This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2015.2399306, IEEE Transactions on Knowledge and Data Engineering

G-Tree: An Efficient and Scalable Index for Spatial Search on Road Networks

Ruicheng Zhong, Guoliang Li, Kian-Lee Tan, Lizhu Zhou, and Zhiguo Gong

Abstract—In the recent decades, we have witnessed the rapidly growing popularity of location-based systems. Three types of location-based queries on road networks, single-pair shortest path query, k nearest neighbor (kNN) query, and keyword-based kNN query, are widely used in location-based systems. Inspired by `R-tree`, we propose a height-balanced and scalable index, namely `G-tree`, to efficiently support these queries. The space complexity of `G-tree` is $\mathcal{O}(|\mathcal{V}| \log |\mathcal{V}|)$ where $|\mathcal{V}|$ is the number of vertices in the road network. Unlike previous works that support these queries separately, `G-tree` supports all these queries within one framework. The basis for this framework is an assembly-based method to calculate the shortest-path distances between two vertices. Based on the assembly-based method, efficient search algorithms to answer kNN queries and keyword-based kNN queries are developed. Experiment results show `G-tree`'s theoretical and practical superiority over existing methods.

Index Terms—Single-Pair Shortest Path, KNN Search, Keyword Search, Road Network, Index, Spatial Databases

1 Introduction

Nowadays, mobile devices have become more and more popular in our daily life. As existing mobile devices are usually equipped with global positioning system (GPS) chips, users can easily get their locations. To provide users with location-based services (LBS), many LBS systems (e.g., Foursquare and Google Maps) have been deployed and widely accepted by mobile users. In our daily life, there are many LBS applications on road networks. For example, in navigation systems, a user may want to find the shortest path between two vertices on a road network. In tourist guide applications, a tourist may look for k nearest "`seafood restaurants`" while walking in a city. There are three widely-used types of queries on road networks, single-pair shortest path query, k nearest neighbor(kNN) query, and keyword-based kNN (k^2N^2) query.

There are many existing works to support these three types of queries. [1], [5], [9], [10], [20], [21] studied the problem of shortest path queries between two vertices, [2], [6], [7], [12], [13], [16], [19] addressed the kNN search problem, and [14], [17], [18] worked on the problem of keyword-based kNN query on road networks. Though they achieve good results for individual type of queries, there are still limitations to be addressed for them to be of practical use. First, some are not efficient enough, especially on kNN query (e.g., [13], [16], [7], [12]). Second, some fail to scale up to very large datasets (e.g., [13], [19], [6]). Third, they cannot be adaptive to different types of queries. For example, if a LBS system needs to support shortest-path query, kNN query and k^2N^2 query at the same time, it has to exploit three different methods or indices, and the overhead is very costly. To this end, we aim to design an efficient and scalable index on road networks, which can not only handle very large datasets effectively, but also support various spatial queries on road networks.

However, to our best knowledge, there is no such index on road networks. Inspired by `R-tree` on the metric space, we devise a similar tree-based index on road networks, namely `G-tree`. We recursively partition the road network into sub-networks and build a tree structure on top of sub-networks where each `G-tree` node corresponds to a sub-network. The space complexity of our `G-tree` structure is only $\mathcal{O}(|\mathcal{V}| \log |\mathcal{V}|)$, which is much better than the state-of-the-art method [19] with $\mathcal{O}(|\mathcal{V}|^{1.5})$ complexity, where $|\mathcal{V}|$ is the number of vertices in the road network.

Using the `G-tree` index, we devise effective algorithms to support three types of spatial queries on road networks, single-pair shortest path query (SPSP), k nearest neighbor query (kNN) and keyword-based kNN query (k^2N^2), which are fundamental and popular for a variety of applications. For SPSP, we devise an assembly-based algorithm to compute the shortest-path distance between two vertices with $\mathcal{O}(|\mathcal{V}|)$ time complexity. For kNN, we develop an efficient top-k search scheme and devise effective pruning techniques based on the minimum boundary between the query location and a `G-tree` node. We also combine the spatial and textual information together to handle k^2N^2 query judiciously. 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 low space overhead but superior performance and good scalability. The space complexity of our `G-tree` structure is only $\mathcal{O}(|\mathcal{V}| \log |\mathcal{V}|)$.
- We propose efficient search algorithms for three types

- *Ruicheng Zhong, Guoliang Li and Lizhu Zhou are with the Department of Computer Science and Technology, Tsinghua National Laboratory for Information Science and Technology (TNList), Tsinghua University, Beijing 100084, China. E-mail: zrc1101001@gmail.com; {liguoliang,dcszlz}@tsinghua.edu.cn.*
- *Kian-Lee Tan is with the Department of Computer Science, National University of Singapore, Singapore. E-mail: tankl@comp.nus.edu.sg.*
- *Zhiguo Gong is with the Department of Computer Science, Univeristy of Macau, Macau, China. E-mail: fstzgg@umac.mo.*

1041-4347 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2015.2399306, IEEE Transactions on Knowledge and Data Engineering

of queries: SPSP, kNN, and k^2N^2. We devise an assembly-based algorithm to compute shortest-path distance for SPSP queries. Based on the algorithm and the G-tree, we devise a best-first search algorithm to support kNN queries on road networks. We also extend the kNN algorithm to support k^2N^2 queries.

- Our method has theoretical and practical superiority over existing methods. Experimental results on real-world datasets show that G-tree significantly outperforms state-of-the-art methods and scales very well.

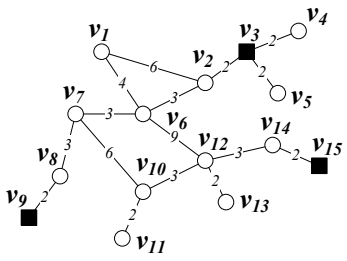
The structure of this paper is organized as follows. We define three types of queries and review related works in Section 2. We present our G-tree index and search algorithms in Sections 3 and 4 respectively. We discuss the path recovery issue in Section 5. Miscellaneous issues are discussed in Section 6. Experiments are reported in Section 7 and we conclude in Section 8.

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 $\mathtt{w}(u, v)$, e.g., road 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. Among all paths between vertexes u and v, the one with the minimum distance is called the shortest path. Let $\mathtt{SP}(u, v)$ denote a shortest path between u and v, and $\mathtt{SPDist}(u, v)$ denote the shortest-path distance between u and v. We will discuss how to extend our method to support directed graphs in Section 6.2. Each vertex v contains a list of keywords, denoted by $v.\mathcal{W}$, which is the textual description of the vertex.

For example, Figure 1(a) shows a road network. The weight of edge (v_2, v_6) is 3. $\mathtt{SP}(v_4, v_9)$=$v_4v_3v_2v_6v_7v_8v_9$ is a shortest path between v_4 and v_9 and $\mathtt{SPDist}(v_4, v_9)$=15. Figure 1(b) displays the textual description of each vertex on the road network in Figure 1(a).



(a) A road network.

v_1	taxi stop
v_2	ϕ
v_3	city square
v_4	Greek restaurant
v_5	junior high school
v_6	ϕ
v_7	primary school
v_8	ϕ
v_9	national park
v_{10}	grand hotel
v_{11}	ϕ
v_{12}	bus stop
v_{13}	Chinese restaurant
v_{14}	ϕ
v_{15}	park

(b) Keyword information.

Fig. 1. An Example of the Road Network.

Query Model. We consider three types of queries: single-pair shortest path query (SPSP), k nearest neighbors query (kNN) and keyword-based kNN query (k^2N^2).

(1) **Single-Pair Shortest Path Query**: Given a graph $\mathcal{G}$ and a query $q = \langle u, v \rangle$, SPSP returns $\mathtt{SPDist}(u, v)$. For example, consider a query $q = \langle v_4, v_9 \rangle$ in Figure 1(a). we can figure out that $\mathtt{SP}(v_4, v_9) = v_4v_3v_2v_6v_7v_8v_9$, thus $\mathtt{SPDist}(v_4, v_9) = 15$.

(2) k **Nearest Neighbor Query**: Given a graph $\mathcal{G}$ and a query $q = \langle v_q, \mathcal{C}, k \rangle$, where v_q is the query vertex, $\mathcal{C}$ is a set of object vertices, and k is an integer, the answer $\mathcal{R}$ 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) $\forall v \in \mathcal{R}, u \in \mathcal{C} - \mathcal{R}$, $\mathtt{SPDist}(v_q, v) \leq \mathtt{SPDist}(v_q, u)$.

For simplicity, in the paper we assume that both the query location and the objects are at vertices. If the query location is on an edge, we use the two end vertices of the edge to do kNN search and merge the answer sets of the two vertices to generate the final results. If the query location is outside the graph, we first find its nearest edge (by Euclidean distance) and then use the nearest location on the edge to find the final results. Similar variations can be applied to the object.

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

(3) **Keyword-based kNN Query**: Given a graph $\mathcal{G}$ and a query $q = \langle v_q, \mathcal{C}, \mathcal{W}, k \rangle$ and a ranking function $\mathcal{F}$, where $\mathcal{W}$ is a set of keywords, the answer $\mathcal{R}$ is a set of top-k objects in $\mathcal{C}$ based on the ranking function $\mathcal{F}$ 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) $\forall v \in \mathcal{R}, u \in \mathcal{C} - \mathcal{R}$, $\mathcal{F}(v_q, v) \geq \mathcal{F}(v_q, u)$.

