_q, v)$ contains a vertex

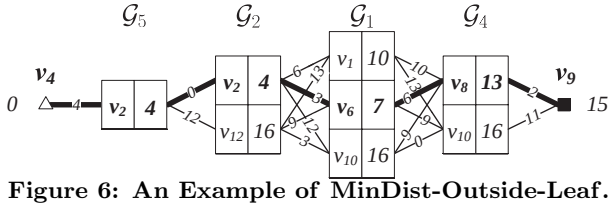


Figure 6: An Example of MinDist-Outside-Leaf.

outside node $\texttt{leaf}(v_q)$. In this case, $\texttt{SP}(v_q, v)$ must contain two borders b_1, b_2 in $\texttt{leaf}(v_q)$. Let $\texttt{BorderDist}(v_q, v)$ denote the shortest distance from v_q to v with outside vertices and,

$$\texttt{BorderDist}(v_q, v) = \min_{b_1, b_2 \in \mathcal{B}(\texttt{leaf}(v_q))} \big(\texttt{SPDist}(v_q, b_1) + \texttt{SPDist}(b_1, b_2) + \texttt{SPDist}(b_2, v)\big). \tag{5}$$

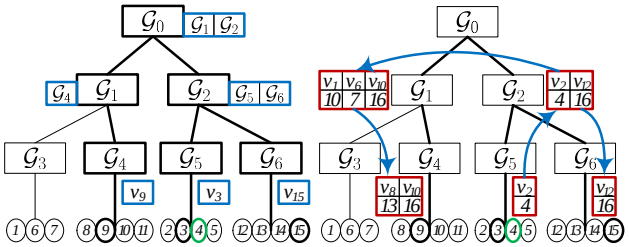
Based on the two cases, we have,

$$\texttt{SPDist}(v_q, v) = \min\big(\texttt{BorderDist}(v_q, v), \texttt{DijkDist}(v_q, v)\big). \tag{6}$$

Our method can correctly compute the shortest-path distance as formalized in Theorem 1.

THEOREM 1. *Given a query location v_q and a node/vertex e, the shortest-path distance between v_q and e computed by our algorithm is exactly $\texttt{SPDist}(v_q, e)$.*

Materialization-based Improvement: Although we can use MINDIST-OUTSIDE-LEAF and MINDIST-OUTSIDE-NONLEAF to calculate $\texttt{SPDist}(v_q, e)$, implementing them individually will result in many duplicated computations. For example, in Figure 7(b), if we calculate $\texttt{SPDist}(v_4, v_9)$ and $\texttt{SPDist}(v_4, v_{15})$ separately, we have to compute $\texttt{SPDist}(v_4, b_i \in \mathcal{G}_2)$ twice. Obviously, given a query location v_q, we only need to calculate $\texttt{SPDist}(v_q, b_i \in \mathcal{G}_i)$ once. Hence, we materialize $\langle b_i, \texttt{SPDist}(v_q, b_i)\rangle$ on the G-tree nodes which have been visited (see Figure 7(b)). Obviously this materialized method can avoid the duplicated computations, with space cost $\mathcal{O}(\frac{\log_2 f}{\sqrt{\tau}}|\mathcal{V}|)$ (see Section 4.3).



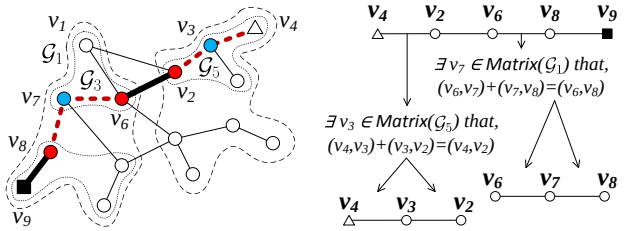
(a) Occurrence List. (b) Dynamic Programming.

Figure 7: Occurrence List and kNN Search.

Compared with the network-expansion search approach, our assembly-based method has two superior advantages. First, our method significantly reduces the overhead for calculating $\texttt{SPDist}(v_q, e)$. It is easy to see that we only traverse subtrees of G-tree which contain promising objects, and each tree node is only accessed once. Second, since $\texttt{SPDist}(v_q, v)$ is computed by means of step-by-step dynamic programming algorithms, and those materialized intermediate results $\texttt{SPDist}(v_q, b_i)$ on G-tree node are indispensable for effective pruning in best-first search (i.e., $\texttt{SPDist}(v_q, n)$), therefore, there are no redundant computations for kNN search, which makes our method very efficient for large road networks.