A general ranking function $\mathcal{F}$ considers not only the spatial proximity but also textual relevancy between the queries and objects, which is defined as below.

$$\mathcal{F}(q, v) = \alpha(1 - \frac{\mathtt{SPDist}(q, v)}{M_S}) + (1 - \alpha)\frac{\mathtt{TXDist}(q, v)}{M_T} \quad (1)$$

where M_S is the maximum spatial distance, M_T is the maximum textual distance, and $\mathtt{TXDist}(q, v)$ can be any textual relevancy and for simplicity, in the paper we take the following tf*idf based textual relevance as an example,

$$\mathtt{TXDist}(q, v) = \sum_{w \in q.\mathcal{W}} tf(w, v) * idf(w, \mathcal{V}) \quad (2)$$

where $tf(w, v)$ is the term frequency of term w in object v and $idf(w, \mathcal{V})$ is the inverse document frequency of term w in the vertex set $\mathcal{V}$.

For example, consider a query $q = \langle v_4, \{v_3, v_9, v_{15}\}, \{\text{`}park\text{'}\}, 2 \rangle$ in Figure 1. Let us denote $\alpha = 0.5, M_S = 21$ and $M_T = 0.5$. Additionally, we have $\mathtt{TXDist}(q, v_3) = 0$, $\mathtt{TXDist}(q, v_9) = 0.25$ and $\mathtt{TXDist}(q, v_{15}) = 0.5$. Thus, the overall ranking scores are respectively $\mathcal{F}(q, v_3) = \frac{1}{2} \cdot \frac{6}{21} + \frac{1}{2} \cdot \frac{0}{0.5} = 0.45$, $\mathcal{F}(q, v_9) = \frac{1}{2} \cdot \frac{19}{21} + \frac{1}{2} \cdot \frac{0.25}{0.5} = 0.43$ and $\mathcal{F}(q, v_{15}) = \frac{1}{2} \cdot \frac{0}{21} + \frac{1}{2} \cdot \frac{0.5}{0.5} = 0.5$. Therefore, the top-2 answer $\mathcal{R} = \{v_{15}, v_3\}$.

It is noteworthy that G-tree can support other functions (e.g., Jaccard and BM25) for textual relevancy. In this paper, we focus on effective index and efficient algorithms.

2.2 Related Works

In this section, we introduce the related works for different types of spatial queries.

Single-Pair Shortest Path Queries: Many previous studies [1], [5], [9], [10], [20], [21] addressed the SPSP problem

1041-4347 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2015.2399306, IEEE Transactions on Knowledge and Data Engineering

on road networks. *TNR* [1] and *CH* [5] are the state-of-the-art methods. The idea of *TNR* is to figure out a set of transit nodes t_{v_x} for every vertex v_x on the road network, and pre-compute all pairs of distances between these transit nodes and the distance between a vertex to its transit nodes. Then, the shortest path between u and v is the minimal one of $\texttt{SPDist}(u, t_u) + \texttt{SPDist}(t_u, t_v) + \texttt{SPDist}(t_v, v)$. However, *TNR* incurs a large amount of memory, and its performance is poor for local queries (where the two vertices are close). *CH* first pre-computes a "contracted" version of the original road network by appending additional edges, and then uses bidirectional Dijkstra algorithm to search for the shortest paths. Though *CH* incurs a lower memory overhead, it has to visit a large number of vertices when two vertices are remote. Note that, though *TNR* is very efficient on SPSP queries, it is hard to be extended to support kNN queries efficiently. The key of kNN search is to prune those unpromising objects, however *TNR* fails to provide any additional information (e.g., boundaries) to support this.

Although *HEPV* [9] and *HiTi* [10] also organize the road network into a hierarchical structure, they suffered from tremendous storage overhead for maintaining large numbers of path information, thus they can only afford at most three hierarchical levels. Furthermore, they still use 'half-blind' Dijkstra-like network expansion algorithm, which is totally different from the dynamic programming scheme of `G-tree` based on the assembly-based method.

K Nearest Neighbor Queries on Road Network: [13], [16], [19] addressed the kNN search problem on road networks. *INE* [16] extended the Dijkstra algorithm by expanding neighbor vertices from the query location until kNN answers have been found. *IER* [16] 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 neither capture the global distance from objects to the query location nor prune unnecessary objects efficiently.

ROAD [13] extends 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 widely scattered. For example, if the objects are widely distributed (e.g., 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* adopts the network partition to form a hierarchical structure, pre-computes shortest distances between borders and uses bottom-up method to derive those distances, the idea and technique of `G-tree` differ greatly as follows. First, the index structure is different. *ROAD* implements "Route Overlay(RO)" and "Association Directory(AD)" to organize the pre-computed shortcuts and objects in a B$^+$-Tree. However, its query cost is high since each node expansion will traverse a tree-like entry under RO, especially when objects are widely scattered. `G-tree` employs light-weight distance matrices, and each distance matrix on `G-tree` node is accessed only once for a single query. Second, the algorithms of finding the kNN answers 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 complexity significantly. Thus our method significantly outperforms *ROAD* (see Section 7).

SILC [19] 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 [2], [6], [7], [12]. They pre-compute and materialize results of potential queries. However these approaches highly depend on the given object set. They also involve high pre-computation cost and large memory overhead.

This paper is a major-value added version of our previous work [24]. The new contributions include (1) We considered three types of spatial queries under the framework of `G-tree` and added how to support single-pair shortest path query in Section 4.1 and keyword-based kNN query in Section 4.3. We improved the kNN search algorithm in Section 4.2; (2) We conducted new experiments to evaluate `G-tree` in Section 7.3.1 and new experiments to evaluate keyword-based kNN query in Section 7.3.3; (3) We discussed the path recovery issue in Section 5.

Keyword-based kNN Query on Road Network: [14], [17], [18] studied the problem of keyword-based kNN query on road networks. However these works are all different from ours, both in the problem definition and the query processing techniques. [17] focuses on keyword-based kNN search without combining textual relevancy and spatial distance. [14] studies keyword-based kNN search in the distributed environment. Though [18] considers both textual and spatial relevancy between the query location and the object, it assumes that $\mathcal{C} = \mathcal{V}$, thus the availability is low. Moreover, [18] supposes objects are on edges, which is different from our assumption that objects are on vertices. Furthermore, [18] still employs a Dijkstra-like expansion algorithm to locate objects, which is rather inefficient for objects that are far away from the query location.

IR-Tree [3] and [4] studied the problem of finding top-k spatial objects on metric space. Though the textual inverted list of `G-tree` is inspired by the idea of pseudo documents in IR-Tree, the motivations are different. IR-Tree focused on efficient solutions to handle textual-spatial queries on metric space, while our goal is to devise a tree index on road

1041-4347 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2015.2399306, IEEE Transactions on Knowledge and Data Engineering

network which provides efficiency, scalability and good extensibility for various types of queries.

kNN Queries for Moving Objects on Road Network: There are a number of studies on kNN queries for moving objects monitoring [8], [15], [22], [23]. 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 static objects (e.g., gas stations), and the ideas and techniques are different.

3 THE G-tree INDEX

In this section, we first introduce the basic idea and the overview of our index in Section 3.1. Then, we formally define the G-tree in Section 3.2 and present how to construct the G-tree in Section 3.3. Finally, we discuss the space complexity of the G-tree in Section 3.4.

3.1 Basic Idea and Overview

R-tree is a height-balanced index structure that has been widely adopted for spatial data because of its efficiency and scalability. On one hand, it can prune unpromising subtrees at query time to reduce the search space. On the other hand, it can adapt to different types of queries, e.g., kNN query and spatial keyword query. Inspired by its salient features, our goal is to devise a similar index on road networks. To this end, we recursively partition the road network into sub-networks and construct a tree-structured index on top of the sub-networks where each G-tree node corresponds to a sub-network. Figure 2 shows an example of G-tree. (We will formally define the G-tree in Section 3.2)

In addition, it is easy to calculate the Euclidean distance between two vertices (or a vertex and a R-tree node) in $\mathcal{O}(1)$ time and space complexity. However on road networks, it is challenging to compute the shortest-path distance between two vertices (or a vertex and a G-tree node). If we pre-compute all shortest-path distances between two vertices and materialize them, it will take $\mathcal{O}(|\mathcal{V}|^2)$ space storage and $\mathcal{O}(|\mathcal{V}|^3)$ pre-precessing time cost, and the query time cost is $\mathcal{O}(1)$. If we calculate the distance online, the space complexity is $\mathcal{O}(|\mathcal{V}|)$ and the time complexity is $\mathcal{O}(|\mathcal{V}| \log |\mathcal{E}|)$. Obviously, these two extreme paradigms are neither efficient nor scalable. Thus it calls for an efficient method to compute shortest-path distances with high efficiency and good scalability (i.e., low storage cost). To address this issue, we devise an *assembly-based method* to efficiently compute the shortest-path distance, which only pre-computes and materializes essential pairs of shortest-path distances on G-tree (Figure 2(b)), so that any shortest-path distance can be assembled piece by piece with this materialization. Furthermore, it costs only $\mathcal{O}(|\mathcal{V}| \log |\mathcal{V}|)$ space overhead, and $\mathcal{O}(|\mathcal{V}|)$ time complexity. Therefore, G-tree can be scalable to very large datasets. We formally define the G-tree in the next section, and present the details of the *assembly-based method* in section 4.1.

3.2 Definition of G-tree

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 $(u, v) \in \mathcal{E}_i$.

To differentiate those *portal* vertices in each subgraph, 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 $\mathcal{E}_i \supseteq \mathcal{E}_j$.

Now, we formally define the G-tree as follows.

Definition 3 (G-tree): 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 this node, and the value of each entry is the shortest-path distance between the border and the vertex.

(5) For keyword queries, each node also stores additional information, e.g. inverted lists, boundaries, etc..

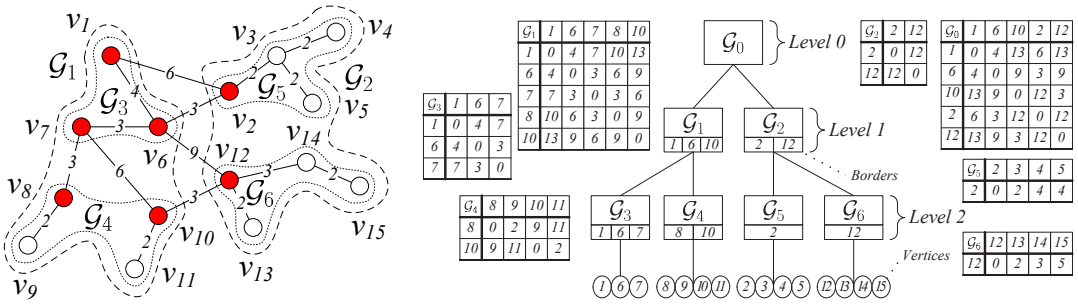
Properties (1)-(3) ensure that the G-tree has a balanced search tree structure. It is worth noting that for each node we do not maintain the physical subgraph. Instead, we only maintain a dummy subgraph ID. To support local Dijkstra and path recovery on leaf nodes, we store the original graph $\mathcal{G} = \langle \mathcal{V}, \mathcal{E} \rangle$ individually. 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

Property (4) is used to compute the shortest-path distances based on the assembly-based method, which will be discussed in Section 4.1. It is worth noting that these matrices are not actually stored on G-tree nodes but maintained on the continuous memory/disk storage space and only an offset pointer is maintained on each node.

Property (5) makes G-tree adaptive for various queries. Boundaries can be added to do pruning at query time, and keyword inverted list can be employed to support keyword search. We will discuss the details in Section 4.

1041-4347 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2015.2399306, IEEE Transactions on Knowledge and Data Engineering



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

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

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. 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/columns 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, 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.

3.3 G-tree Construction

In this section, we present how to construct the G-tree. First, we discuss how to establish a tree structure from the road network with the help of graph partition algorithm. Then, we introduce how to calculate the distance matrices.

3.3.1 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. 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 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 optimal graph partitioning is NP-Hard. In this paper, we adopt a famous heuristics algorithm, called the multilevel partitioning algorithm [11]. It first reduces the graph size by coarsening the vertices and edges, and then partitions on the coarsened graph using traditional graph partitioning algorithms, e.g. Kernighan-Lin algorithm. Finally it uncoarsens the subgraphs to generate partitions of the original graph. Moreover, the multilevel partition algorithm can guarantee that each subgraph nearly has the same size and thus G-tree is a balanced search tree structure.

3.3.2 Distance Matrices

For distance matrices of G-tree, we need to compute the shortest-path distance between a border and a border(non-leaf node)/vertex(leaf node). We can exploit a single source shortest-path algorithm, e.g. Dijkstra algorithm, which starts from each border/vertex within one G-tree node, and expands the edges until all borders of such a node have been reached. In Section 6.1, we will introduce an efficient bottom-up algorithm to speed up this procedure.

3.4 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 [19]. 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, the f-partition on the parent totally contains $\mathcal{O}(\log_2 f \cdot \sqrt{|\mathcal{V}|/f^{i-1}})$ borders. The f children of the parent share these borders. Thus each node at level i has $\mathcal{O}(\log_2 f \cdot \sqrt{|\mathcal{V}|/f^{i+1}})$ borders on average. As there are f^i nodes, the number of borders in level i is $\mathcal{O}(\log_2 f \cdot \sqrt{|\mathcal{V}|f^{i-1}})$. 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 Matrices: The average number of borders in a leaf node is $\mathcal{O}(\log_2 f \cdot \sqrt{|\mathcal{V}|/f^{\mathcal{H}+1}}) = \mathcal{O}(\log_2 f \cdot \sqrt{\tau})$. The number of vertices in a leaf node is τ. Thus the distance-matrix size of a leaf node is $\mathcal{O}(\log_2 f \cdot \tau^{1.5})$. 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 the 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 $\mathcal{O}(\mathcal{H}\log_2^2 f \cdot |\mathcal{V}|) = \mathcal{O}(\log_2^2 f \cdot \log_f \frac{|\mathcal{V}|}{\tau} \cdot |\mathcal{V}|)$.

Road Networks: A hash table leaf(v) is maintained to map a vertex to the corresponding leaf node with cost $\mathcal{O}(|\mathcal{V}|)$. We also store the original graph for local Dijkstra search and path recovery (Section 5) with cost $\mathcal{O}(|\mathcal{E}|+|\mathcal{V}|)$.

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

4 Search Algorithm

In this section, we first present the search algorithms for three types of queries: SPSP, kNN and k^2N^2 in Sec-

1041-4347 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2015.2399306, IEEE Transactions on Knowledge and Data Engineering

tions 4.1, 4.2, and 4.3 respectively. Then we analyze the time and space complexity in Section 4.4.

4.1 Single-Pair Shortest Path Query

Given a query $q = \langle u, v \rangle$, the single-pair shortest-path query returns $\texttt{SPDist}(u, v)$. However, calculating shortest-path distance on road networks is either a time-consuming or space-consuming task. To address this problem, we propose an *assembly-based method* which pre-computes and materializes essential pairs of graph distances on G-tree, so that any shortest-path distance can be assembled piece by piece using the materialized pairs.

Formally we consider two general cases: (1) u and v are in different leaf nodes of the G-tree, and we use function MINDIST-OUTSIDE-LEAF; (2) u and v are in the same leaf node, and we use function MINDIST-INSIDE-LEAF.

MINDIST-OUTSIDE-LEAF: For ease of presentation, we first introduce a closure-based method.

Closure-based Method: We introduce the notion of closure as formalized in Lemma 1, and

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 prove it by contradiction. Let $\texttt{SP}(u, v) = uv_1v_2 \cdots v_xv$. Suppose $v_1, v_2, \cdots, v_x, v$ are not borders in $\mathcal{B}(\mathcal{G}')$. If $v_x \notin \mathcal{G}'$, then v is a border based on the definition which contradicts with the assumption. If $v_x \in \mathcal{G}'$, we find the vertex with the largest subscript which is in $\mathcal{G}'$, e.g., v_i. Then $v_{i-1} \notin \mathcal{G}'(v_0 = u)$. Based on the definition of border, v_i is a border which contradicts with the assumption. Thus the lemma is proved. □

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.

Based on Lemma 1, we know that any path from u to v can be decomposed into two parts: the first part is from u to the border of a G-tree leaf node which v belongs to; and the second part is from such border to v. Thus,

$$\texttt{SPDist}(u, v) = \min_{b \in \mathcal{B}(\texttt{leaf}(v))} \big(\texttt{SPDist}(u, b) + \texttt{SPDist}(b, v)\big). \tag{3}$$

where $\texttt{leaf}(v)$ denotes the leaf node that contains v. Basically, $\texttt{leaf}(v)$ is implemented by a hash table which maps a vertex to the corresponding leaf node. Equation 3 suggests that we only need to materialize the distances between vertices and borders, and utilize these distances to compute any $\texttt{SPDist}(u, v)$. As the number of borders is less than the total number of vertices, this method seems to work. However, in the worst case, the total number of borders is still $\mathcal{O}(|\mathcal{V}|)$(see Section 3.4). To address this issue, we propose an assembly-based method.

Assembly-based Method: To overcome the problem of high space overhead in the closure-based method, we propose a novel way which only uses $\mathcal{O}(|\mathcal{V}| \log |\mathcal{V}|)$ space overhead and $\mathcal{O}(|\mathcal{V}|)$ time complexity.

The assembly-based method is the generalized version of the closure-based method. Rather than decomposing one path from u to v into two sub-paths by a single border, i.e. (u, b) and (b, v), the it decomposes one path into $m+1$ sub-paths by m borders, i.e. $(u, b_1)(b_1, b_2), \ldots, (b_m, v)$, while these m borders belong to m *distinct* and *consecutive* G-tree nodes on the path from $\texttt{leaf}(u)$ to $\texttt{leaf}(v)$. Figure 3 illustrates this procedure.