4.3 Time and Space Complexity of kNN Search

Time Complexity: The kNN search consists of two parts. The first one is the local Dijkstra search within MINDIST-INSIDE-LEAF. The time complexity is $\mathcal{O}(\tau \log \tau)$. The second one is MINDIST-OUTSIDE-LEAF. Since the dynamic-programming algorithm will only access each node of G-tree



(a) Paths on Graph. (b) Path Recovery.

Figure 8: An Example of Shortest-path Recovery.

once and each time it only scans the distance matrix of the tree node, the worst-case time complexity of MinDist-Outside-Leaf is the total size of the distance matrices of `G-tree`, i.e., $\mathcal{O}(\log_2^2 f \cdot \log_f \frac{|\mathcal{V}|}{\tau} \cdot |\mathcal{V}|)$. To sum up, the worst-case time complexity of the assembly-based method is $\mathcal{O}(\tau \log \tau + \log_2^2 f \cdot \log_f \frac{|\mathcal{V}|}{\tau} \cdot |\mathcal{V}|)$. In practice, the complexity is much smaller than the worst-case complexity.

Space Complexity: For each node of `G-tree`, the dynamic-programming algorithm maintains $\texttt{SPDist}(v_q, b_i)$, where b_i is a border. Thus, the worst-case space complexity is the total number of borders, i.e. $\mathcal{O}(\frac{\log_2 f}{\sqrt{\tau}}|\mathcal{V}|)$.

5. DISCUSSIONS

5.1 Path Recovery

It is worth noting that Algorithm 1 only returns distance rather than vertex-by-vertex path. However, the latter is sometimes very useful (e.g., in navigation system). In this section, we briefly discuss how to recover the path from the query location v_q to an answer $v_a \in \mathcal{R}$ selected by a user.

Since we use assembly-based method for the kNN finding, we can only get a list of selected borders from v_q to v_a, i.e. the imperfect shortest path $\texttt{SP}'(v_q, v_a) = v_q b_1 b_2 \cdots b_m v_a$. As there may be no direct edges between two adjacent vertices, e.g., $\langle b_i, b_{i+1} \rangle$, we need to add some other vertices between them to generate the real shortest path $\texttt{SP}(v_q, v_a)$.

The main idea is to apply divide-and-conquer to iteratively add new vertices into the $\texttt{SP}'(v_q, v_a)$. For example, in Figure 8, to compute the shortest-path distance from v_4 to v_9, we get $\texttt{SP}'(v_4, v_9) = v_4 v_2 v_6 v_8 v_9$. As there is no edge between v_4 and v_2, we need to find a vertex, i.e. v_3, to add between them(since $\texttt{SPDist}(v_4, v_2) = \texttt{SPDist}(v_4, v_3) + \texttt{SPDist}(v_3, v_2)$). Similarly we add vertex v_7 between v_6 and v_8. Thus the shortest path is $\texttt{SP}(v_4, v_9) = v_4 v_3 v_2 v_6 v_7 v_8 v_9$.

Luckily, we can always find a border b_w to split $\langle b_i, b_{i+1} \rangle$ into $\langle b_i, b_w \rangle$ and $\langle b_w, b_{i+1} \rangle$, where b_i, b_w, b_{i+1} must all appear in the same distance matrix from $\texttt{LCA}(\mathcal{G}(b_i), \mathcal{G}(b_{i+1}))$ to the root node of `G-tree`. Thus, each vertex finding only costs $\mathcal{O}(\mathcal{HB}_{max})$. However, due to lack of space, we have to omit the details and proofs here. Any interested readers may contact the authors for further information.