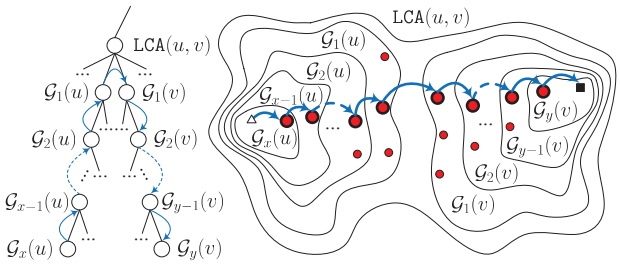


Fig. 3. A sketch of the assembly-based method.

Formally, let $\texttt{LCA}(u, v)$ denote the least common ancestor of nodes $\texttt{leaf}(u)$ and $\texttt{leaf}(v)$. Based on Lemma 1, any path from u to v must bypass a series of G-tree nodes, $\mathcal{G}_x(u)$=$\texttt{leaf}(u)$, $\mathcal{G}_{x-1}(u)$, $\cdots$, $\mathcal{G}_1(u)$, $\mathcal{G}_1(v)$, $\mathcal{G}_2(v)$, $\cdots$, $\mathcal{G}_y(v)$=$\texttt{leaf}(v)$. To compute the shortest-path distance from u to v, we consider the following cases.

(1) The shortest-path distance from u to $\mathcal{G}_x(u)$. In this case,

$$\texttt{SPDist}(u, \mathcal{G}_x(u)) = \min_{u_i \in \mathcal{B}(\mathcal{G}_x(u))} \big(\texttt{SPDist}(u, u_i)\big). \tag{4}$$

where $\texttt{SPDist}(u, u_i)$ is materialized in the distance matrix of the leaf node $\mathcal{G}_x(u)$.

(2) The shortest-path distance from u to $\mathcal{G}_{i-1}(u)$. As 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, we have,

$$\begin{aligned}\texttt{SPDist}(u, \mathcal{G}_{i-1}(u)) = \min_{u_i \in \mathcal{B}(\mathcal{G}_i(u))} \big(&\texttt{SPDist}(u, u_i) \\ &+ \texttt{SPDist}(u_i, \mathcal{G}_{i-1}(u))\big).\end{aligned} \tag{5}$$

where $\texttt{SPDist}(u, u_i)$ can be computed iteratively.

(3) The shortest-path distance from u to $\mathcal{G}_1(v)$. As the shortest path from u to $\mathcal{G}_1(v)$ must contain a border in $\mathcal{G}_1(u)$, we have,

$$\begin{aligned}\texttt{SPDist}(u, \mathcal{G}_1(v)) = \min_{u_1 \in \mathcal{B}(\mathcal{G}_1(u))} \big(&\texttt{SPDist}(u, u_1) \\ &+ \texttt{SPDist}(u_1, v_1)\big).\end{aligned} \tag{6}$$

Based on Equations 4, 5 and 6, we have

$$\begin{aligned}\texttt{SPDist}(u, v) = \min_{u_x \in \mathcal{B}(\mathcal{G}_x(u))} \Big(&\texttt{SPDist}(u, u_x) + \\ &\min_{u_{x-1} \in \mathcal{B}(\mathcal{G}_{x-1}(u))} \big(\texttt{SPDist}(u_x, u_{x-1}) + \\ &\cdots + \min_{u_1 \in \mathcal{B}(\mathcal{G}_1(u))} \big(\texttt{SPDist}(u_2, u_1) + \\ &\min_{v_1 \in \mathcal{B}(\mathcal{G}_1(v))} \big(\texttt{SPDist}(u_1, v_1) + \\ &\min_{v_2 \in \mathcal{B}(\mathcal{G}_2(v))} \big(\texttt{SPDist}(v_1, v_2) + \\ &\cdots + \min_{v_x \in \mathcal{B}(\mathcal{G}_x(v))} \big(\texttt{SPDist}(v_x, v)\Big).\end{aligned} \tag{7}$$

1041-4347 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2015.2399306, IEEE Transactions on Knowledge and Data Engineering

Equation 7 indicates that we can use a *dynamic-programming algorithm* to efficiently calculate $\texttt{SPDist}(u, v)$. Based on the distance matrices on G-tree, we first compute $\texttt{SPDist}(u, u_x)$, $u_x \in \mathcal{B}(\mathcal{G}_x(u))$. Next, we move forward to the next level to compute $\texttt{SPDist}(u, u_{x-1})$ for $u_{x-1} \in \mathcal{B}(\mathcal{G}_{x-1}(u))$, and iteratively we can reach $\mathcal{G}_1(u)$. Then we cross from $\mathcal{G}_1(u)$ to $\mathcal{G}_1(v)$, and move to the other branch. Iteratively, we can finally compute $\texttt{SPDist}(u, v)$.

To summarize, given two vertices u and v where $u \notin \texttt{leaf}(v)$, to compute $\texttt{SPDist}(u, v)$, we first compute their least common ancestor and the nodes on the paths from $\texttt{LCA}(u, v)$ to $\texttt{leaf}(u)$ and $\texttt{leaf}(v)$. Then we use dynamic programming to compute $\texttt{SPDist}(u, v)$. Figures 4 and 5 show how to implement MINDIST-OUTSIDE-LEAF.

We had proved that this paradigm takes only $\mathcal{O}(|\mathcal{V}| \log |\mathcal{V}|)$ space overhead in Section 3.4, and we will show its computation cost is $\mathcal{O}(|\mathcal{V}|)$ in Section 4.4. Compared with previous work [19](which takes $\mathcal{O}(|\mathcal{V}|^{1.5})$ space), our method significantly reduces the time/space overhead and can scale up to very large datasets. Furthermore, we will show the advantages of our assembly-based method to support other queries, e.g., kNN queries.

MINDIST-INSIDE-LEAF: Given two vertices u and v, and $u, v \in \texttt{leaf}(u)$, consider a shortest path $\texttt{SP}(u, v)$. There are two cases: (1) $\texttt{SP}(u, v)$ does not contain a vertex outside node $\texttt{leaf}(u)$. In this case, we use the Dijkstra algorithm to compute the shortest path in node $\texttt{leaf}(u)$. Let $\texttt{DijkDist}(u, v)$ denote the distance inside $\texttt{leaf}(u)$. Since the subgraph w.r.t. the leaf node is not large, the Dijkstra algorithm is efficient enough. (2) $\texttt{SP}(u, v)$ contains a vertex outside node $\texttt{leaf}(u)$. In this case, $\texttt{SP}(u, v)$ must contain two borders b_1, b_2 in $\texttt{leaf}(u)$. Let $\texttt{BorderDist}(u, v)$ denote the shortest distance from u to v with outside vertices,

$$\texttt{BorderDist}(u, v) = \min_{b_1, b_2 \in \mathcal{B}(\texttt{leaf}(u))} \big(\texttt{SPDist}(u, b_1) + \texttt{SPDist}(b_1, b_2) + \texttt{SPDist}(b_2, v)\big). \quad (8)$$

Based on the two cases, we have,

$$\texttt{SPDist}(u, v) = \min\big(\texttt{BorderDist}(u, v), \texttt{DijkDist}(u, v)\big). \quad (9)$$

Algorithm 1 shows the pseudo-code of this algorithm.

Algorithm 1: SPSPSEARCH ($q = \langle u, v \rangle$, $\mathcal{G}$)

Input: $q = \langle u, v \rangle$: A query; $\mathcal{G}$: A G-tree
Output: $\mathcal{R}$: $\texttt{SPDist}(u, v)$;

```
Locate leaf(u) and leaf(v) by a hash table ;
if leaf(u) = leaf(v) then
    R = min(BorderDist(u, v), DijkDist(u, v)) ;
else
    Find a path N from leaf(u) to leaf(v) on G ;
    foreach n_i ∈ N do
        Calculate SPDist(u, b ∈ B(n_i)) upon previous
        SPDist(u, b ∈ B(n_{i-1})) ;
```

Example 2: Figure 4 and Figure 5 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 which maps vertex to leaf node id. 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 4 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 v_4, v_2, v_6, v_8 and v_9.*

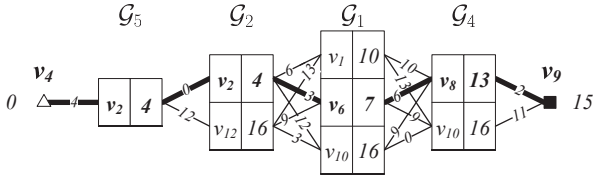


Fig. 4. Dynamic Programming of MINDIST-OUTSIDE-LEAF.

4.2 K Nearest Neighbor Query

Given a query $q = \langle v_q, \mathcal{C}, k\rangle$, the k nearest neighbor query returns top-k objects in $\mathcal{C}$ ranked by distance to v_q. Basically, kNN query is more difficult than SPSP query, since kNN search involves multiple path distance calculation (i.e., $\texttt{SPDist}(v_q, v \in \mathcal{C})$) and top-$k$ ranking. However, if we take full advantage of the assembly-based method, we can support kNN query easily.

The first issue is how to effectively calculate $\texttt{SPDist}(v_q, v \in \mathcal{C})$. We observe that, different shortest paths started from v_q share common sub-paths. For example, in Figure 2(a), consider two paths from v_4 to v_9 and v_{15}. The shortest path from v_4 to v_9 is $v_4v_3v_2v_6v_7v_8v_9$ and the shortest path from v_4 to v_{15} is $v_4v_3v_2v_6v_{12}v_{14}v_{15}$. The two paths share the common sub-path $v_4v_3v_2$, which is exactly the path of $\texttt{SPDist}(v_4, \mathcal{G}_2)$. This indicates that, when applying the assembly-based method, we can materialize the intermediate results, i.e., $\texttt{SPDist}(v_q, b_i \in \mathcal{G}_i)$, on each G-tree node, thus further calculation can be based on previous materialized results. Obviously, each G-tree node will be at most accessed once for one kNN query, therefore, such materialization-based method can avoid the duplicated computation. In Section 4.4, we will show it only takes $\mathcal{O}(\frac{\log_2 f}{\sqrt{\tau}}|\mathcal{V}|)$ space complexity.

The second issue is how to efficiently obtain the top-k answers. Intuitively, the nearer those subgraphs (nodes) are to v_q, the greater likelihood that they may contain the top-k objects. To this end we utilize a priority-queue to manage those traversed G-tree nodes. In this way, we can always dequeue the most promising (nearest) nodes each time. So the problem is to calculate the minimal distance between v_q and a G-tree node n, denoted by $\texttt{SPDist}(v_q, n)$. Based on the assembly-based method, we have

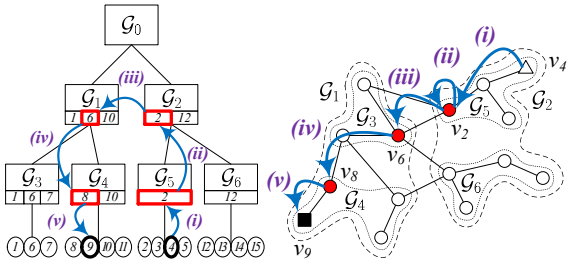
$$\texttt{SPDist}(v_q, n) = \min_{b \in \mathcal{B}(n)} \texttt{SPDist}(v_q, b) \quad (10)$$

As $\texttt{SPDist}(v_q, b)$ is calculated and materialized progressively at query time, hence, we only need $\mathcal{O}(1)$ time to figure out $\texttt{SPDist}(v_q, n)$. This implies that, during the process of the step-by-step node expansion on the G-tree, we can exploit the intermediate materialization to obtain $\texttt{SPDist}(v_q, n)$, which is critical and indispensable for top-k finding and effective pruning. Thus G-tree is efficient enough to deal with very large datasets for kNN queries.

Notice that, based on the candidate object set $\mathcal{C}$ specified by the user, we can filter out those unpromising G-tree

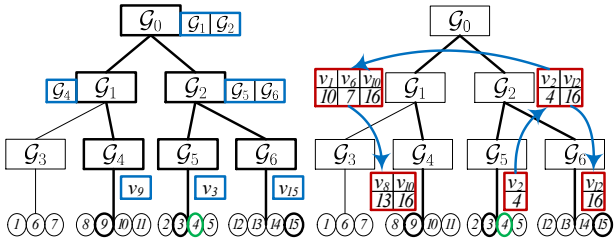
1041-4347 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2015.2399306, IEEE Transactions on Knowledge and Data Engineering



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

Fig. 5. An Example of the Assembly-based Method.



(a) Occurrence List. (b) Materialized Improvement.

Fig. 6. Occurrence List and kNN Search.

nodes before enqueued into the priority-queue. To this end, we introduce the concept of *occurrence list, i.e.,* $\mathcal{L}(n)$. *The occurrence list for a leaf node is the list of objects in this node; the occurrence list of a non-leaf node is a list of its children who contain objects. Figure 6(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 in* $\mathcal{C}$. *The benefit of the occurrence list is to decouple* G-tree*'s structure with object information, thus* G-tree *can effectively support various object sets.*

Algorithm 2 shows the pseudo-code of our algorithm, which works as follows. First of all, we locate the leaf node of query location v_q and objects $\mathcal{C}$ by $\texttt{leaf}(v)$, and construct the occurrence lists in a bottom-up manner(line 1). Then we create a priority queue $\mathcal{Q}$ and a result set $\mathcal{R}$ (line 2). Initially we use MINDIST-INSIDE-LEAF as discussed in Section 4.1 to calculate $\texttt{SPDist}(v_q, v \in \mathcal{L}(\texttt{leaf}(v_q))$, and put $\langle v, \texttt{SPDist}(v_q, v)\rangle$ into $\mathcal{Q}$ (line 4). Next, we iteratively dequeue the first element e of $\mathcal{Q}$ and handle it separately according to whether e is an object, or a G-tree node (line 6 to line 15).

It is worth noting that, for the purpose of effective pruning, we set up a global pointer $\mathcal{T}_n$, which keeps track of the current uppermost node ever visited, together with the minimum graph distance $\mathcal{T}_{min}$ within $\mathcal{T}_n$ from v_q, i.e., $\mathcal{T}_{min} = \min_{b \in \mathcal{B}(\mathcal{T}_n)} \texttt{SPDist}(v_q, b)$(line 3). Initially, $\mathcal{T}_n$ is set to $\texttt{leaf}(v_q)$. $\mathcal{T}_{min}$ signifies the minimum graph distance to step off the current $\mathcal{T}_n$. Hence, once we dequeue an object e from $\mathcal{Q}$ whose $\texttt{SPDist}(v_q, e) \leq \mathcal{T}_{min}$, we can confirm that e is a top-k answer (line 12). Otherwise, we should move the $\mathcal{T}_n$ to its parent node, expand its sibling nodes into $\mathcal{Q}$, and update the $\mathcal{T}_{min}$(line 10).

Example 3: Consider a query $q = \{v_4, \{v_3, v_9, v_{15}\}, 2\}$ *on the graph in Figure 1(a). We first construct the occurrent list and then compute the top-2 answers as follows.*

Step 1: *Enqueue the object in* $\mathcal{L}(\mathcal{G}_5)$, *i.e.* $\langle v_3 \in \mathcal{L}(\mathcal{G}_5), \texttt{SPDist}\ (v_4, v_3) = 2\rangle$. *Initially,* $\mathcal{T}_n = \mathcal{G}_5$, $\mathcal{T}_{min} = 4$.
Queue: $\langle v_3, 2\rangle$

Step 2: *Dequeue* $\langle v_3, 2\rangle$. *Since* $\texttt{SPDist}(v_4, v_3) = 2 < 4 = \mathcal{T}_{min}$, *thus,* $\mathcal{R} = \{v_3\}$. *Now,* $\mathcal{Q} = \phi$.

Step 3: *As* $\mathcal{Q} = \phi$ *but* $\mathcal{T}_n \neq \mathcal{G}_0$, *update* $\mathcal{T}_n = \mathcal{G}_2$ *and* $\mathcal{T}_{min} = 4$. *Enqueue* $\langle \mathcal{G}_6 \in \mathcal{L}(\mathcal{G}_2), \texttt{SPDist}(v_4, \mathcal{G}_6) = 16\rangle$.
Queue: $\langle \mathcal{G}_6, 16\rangle$

Step 4: *Dequeue* $\langle \mathcal{G}_6, 16\rangle$. *Since* $\texttt{SPDist}(v_4, \mathcal{G}_6) = 16 > 4 = \mathcal{T}_{min}$, *update* $\mathcal{T}_n = \mathcal{G}_0$ *and* $\mathcal{T}_{min} = \infty$. *Enqueue* $\langle \mathcal{G}_1 \in \mathcal{L}(\mathcal{G}_0), \texttt{SPDist}(v_4, \mathcal{G}_1) = 7\rangle$. *Enqueue* $\langle \mathcal{G}_6, 16\rangle$ *back.*
Queue: $\langle \mathcal{G}_1, 7\rangle$ | $\langle \mathcal{G}_6, 16\rangle$

Step 5: *Dequeue* $\langle \mathcal{G}_1, 7\rangle$. *Find* $\mathcal{G}_4 \in \mathcal{L}(\mathcal{G}_1)$. *Get* $\texttt{SPDist}\ (v_4, \mathcal{G}_4) = 13$ *and enqueue* $\langle \mathcal{G}_4, 13\rangle$.
Queue: $\langle \mathcal{G}_4, 13\rangle$ | $\langle \mathcal{G}_6, 16\rangle$

Step 6: *Dequeue* $\langle \mathcal{G}_4, 13\rangle$. *Locate* $v_9 \in \mathcal{L}(\mathcal{G}_4)$. *Calculate* $\texttt{SPDist}\ (v_4, v_9) = 15$ *and enqueue* $\langle v_9, 15\rangle$.
Queue: $\langle v_9, 15\rangle$ | $\langle \mathcal{G}_6, 16\rangle$

Step 7: *Dequeue* $\langle v_9, 15\rangle$. *Since* $\texttt{SPDist}(v_4, v_9) = 15 < \infty = \mathcal{T}_{min}$, $\mathcal{R} = \{v_3, v_9\}$. *Top-2 answers have been generated. Algorithm terminates.*