5.2 Computing Distance Matrix Efficiently

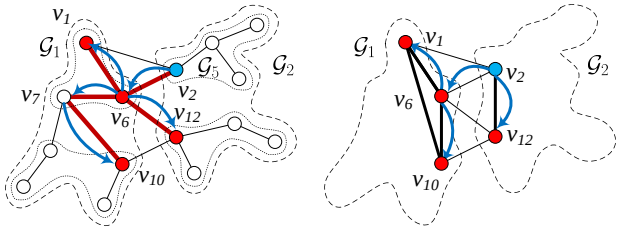
We propose a bottom-up method to efficiently compute the distance matrix. The basic idea is that we first compute the shortest-path distance of borders of nodes in the lower level and then use these distances to compute the shortest-path distance of borders of nodes in the upper level. For example, in Figure 9, to compute the shortest-path distance between borders v_2 and v_{10}. A naive method needs to access v_6 and v_7. As we have calculated the shortest-path distance between v_6 and v_{10} in $\mathcal{G}_1$, we skip vertex v_7 and directly use $\langle v_6, v_{10} \rangle$ in $\mathcal{G}_0$. We use this property to compute the distance matrix. Our algorithm works as follows.

(1) Initially, for each leaf node, we use the Dijkstra algorithm to compute the shortest-path distance between any two borders in the leaf node.

(2) We remove all non-border vertices in the leaf node and add shortcuts between any two borders of the leaf node.

(3) We move to the parent of leaf nodes and use the Dijkstra algorithm to compute the shortest-path distance between any two borders in the parent based on the updated graph.

(4) We repeat steps 2 and 3 and terminate the algorithm if we have processed the root node.



(a) Naive Method. (b) A Bottom-up Method.

Figure 9: Distance Matrix Computation (Distances of Borders to v_2).

5.3 `G-tree` Maintenance for Network Updates

We discuss how to maintain the `G-tree` for network updates. Although it is a very hard problem to support updates for shortest-path queries in graphs [22], we propose a feasible method to adjust the `G-tree` with a low overhead for network updates. We consider four basic operations and other operations can be split into these basic operations.

Insert a new vertex u with an edge to an existing vertex v: We first locate the leaf node $\texttt{leaf}(v)$ and insert u into $\texttt{leaf}(v)$. If $\texttt{leaf}(v)$ has more than τ vertices, we partition the $\texttt{leaf}(v)$ into f nodes. Then, we recursively repartition the ancestor of $\texttt{leaf}(v)$ until all nodes have no more than τ children. Distance matrices are also updated.

Remove a vertex u with only one edge to another vertex v: We first locate node $\texttt{leaf}(u)$ and remove u from $\texttt{leaf}(u)$ and the corresponding distance matrix. If $\texttt{leaf}(u)$ is empty, we recursively repartition the ancestor of $\texttt{leaf}(u)$.

Add an edge (u, v): If u (or v) becomes a border from a non-border vertex, we add it into the corresponding distance matrix. If $\texttt{weight}(u, v) \geq \texttt{SPDist}(u, v)$, we do not update the `G-tree`; otherwise we update the distance matrix as follows. Consider a pair $\langle b_i, b_j \rangle$ in a distance matrix. We check whether $\texttt{SPDist}(b_i, u) + \texttt{weight}(u, v) + \texttt{SPDist}(v, b_j) \leq \texttt{SPDist}(b_i, b_j)$. If so, we directly update $\texttt{SPDist}(b_i, b_j) = \texttt{SPDist}(b_i, u) + \texttt{weight}(u, v) + \texttt{SPDist}(v, b_j)$.

Remove an edge (u, v): If u (or v) becomes a non-border vertex from a border, we remove it from the corresponding distance matrix. If $\texttt{weight}(u, v) \geq \texttt{SPDist}(u, v)$, we do not update the `G-tree`; otherwise we update the distance matrix as follows. Consider a pair $\langle b_i, b_j \rangle$ in a distance matrix. If $\texttt{SPDist}(b_i, u) + \texttt{weight}(u, v) + \texttt{SPDist}(v, b_j) > \texttt{SPDist}(b_i, b_j)$, we do not update $\texttt{SPDist}(b_i, b_j)$ as we will not use (u, v) to compute $\texttt{SPDist}(b_i, b_j)$; otherwise we recompute $\texttt{SPDist}(b_i, b_j)$.

5.4 Extension to Directed Graphs

In this section we discuss how to use the `G-tree` to support directed graphs with a minor change. First, in the distance matrix, we keep the shortest distances of directed paths from a vertex to a border/vertex. We only need to slightly modify the pre-computation method in Section 5.2 to compute the

Table 1: Datasets.