Algorithm 2: 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  foreach v ∈ L(leaf(v_q)) do
4      Q.Enqueue(⟨v, MINDIST-INSIDE-LEAF(v_q, v)⟩) ;
5  Initialize T_n = leaf(v_q) and T_min ;
6  while R.Size() < k and (Q ≠ φ or T_n ≠ root) do
7      if Q = φ then UpdateT (T_n, T_min, Q);
8      ⟨e, dis⟩ ← Q.Dequeue() ;
9      if dis > T_min then
10         UpdateT (T_n, T_min, Q) ;
11         Q.Enqueue(⟨e, dis⟩) ;
12     else if e is an object then Insert e into R ;
13     else if e is a node then
14         foreach c ∈ L(e) do
15             Q.Enqueue(⟨c, SPDist(v_q, c)⟩) ;
16 Function UpdateT (T_n, T_min, Q)
17     T_n ← T_n.father and update T_min ;
18     foreach c ∈ L(T_n) do
19         Q.Enqueue(⟨SPDist(v_q, c)⟩) ;
```

4.3 Keyword-based kNN Query

Given a query $q = \langle v_q, \mathcal{C}, \mathcal{W}, k\rangle$ and the ranking function $\mathcal{F}$, the k^2N^2 query returns top-k objects in $\mathcal{C}$ based on the ranking function $\mathcal{F}$. Compared with kNN search, k^2N^2 is more challenging since it considers not only spatial distance but also textual relevancy for ranking. In this section, we extend the kNN algorithm to efficiently answer k^2N^2 queries. From previous sections, the most important component of kNN search is the distance calculation between the query location and a G-tree node, i.e. $\texttt{SPDist}(v_q, n)$ in Equation 10, which is used as the

1041-4347 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2015.2399306, IEEE Transactions on Knowledge and Data Engineering

distance boundary for effective pruning. Similarly, we can define the textual boundary between a query q and a node n, denoted by $\texttt{TXDist}(q, n)$, and then aggregate the spatial and textual boundary to obtain the overall boundary $\mathcal{F}(q, n)$ for pruning spatially and textually at the same time.

To this end, we add an inverted list of $\langle kw, wt\rangle$ on each node of the G-tree, which represents the maximum tf*idf, denoted wt, for keyword kw under the sub-tree of this node. Formally, given a G-tree node n and a keyword w, wt is:

$$wt(w, n) = \max_{v \in \texttt{SubTree}(n)} tf(w, v) * idf(w, \mathcal{V}) \quad (11)$$

where $\texttt{SubTree}(n)$ denotes all the vertices in the sub-tree of node n. Thus, for a query with the keyword set $q.\mathcal{W}$,

$$\texttt{TXDist}(q, n) = \sum_{w \in q.\mathcal{W}} wt(w, n) \quad (12)$$

Note that, $wt(w, n)$ is only calculated once in a bottom-up manner offline. Thus, $\texttt{TXDist}(q, n)$ can be aggregated online. Similar to the distance matrices, we maintain an offset pointer on each G-tree node, and $wt(w, n)$ is physically stored in continuous memory/disk storage. Once we get both $\texttt{SPDist}(v_q, n)$ and $\texttt{TXDist}(q, n)$, we can pack them together to obtain $\mathcal{F}(q, n)$ by Equation 1.

Based on $\mathcal{F}(q, n)$, we can devise the k^2N^2 algorithm by making minor modifications on Algorithm 2:

(1). Pre-compute the $wt(w, n)$ before the query;

(2). Replace all $\texttt{SPDist}(v_q, n)$ into $\mathcal{F}(q, n)$;

(3). Change the $\mathcal{Q}$ into a max priority-queue;

(4). Alter the intention of $\mathcal{T}_{min}$ into $\mathcal{T}_{max}$. It is worth noting that the textual part of $\mathcal{T}_{max}$ is the global one, i.e., $\sum_{w \in q.\mathcal{W}} wt(w, root)$.

Algorithm 3 presents the algorithmic description for k^2N^2. Here, we use an example to show the process.

Example 4: Consider the query $q = \langle v_4, \{v_3, v_9, v_{15}\}, \{\text{'}park\text{'}\}, 2\rangle$ *in Figure 1, where* $\alpha = 0.5, M_S = 21$ *and* $M_T = 0.5$. *To save space, we omit the process here. The top-2 answers are* $\langle v_{15}, 0.50\rangle$ *and* $\langle v_3, 0.45\rangle$.

Algorithm 3: K^2N^2 $(q = \langle v_q, \mathcal{C}, \mathcal{W}, k\rangle, \mathcal{F}, \mathcal{G})$

Input: $q = \langle v_q, \mathcal{C}, \mathcal{W}, k\rangle$: A query; $\mathcal{F}$: A ranking function; $\mathcal{G}$: A G-tree

Output: $\mathcal{R}$: The top-k result list;

1 Line 1 to Line 4 in Algorithm 2;
2 Compute global $wt(w, n)$ based on occurrence list $\mathcal{L}$;
3 Initialize $\mathcal{T}_n = \texttt{leaf}(v_q)$ and $\mathcal{T}_{max}$;
4 Line 6 to Line 15 in Algorithm 2, replace all $\texttt{SPDist}(v_q, n)$ into $\mathcal{F}(q, n)$, replace all $\mathcal{T}_{min}$ into $\mathcal{T}_{max}$;
5 **Function** UpdateT $(\mathcal{T}_n, \mathcal{T}_{max}, \mathcal{Q})$
6 $\quad \mathcal{T}_n \leftarrow \mathcal{T}_n.father$ and update $\mathcal{T}_{max}$, where $\texttt{TXDist}(v_q, \mathcal{T}_n) = \sum_{w \in q.\mathcal{W}} wt(w, root)$;
7 $\quad$ **foreach** $c \in \mathcal{L}(\mathcal{T}_n)$ **do**
8 $\quad\quad \mathcal{Q}$.Enqueue$(\langle \mathcal{F}(v_q, c)\rangle)$;

Theorem 1: Algorithms 1(SPSP), 2(kNN) and 3(k^2N^2) can correctly report all answers.

Proof: **SPSP:** According to Lemma 1, let $b_1 \in \mathcal{G}_1$, $b_2 \in \mathcal{G}_2$, $\cdots$, $b_p \in \mathcal{G}_p$ denote p consecutive borders on the shortest path of $\texttt{SP}(u, v)$. Suppose there exists another path $\texttt{SP}'(u, v) = ub'_1b'_2 \cdots b'_pv$ that $|\texttt{SP}'(u, v)| < |\texttt{SP}(u, v)|$, where $b'_x \in \mathcal{G}_x$. If $\exists i$ such that $b'_i = b_i$, then $|\texttt{SP}'(u, b'_i)| + |\texttt{SP}'(b'_i, v)| = |\texttt{SP}(u, b_i)| + |\texttt{SP}(b_i, v)|$, thus, $\forall i$ that $b'_i \neq b_i$. However, according to Equation 7, b'_1 must be selected since $|\texttt{SP}'(u, b'_1)| + |\texttt{SP}'(b'_1, v)| < |\texttt{SP}(u, b_1)| + |\texttt{SP}(b_1, v)|$, i.e., $b'_1 = b_1$, which contradicts that $b'_1 \neq b_1$. Thus SPSP can correctly find the shortest-path answer.

kNN and k^2N^2: Suppose $\exists v \in \mathcal{C} - \mathcal{R}$ that $\texttt{SPDist}(v_q, v) < \texttt{SPDist}(v_q, \bar{v} \in \mathcal{R})$, where $\texttt{SPDist}(v_q, \forall v' \in \mathcal{R}) \leq \texttt{SPDist}(v_q, \bar{v})$. Thus, $\texttt{SPDist}(v_q, \texttt{leaf}(v)) < \texttt{SPDist}(v_q, \bar{v} \in \mathcal{R})$. According to the best-first search, $\texttt{leaf}(v)$ must dequeue before $\bar{v}$, hence, v must be one of top-k answers, i.e., $v \in \mathcal{R}$. This contradicts the assumption of $v \in \mathcal{C} - \mathcal{R}$. Therefore, kNN can correctly find top-k answers. The correctness of k^2N^2 can be proved similarly. □

4.4 Time and Space Complexity

1) Single-Pair Shortest Path Query:

Time Complexity: If two vertices u and v are within the same leaf node, its time cost is $\mathcal{O}(\tau \log \tau)$ for local Dijkstra search. Otherwise, it will at most scan a series of distance matrices from $\texttt{leaf}(u)$ to $\texttt{leaf}(v)$ bypassing the *root*, and the worst-case time complexity is the size of distance matrices scanned. According to analysis in Section 3.4, we can derive the time cost to be $\mathcal{O}(2 \cdot \sum_{i=1 \cdots \mathcal{H}} \log_2^2 f \cdot |\mathcal{V}|/f^i) = \mathcal{O}(\log_2^2 f \cdot |\mathcal{V}|)$.