Data	Description	# Vertices	# Edges
CAL	California(Undirected)	21,048	21,693
SF	San Francisco(Undirected)	174,956	223,001
COL	Colorado(Undirected)	435,666	528,533
FLA	Florida(Undirected)	1,070,376	1,356,399
E-USA	East USA(Undirected)	3,598,623	4,389,057
C-USA	Center USA(Undirected)	14,081,816	17,146,248
USA	USA(Undirected)	23,947,347	29,166,672
WA	Washington(Directed)	514,654	1,246,353

directed distances. Second, our method relies on using the assembly based algorithm to implement the `SPDist` function. Nevertheless, the assembly based method still works for directed graphs based on the following reasons. (1) The closure property (Lemma 1) still holds for directed graphs, i.e., given a subgraph $\mathcal{G}' = \langle \mathcal{V}', \mathcal{E}' \rangle$, any directed path from $u \in \mathcal{V}'$ to $v \notin \mathcal{V}'$ must contain at least one border in $\mathcal{B}(\mathcal{G}')$. (2) We can still use the dynamic-programming algorithm in Section 4.2 to compute the shortest distance of a directed path. Third, we slightly modify the Dijkstra algorithm to support the directed graphs.

6. EXPERIMENTS

Datasets: We used eight real-world datasets with various sizes from 20,000 vertices to 24 million vertices. *CAL* consists of highways and main roads in California and *SF* contains detailed street networks in San Francisco [1], which are widely used in previous studies [15]. *COL*, *FLA*, *E-USA*, *C-USA* and *USA* are road networks of USA[2], which are composed of detailed streets, roads and highways. *WA* is the road network of a directed graph of Washington State[3]. The statistics of these datasets is illustrated in Table 1.

Query Sets: To evaluate the kNN search performance, we randomly chose 100 vertices as the query location, and for each query location we generated 100 groups of objects, thus we had 10,000 queries for each query set. For objects we uniformly selected 0.0001, 0.001, **0.01**, 0.05, 0.1 of vertices from the dataset as objects (the default value is 0.01). For k, we used 1, 5, **10**, 20, 50 (the default value is 10).

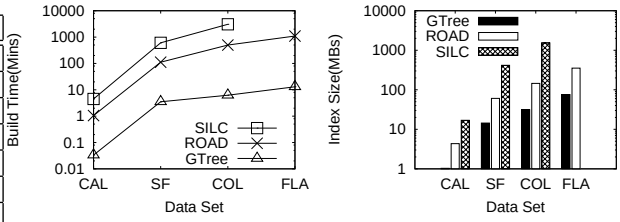
We compared with state-of-the-art methods *SILC* [23] and *ROAD* [15, 16]. *SILC* was implemented by ourselves and *ROAD* was provided by the authors. All the algorithms were implemented in C++. In `G-tree`, the default fanout is $f = 4$ and τ is set to 64, 128, 128, 256, 256, 512 and 512 respectively for the first seven datasets. For implementing the `SPDist` function, as two naive methods were not scalable in space($\mathcal{O}(c|\mathcal{V}|^2)$), we only used the assembly-based method. For *SILC* and *ROAD*, we used default settings as stated in the original papers and both were conducted under memory-based setting. All experiments were conducted on a Linux computer with Intel 2.50GHz CPU and 16GB memory.

6.1 Evaluation on Parameters: Fanout and τ

`G-tree` has two parameters - fanout f and the number of vertices in a leaf node τ. We tested the effect on the two parameters. We varied τ in {32, 64, 128, 256, 512} and f in {2, 4, 8, 16}. We evaluated the number of borders, the index size, index build time, and the average k nearest neighbors search performance of 10,000 queries. We used the *COL* dataset. Figure 10 shows the results.

We made two observations. First, with the increase of f, the number of borders, the index size, the index build time and the query time first decreased and then increased. Our method achieved the best results when $f = 4$. The main reasons are as follows. On the one hand larger fanouts will generate larger numbers of borders to partition a subgraph. On the other hand, larger fanouts will reduce the height of `G-tree` and the number of nodes that need to be partitioned.

Second, with the increase of τ, the number of borders decreased. This is because bigger τ results in smaller tree size. Besides, the index size and construction time also decreased, as the index size depends on the number of borders. With the increase of τ, the search time first decreased and then increased. Because if τ is larger, it takes more time on the Dijkstra search in large leaf nodes; if τ is smaller, it involves large numbers of borders. We selected $\tau = 128$ as a trade off between query efficiency and indexing size.



(a) Build Time. (b) Index Size.

Figure 11: Index Comparison.