2) kNN Query and k^2N^2 Query:

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 materialized algorithm will only access each node of G-tree once, the worst-case time 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 overall time cost 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:

Object Occurrence List $\mathcal{L}(n)$: The occurrence list $\mathcal{L}(n)$ for leaf nodes of G-tree requires $\mathcal{O}(|\mathcal{C}|)$ space. In the worst case, $\mathcal{L}(n)$ for all non-leaf nodes is equal to $\mathcal{O}(f \cdot \frac{|\mathcal{V}|}{\tau})$. To sum up, the overall space cost is $\mathcal{O}(|\mathcal{C}| + f \cdot \frac{|\mathcal{V}|}{\tau})$. Compared with distance matrices of G-tree, the space cost of $\mathcal{L}(n)$ is negligible.

Inverted List for k^2N^2: Let $|\Psi|$ denotes $\sum_{v \in \mathcal{V}} |v.\mathcal{W}|$, i.e., the total number of keywords for all vertices. Suppose that each keyword on vertices is distinct (the worst case), thus, given a keyword w on v, $wt(w, n)$ is materialized in all nodes on the branch from $\texttt{leaf}(v)$ to the root. Therefore, the overall space cost is $\mathcal{O}(|\Psi|\mathcal{H})$.

5 SHORTEST PATH RECOVERY

It is worth noting that all the algorithms in Section 4 returns shortest-path distance rather than vertex-by-vertex path. However, the latter is important and useful (e.g., in navigation system). In this section, we discuss how to

1041-4347 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2015.2399306, IEEE Transactions on Knowledge and Data Engineering

recover the path from the query location v_q to an answer $v_a \in \mathcal{R}$.

5.1 Overview

We consider the case that v_q and v_a are in different nodes. Since we facilitate the graph distance calculation by the assembly-based method, 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 borders, 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 the divide-and-conquer technique to iteratively add new vertices into the $\texttt{SP}'(v_q, v_a)$. Given two vertices v and v', once we can find a v_x such that $\texttt{SPDist}(v, v_x) + \texttt{SPDist}(v_x, v') = \texttt{SPDist}(v, v')$, we can confirm that v_x is one vertex on $\texttt{SP}(v, v')$. For example, in Figure 7, 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 vertex-by-vertex shortest path is $\texttt{SP}(v_4, v_9) = v_4 v_3 v_2 v_6 v_7 v_8 v_9$.

5.2 Shortest Path Computation

Formally, given a section of a shortest path $\texttt{SP}'(v_q, v_a) = v_q b_1 b_2 \cdots b_m v_a$, for each two adjacent vertices pair $\langle v, v' \rangle$ in $\texttt{SP}'(v_q, v_a)$, there are generally three cases:

Case 1: v, v' are borders in different leaf nodes:

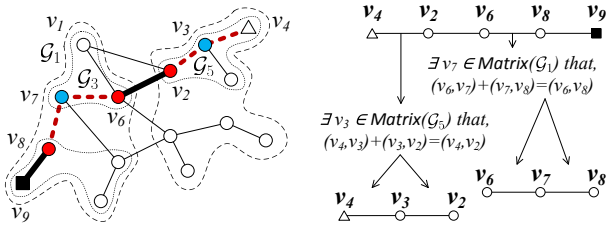
Lemma 2: Given two borders v and v' in different leaf nodes, there must exist a border b in the distance matrix from $\texttt{LCA}(v, v')$ to the *root* such that v, v', b appear in the matrix and $\texttt{SPDist}(v, v') = \texttt{SPDist}(v, b) + \texttt{SPDist}(b, v')$.

Proof: Suppose there is at least one vertex except v and v' on $\texttt{SP}(v, v')$ which is in the subgraph of $\texttt{LCA}(v, v')$. According to Lemma 1, there must exist a border b on $\texttt{SP}(v, v')$ such that v, v' and b are in the distance matrix of $\texttt{LCA}(v, v')$ by Definition 3. Otherwise, all $\texttt{SP}(v, v')$ vertices except v and v' are outside of $\texttt{LCA}(v, v')$. Since the root contains all vertices and paths, there must exist one ancestor node $\mathcal{G}'$ of $\texttt{LCA}(v, v')$ which contains at least one vertex except v and v' of $\texttt{SP}(v, v')$. Therefore, one border b must exist in the distance matrix of $\mathcal{G}'$, together with v, v'. □

For example, consider $\langle v_6, v_8 \rangle$, where $v_6 \in \mathcal{G}_3$ and $v_8 \in \mathcal{G}_4$. $\texttt{LCA}(v_6, v_8) = \mathcal{G}_1$. We find a border v_7 in the distance matrix of $\mathcal{G}_1$ such that $\texttt{SPDist}(v_6, v_8) = \texttt{SPDist}(v_6, v_7) + \texttt{SPDist}(v_7, v_8)$. Then we add v_7 into $\langle v_6, v_8 \rangle$ and check $\langle v_6, v_7 \rangle$ and $\langle v_7, v_8 \rangle$. As $\texttt{w}(v_6, v_7) = \texttt{SPDist}(v_6, v_7)$ and $\texttt{w}(v_7, v_8) = \texttt{SPDist}(v_7, v_8)$, hence, the shortest path is $v_6 v_7 v_8$.

Case 2: v, v' are borders in the same leaf nodes:

Lemma 3: Given two borders v and v' in the same leaf node, if there does not exist a non-border vertex v_x such that $\texttt{SPDist}(v, v') = \texttt{SPDist}(v, v_x) + \texttt{SPDist}(v_x, v')$, then there must exist a border b in the distance matrix of ancestors of the node such that v, v', b appear in the matrix and $\texttt{SPDist}(v, v') = \texttt{SPDist}(v, b) + \texttt{SPDist}(b, v')$.



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

Fig. 7. An Example of Shortest-path Recovery.

Proof: Lemma 3 can be proved similar to Lemma 2 on provided that $\texttt{LCA}(v, v') = \texttt{leaf}(v) = \texttt{leaf}(v')$. □

For example, consider $\langle v_1, v_7 \rangle$ in $\mathcal{G}_3$. We find a border v_6 such that $\texttt{SPDist}(v_1, v_7) = \texttt{SPDist}(v_1, v_6) + \texttt{SPDist}(v_6, v_7)$. We add v_6 into $\langle v_1, v_7 \rangle$ and check $\langle v_1, v_6 \rangle$ and $\langle v_6, v_7 \rangle$. As $\texttt{w}(v_1, v_6) = \texttt{SPDist}(v_1, v_6)$ and $\texttt{w}(v_6, v_7) = \texttt{SPDist}\ (v_6, v_7)$, the shortest path is $v_1 v_6 v_7$.

Case 3: v, v' are in the same leaf and v is not a border:

Lemma 4: Given a non-border vertex v and a border v' in the same leaf node, if there does not exist a border b such that $\texttt{SPDist}(v, v') = \texttt{SPDist}(v, b) + \texttt{SPDist}(b, v')$, then there must exist a shortest path from v to v' which only includes vertices in the leaf node.

Proof: If not all vertices on $\texttt{SP}(v, v')$ are within $\texttt{leaf}(v)$, there must exist another border b on $\texttt{SP}(v, v')$ according to Lemma 1. Otherwise, $\texttt{SP}(v, v')$ only includes vertices in $\texttt{leaf}(v)$. Thus the lemma is proved. □

For example, consider $\langle v_4, v_2 \rangle$ in $\mathcal{G}_5$. As there is only one border v_2 in $\mathcal{G}_5$, we cannot add borders into the path. We visit v_2's neighbor vertex and find vertex v_3 such that $\texttt{SPDist}(v_4, v_2) = \texttt{w}(v_4, v_3) + \texttt{SPDist}(v_3, v_2)$. We add v_3 into $\langle v_4, v_2 \rangle$. Since both (v_4, v_3) and (v_3, v_2) are edges in $\mathcal{G}$, thus the shortest path is $v_4 v_3 v_2$.

Based on these properties, we propose a recursive algorithm to recover the vertex-by-vertex path for a given $\langle v, v' \rangle$ pair, which is shown in Algorithm 4.

Algorithm 4: PREC ($\langle v, v' \rangle$, $\mathcal{G}$)

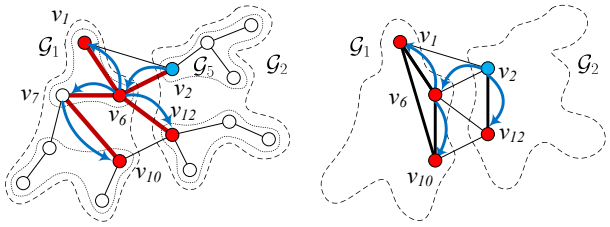
Input: $\langle v, v' \rangle$: Two adjacent vertices; $\mathcal{G}$: A G-tree
Output: $\mathcal{R}$: The vertex-by-vertex path from v to v';