6.2 Comparison with State-of-the-art Schemes

We compared our proposed method against state-of-the-art methods *SILC* [23] and *ROAD* [15, 16], in terms of index overhead and kNN search time. Our method can process all datasets. Since *ROAD* and *SILC* took a mass of pre-processing time and consumed large amount of memory, both schemes failed on *E-USA*, *C-USA* and *USA*. In addition, *SILC* also failed to run on *FLA*. For example, on *E-USA*, we estimated 4.8 days to be required for *ROAD*, and 36.5GB memory cost for *SILC*.

Evaluation on Index Construction: We first evaluated the time and space overhead of indexing. Figure 11 illustrates the index sizes and index build time.

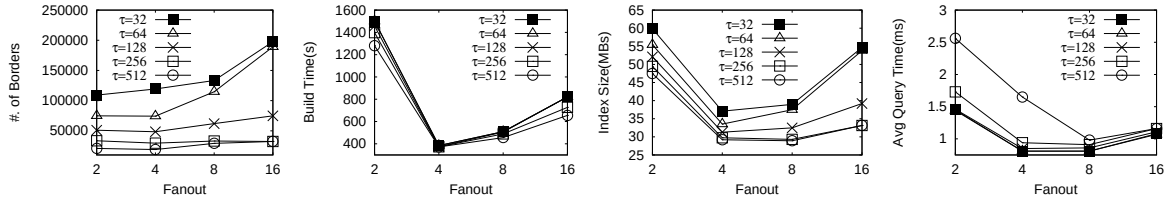
We can see that `G-tree` outperformed *ROAD* and *SILC* in index build time and sizes. On *COL*, the index build time of `G-tree` was better than *ROAD* by an order of magnitude and nearly three orders of magnitude better than *SILC*. For index sizes, on *COL*, `G-tree` consumed 45.5 MB, *ROAD* took up 145 MB and *SILC* required 1535 MB. This is because the space overhead of *SILC* is $\mathcal{O}(|\mathcal{V}|^{1.5})$ and it is rather expensive to compute all-pair shortest paths. *ROAD* involved larger numbers of borders than our methods and needed to store shortest-path distances of all border pairs. Thus *ROAD* took more space and time than ours. Furthermore, `G-tree` only took 16.8 hours to build index on *USA*.

Evaluation on kNN Search: We evaluated the kNN search efficiency of `G-tree`, *ROAD* and *SILC* by varying the number of answers k, the number of objects $|\mathcal{C}|$, datasets, and distances from query location to top-k answers.

kNN Search by Varying k: We used the *COL* dataset and evaluated the average search time of 10,000 queries. We set $|\mathcal{C}| = 0.01|\mathcal{V}|$. Figure 12(a) shows the results.

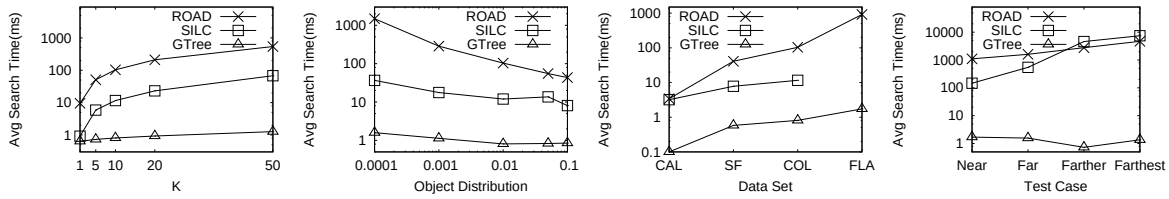
We can see that `G-tree` outperformed the two state-of-the-art methods for different k, and even by 2-3 orders of magnitude for $k \geq 10$. The main reason is as follows. Since *SILC* had to search multiple quadtrees to find distances between query location and objects, this operation was very costly and inefficient for larger k. As *ROAD* employs an expansion-based method, it only pruned the nodes which have no objects and cannot use distance-based pruning. On

[1] http://www.cs.fsu.edu/~lifeifei/SpatialDataset.htm
[2] http://www.dis.uniroma1.it/challenge9/index.shtml
[3] http://depts.washington.edu/giscup/roadnetwork



(a) Number of Borders. (b) Index Build Time(s). (c) Index Size(MB). (d) Query Time(ms).

Figure 10: Evaluation on Parameters: f and τ (*COL* dataset, $k = 10$, 1% uniform vertices as objects).



(a) Varying k. (b) Varying Object Sizes. (c) Varying Datasets. (d) Varying Object Distance.

Figure 12: Performance Comparison on kNN Search (*COL* dataset, $k = 10$, 1% uniform vertices as objects).

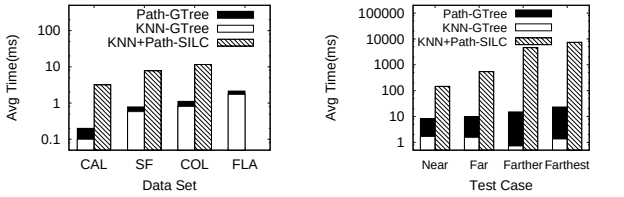
the contrary, `G-tree` used the `SPDist` function to prune unpromising nodes. Thus if k is larger, the improvement of `G-tree` over *SILC* and *ROAD* becomes larger too.

kNN Search by Varying Object Sizes: We evaluated the kNN search performance by varying object sizes. We used the *COL* dataset. We generated five sets of objects with different sizes, where the sizes are respectively 0.0001, 0.001, 0.01, 0.05 and 0.1 of the number of the vertices in the dataset. Note that, we stopped at 0.1 as candidate objects are usually in small quantity compared with vertex size. We randomly generated 10,000 queries, set $k = 10$, and evaluated the average time. Figure 12(b) shows the results.

We made two observations. First, with the increase of the number of objects, the elapsed time of the three algorithms decreased. This is because for smaller number of objects, the objects are sparse and uniformly distributed in the dataset, and the average distance from the query location to the nearest neighbor tends to be larger. Thus the algorithms need to visit more vertices. Second, `G-tree` outperformed *SILC* and *ROAD* a lot. With `G-tree`, we can directly locate the promising tree nodes and prune unpromising ones by best-first search, while this is what *ROAD* and *SILC* fail to do. Therefore, changing the sizes of objects has no significant effect on our `G-tree`.

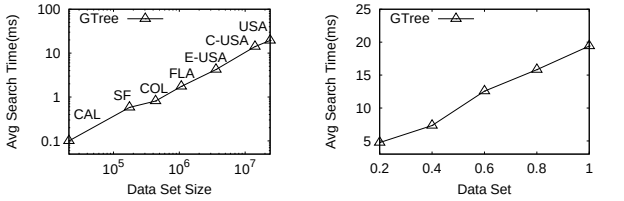
kNN Search by Varying Datasets: We tested the performance of three algorithms on four datasets *CAL*, *SF*, *COL* and *FLA*. We set k=10 and $|\mathcal{C}|$=0.01$|\mathcal{V}|$. Figure 12(c) shows the results. We can see that `G-tree` outperformed *ROAD* and *SILC* on every dataset. For example, on *FLA*, *ROAD* took about 1000 milliseconds, *SILC* cannot support this large road network as it consumed too much memory. `G-tree` only took 2 milliseconds. Notice that with the increase of the dataset size, the improvement of `G-tree` over *SILC* and *ROAD* becomes large, because `G-tree` can efficiently prune unnecessary subgraphs based on the `SPDist` function.

kNN Search by Varying Object Distances: We generated four query sets based on the distances from the query location to the objects on *COL* dataset. We first computed the minimum bounding box of the geometric coordinates for all the vertices on the dataset and then calculated the length of the diagonal line denoted by l_d. Next, we generated four query sets where the distances of objects to the query location are respectively larger than $\frac{l_d}{8}$, $\frac{l_d}{4}$, $\frac{l_d}{2}$, $\frac{3l_d}{4}$. Each query set contained 10,000 queries. The four query sets are respectively



(a) Varying Networks. (b) Varying Object Distances.

Figure 13: Evaluation on Path Recovery.



(a) All Datasets. (b) Subsets of *USA*.

Figure 14: Scalability on Performance of `G-tree`.

called "`near`", "`far`", "`farther`", "`farthest`". Figure 12(d) shows the results. We can see that `G-tree` significantly outperformed *ROAD* and *SILC*, even in 2-3 orders of magnitude. *SILC* and *ROAD* achieved very poor performance since they had to traverse long distance paths before they accessed all top-k answers.

6.3 Evaluation on Path Recovery

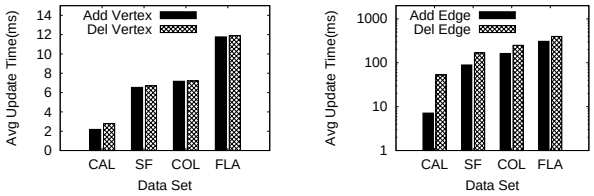
We evaluated the efficiency of our path-recovery algorithm. As *ROAD* cannot support path recovery[4], we only compared with *SILC*. We used the same setting as the kNN search. The result is shown in Figure 13.

In the figure we show both the search time and the path recovery time. We can see that our algorithm can efficiently find the path. For example, on the *FLA* dataset, it only took 0.1 milliseconds. With the increase of the distance from the objects to the query location, the path recovery took more time as our method will access more vertices. However, Our method was still much better than *SILC*.

6.4 Scalability

We evaluated the time and space scalability of the `G-tree`. As the first seven datasets had various sizes, we evaluated the scalability using the first seven datasets. We set $k = 10$ and $|\mathcal{C}| = 0.01|\mathcal{V}|$. Figure 14(a) shows the efficiency results.

[4]In the *ROAD* paper, the authors did not discuss the path recovery issue.



(a) On Add/Delete Vertices. (b) On Add/Delete Edges.

Figure 15: Evaluation on G-tree Maintenance Cost.

We can see that G-tree scaled well as the data size increased from 0.01 million to 24 million. The average search time on the USA dataset 24 million vertices was only 20 milliseconds. Table 2 shows the space scalability of the G-tree. We can see that the index size of G-tree increased linearly with the increases of the data size.

Table 2: Scalability on Index Sizes(MB) of G-tree.

Size	*CAL*	*SF*	*COL*	*FLA*	*E-USA*	*C-USA*	*USA*
Data	1.13	11.7	28.0	70.8	244.3	990	1725
Index	1.34	22.4	45.5	109	425.5	1943	3184

We evaluated the scalability on the *USA* dataset by partitioning the dataset into five equal-sized subgraphs. Then we merged 1, 2, 3, 4, 5 subgraphs to test the scalability. Figure 14(b) shows the results and G-tree scaled well.

6.5 Evaluation on G-tree for Updates

We evaluated the G-tree maintenance cost for network updates by inserting/deleting vertices and edges. Figure 15 shows the results. Our method can efficiently support updates. For vertex updates, the average time was 10 milliseconds on the *FLA* dataset. For edge updates, the average time was 200 milliseconds. Notice that it is a very hard problem to support edge updates [22]. As road networks are not updated frequently, the update time is acceptable.

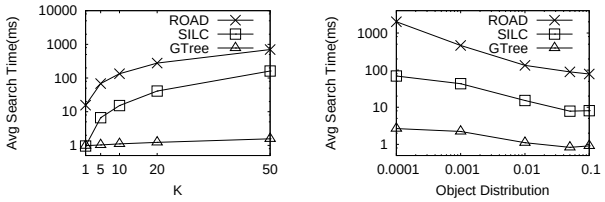
6.6 Evaluation on Directed Graph

We evaluated the search efficiency of G-tree on a directed graph *WA*. We set $f = 4$ and $\tau = 128$. Table 3 shows the overview of the G-tree on dataset *WA*.

Table 3: G-tree Overview on Dataset *WA*.

Data Size	Index Size	Build Time	# Borders
47.55(MB)	52.13(MB)	688(s)	55875

We compared the kNN performance with *SILC* and *ROAD* by varying k and object size $|\mathcal{C}|$, as shown in Figure 16. G-tree still significantly outperformed existing methods. The results are consistent with those on undirected graphs.



(a) Varying k. (b) Varying Objects Sizes.

Figure 16: Evaluation on Directed Graphs ($k = 10$, 1% uniform vertices as objects).

7. CONCLUSION

In this paper we have studied the problem of kNN search on road networks. We proposed a balanced search tree structure G-tree and devised an efficient best-first search algorithm on the basis of the assembly-based method. Experimental results show that G-tree significantly outperforms state-of-the-arts in terms of both efficiency and index sizes.

8. ACKNOWLEDGEMENT

This work was partly supported by the National Natural Science Foundation of China under Grant No. 60833003, 61003004 and 61272090, National Grand Fundamental Research 973 Program of China under Grant No. 2011CB302206, a project of Tsinghua University under Grant No. 20111081073, Tsinghua-Tencent Joint Laboratory for Internet Innovation Technology, and the NExT Research Center which is supported by the Singapore National Research Foundation & Interactive Digital Media R&D Program Office, MDA under research grant (WBS:R-252-300-001-490).

9. REFERENCES

[1] I. Abraham, D. Delling, A. V. Goldberg, and R. F. F. Werneck. Hierarchical hub labelings for shortest paths. In *ESA*, pages 24–35, 2012.

[2] H. Bast, S. Funke, and D. Matijevic. Transit - ultrafast shortest-path queries with linear-time preprocessing. In *In 9th DIMACS Implementation Challenge*, pages 175–192, 2006.

[3] H.-J. Cho and C.-W. Chung. An efficient and scalable approach to cnn queries in a road network. In *VLDB*, pages 865–876, 2005.

[4] E. W. Dijkstra. A note on two problems in connexion with graphs. *NUMERISCHE MATHEMATIK*, 1(1):269–271, 1959.

[5] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[6] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *WEA*, pages 319–333, 2008.

[7] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM Trans. Database Syst.*, 24(2):265–318, 1999.

[8] H. Hu, D. L. Lee, and V. C. S. Lee. Distance indexing on road networks. In *VLDB*, pages 894–905, 2006.

[9] H. Hu, D. L. Lee, and J. Xu. Fast nearest neighbor search on road networks. In *EDBT*, pages 186–203, 2006.

[10] C. S. Jensen, J. Kolárvr, T. B. Pedersen, and I. Timko. Nearest neighbor queries in road networks. In *GIS*, pages 1–8, 2003.

[11] N. Jing, Y.-W. Huang, and E. A. Rundensteiner. Hierarchical encoded path views for path query processing: An optimal model and its performance evaluation. *IEEE Trans. Knowl. Data Eng.*, 10(3):409–432, 1998.

[12] S. Jung and S. Pramanik. An efficient path computation model for hierarchically structured topographical road maps. *IEEE Transactions on Knowledge and Data Engineering*, 14(5):1029–1046, 2002.

[13] G. Karypis and V. Kumar. Analysis of multilevel graph partitioning. In *SC*, 1995.

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

[15] K. C. K. Lee, W.-C. Lee, and B. Zheng. Fast object search on road networks. In *EDBT*, pages 1018–1029, 2009.

[16] K. C. K. Lee, W.-C. Lee, B. Zheng, and Y. Tian. Road: A new spatial object search framework for road networks. *IEEE Trans. Knowl. Data Eng.*, 24(3):547–560, 2012.

[17] G. Li, J. Feng, and J. Xu. Desks: Direction-aware spatial keyword search. In *ICDE*, pages 474–485, 2012.

[18] R. J. Lipton and R. E. Tarjan. A Separator Theorem for Planar Graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, 1979.

[19] K. Mouratidis, M. L. Yiu, D. Papadias, and N. Mamoulis. Continuous nearest neighbor monitoring in road networks. In *VLDB*, pages 43–54, 2006.

[20] D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao. Query processing in spatial network databases. In *VLDB*, pages 802–813, 2003.

[21] J. B. Rocha-Junior and K. Nørvåg. Top-k spatial keyword queries on road networks. In *EDBT*, pages 168–179, 2012.

[22] L. Roditty and U. Zwick. On dynamic shortest paths problems. *Algorithmica*, 61(2):389–401, 2011.

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

[24] J. Sankaranarayanan, H. Alborzi, and H. Samet. Efficient query processing on spatial networks. In *GIS*, pages 200–209, 2005.

[25] J. Sankaranarayanan, H. Samet, and H. Alborzi. Path oracles for spatial networks. *PVLDB*, 2(1):1210–1221, 2009.

[26] H. Wang and R. Zimmermann. Snapshot location-based query processing on moving objects in road networks. In *GIS*, page 50, 2008.

[27] H. Wang and R. Zimmermann. Processing of continuous location-based range queries on moving objects in road networks. *IEEE Trans. Knowl. Data Eng.*, 23(7):1065–1078, 2011.

[28] R. Zhong, J. Fan, G. Li, K.-L. Tan, and L. Zhou. Location-aware instant search. In *CIKM*, pages 385–394, 2012.