```
1  if (v, v') ∈ E then return ∅ ;
2  if leaf(v) = leaf(v') then
3      b ←FIND_BORDER(LCA(v, v'), root) ;
4      if b = ∅ then
5          u ←FIND_NEIGHBOR(v) ;
6          return (v, u, PREC(u, v'), v')
7      else return (v, PREC(v, b), b, PREC(b, v'), v');
8  else if leaf(v) ≠ leaf(v') then
9      b ←FIND_BORDER(LCA(v, v'), root) ;
10     return (v, PREC(v, b), b, PREC(b, v'), v')
```

Space and Time Complexity: The space complexity is the number of vertices in $\texttt{SP}(v_q, v_a)$, denoted by N_v. Let $\mathcal{B}_{max}$ denote the maximal number of borders in each node. In each step, we add a border from the LCA node to the root with time complexity $\mathcal{O}(\mathcal{H}\mathcal{B}_{max})$ or enumerate vertices in a leaf node with time complexity $\mathcal{O}(\tau\mathcal{B}_{max})$. Thus the overall time complexity is $\mathcal{O}((\mathcal{H} + \tau)\mathcal{B}_{max} N_v)$.

1041-4347 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2015.2399306, IEEE Transactions on Knowledge and Data Engineering



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

Fig. 8. Distance Matrix Computation (From v_2).

6 Discussions

6.1 Computing Distance Matrix Efficiently

The naive way to generate the distance matrices is to calculate the graph distances node by node, and pair by pair. This is obviously ineffective for very large datasets.

To address this issue, we propose a bottom-up method to efficiently compute the distance matrices. The basic idea is that we first compute the distance matrices in the lower level and then use these distances to compute the distance matrices in the upper level(Figure 8(b)). Given two borders u and v from two sibling nodes $\mathcal{G}(u)$ and $\mathcal{G}(v)$, consider a shortest path between u and v, e.g., $\texttt{SP}(u,v) = uu_1u_2\cdots u_xw_1w_2\cdots w_zv_yv_{y-1}v_1v$. Suppose u_x is the last vertex in $\mathcal{G}(u)$ and v_y is the first vertex in $\mathcal{G}(v)$. Obviously u_x is a border of $\mathcal{G}(u)$ and v_y is a border of $\mathcal{G}(v)$. As $\texttt{SPDist}(u,u_x)$ has been computed in the lower level, we can skip vertexes $u_1u_2\cdots u_{x-1}$ and directly use the "shortcut" $\langle u,u_x\rangle$. 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.

6.2 Extension to Directed Graphs

G-tree can 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 6.1 to compute the directed distances. Second, our method relies on using the assembly-based method 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 algorithm in Section 4.1 to compute the shortest distance of a directed path. Third, we slightly modify the Dijkstra algorithm to support the directed graphs.

6.3 On Planarity of Network

Though we assume that the underlying road network is a planar graph, this is only for the purpose of analyzing the bounds of space complexity. The experimental result in Figure 10 shows that the index sizes of G-tree on real networks actually follow the derived bound, and G-tree works very well with non-planar networks, and the bridges or tunnels will not affect the correctness of the algorithms.

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

7 Experiments

Datasets: We used eight real-world datasets [1,2,3] with various sizes from 20,000 vertices to 24 million vertices, which are widely used in previous studies [13]. The statistics of these datasets are shown in Table 1.

Settings: For the G-tree, the default fanout is $f = 4$ and τ is set to 64(*CAL*), 128(*SF*), 128(*COL*), 256(*FLA*), 256(*E-USA*), 512(*C-USA*) and 512(*USA*). To evaluate the efficiency of SPSP query, we randomly chose 10,000 pairs of vertices for each dataset; to evaluate the performance of kNN and k^2N^2 query, we randomly chose 100 vertices as the query location, and for each 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**, 0.1, 1 of vertices from the dataset as objects (default is 0.01). For k, we used 1, 5, **10**, 20, 50 (default is 10).

For SPSP query, we compared our G-tree with *TNR* [1], *CH* [5] and $A*$ algorithm. For kNN query, we compared with *INE* [16], *SILC* [19] and *ROAD* [13][4]. For k^2N^2 query, since no previous works addressed the same problem as ours, we implemented a heuristic method, denoted by *NKS*, which is based on *INE* but used the k-th answer's boundary to do pruning. All the algorithms were implemented in C++. All experiments were conducted on a Linux computer with Intel 2.50GHz CPU and 16GB memory.

7.1 Evaluation on G-tree Parameters: f and τ

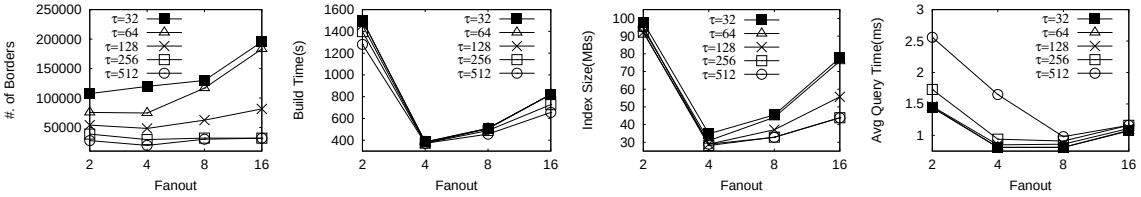
We evaluated the impact of f(the fanout) and τ(the number of vertices in a leaf node) of the G-tree by investigating the number of borders, the index size, the index build time and the query time. We varied τ in {32, 64, 128, 256, 512} and f in {2, 4, 8, 16}.

Figure 9 shows the results. We made two observations. First, with the increase of f, the number of borders and 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 reason is that larger f will generate more borders as there are more partitions, however, larger f will reduce the height of G-tree and

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
4. We set "No. of levels(l)" to 6, 7, 7 and 8 respectively on *CAL*, *SF*, *COL* and *FLA*, so that the height of *ROAD* is the same with G-tree.

1041-4347 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2015.2399306, IEEE Transactions on Knowledge and Data Engineering



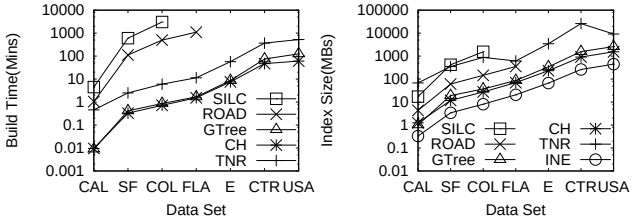
(a) Number of Borders. (b) Index Build Time(s). (c) Index Size(MB). (d) Query Time(ms).

Fig. 9. Evaluation on Parameters: f and τ (*COL* dataset, $k = 10$, 1% uniform vertices as objects).

the number of nodes that need to be partitioned. Second, with the increase of τ, the number of borders, the index size and the build time decreased, since bigger τ results in smaller tree size. However, larger τ incurs more time on local Dijkstra search within one leaf node. To balance between the query efficiency and indexing size, we selected $\tau = 128$ for *COL* dataset as a trade off.

7.2 Evaluation on `G-tree` Construction

We evaluated the time and space overhead of indexing, and compared with *SILC*, *ROAD*, *TNR* and *CH*. Figure 10 illustrates the index sizes and index construction time. Note that, the index size of `G-tree` included tree structure, distance matrices, `leaf`(u) hashtable and the original graph.



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

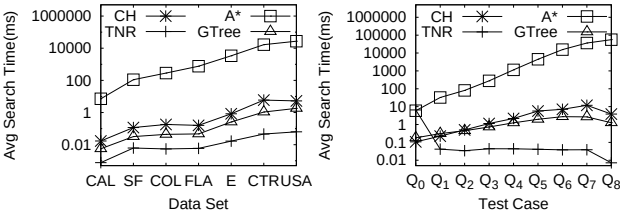
Fig. 10. Index Comparison.

`G-tree` outperformed *ROAD*, *SILC* and *TNR* both in construction time and index sizes, and achieved the same result as *CH*. For the construction time, `G-tree` was better than *ROAD* and *TNR* by an order of magnitude and three orders of magnitude than *SILC*. For index sizes, on *COL*, `G-tree` consumed 37.0 MB, *ROAD* used 145 MB and *SILC* required 1543 MB; on *USA*, `G-tree` consumed 2.6GB but *TNR* cost 9.1GB. This is because *SILC* required $\mathcal{O}(|\mathcal{V}|^{1.5})$ space to compute all-pair shortest paths. *ROAD* incurred a large number of shortcuts pre-computation. *TNR* needed to calculate the distances between all pairs of transit nodes.

7.3 Evaluation on Query Efficiency

Evaluation on SPSP Search: We compared the SPSP search efficiency of `G-tree` with *TNR*, *CH* and *A**. First, we tested the performance by randomly choosing two nodes on each dataset. Second, we tested the performance varying the distance between two nodes from near to far. We partitioned *USA* road network into 128*128 grids, and generated nine test cases $Q_i (i = 0 \dots 8)$ which indicated that two nodes were exactly 2^{i-1} grids away (For Q_0, two nodes were within one grid). The result is shown in Figure 11.

For the random dataset, `G-tree` is approximately 5 times faster than *CH*, and nearly three orders of magnitudes faster than *A**. Though *TNR* was faster than `G-tree`, it was actually trading space for time. Thus, it is not surprising that *TNR* took almost 20 times more space than `G-tree`



(a) Random Test. (b) Varying Distances(*USA*).

Fig. 11. Evaluation on SPSP Query

(see Figure 10(b)). When two nodes were within one grid, `G-tree` and *CH* were two orders of magnitudes faster than *TNR*, since *TNR* failed to handle local queries efficiently by Dijkstra algorithm.

Evaluation on kNN Search: We evaluated the kNN search efficiency of `G-tree`, *ROAD*, *SILC* and *INE* by varying the number of answers k, the number of objects $|\mathcal{C}|$, datasets, and distances from query location to top-k answers. As default, we tested on *COL* dataset, set $k = 10$, and $|\mathcal{C}| = 0.01|\mathcal{V}|$. Note that, we used 1% vertices as objects, since in most cases the number of candidate objects were relatively few compared with the size of the road network. Even in other cases, we can see that it was much easier for top-k search when $|\mathcal{C}| = |\mathcal{V}|$(Figure 12(b)), since it would quickly find top-k answers around the query location.

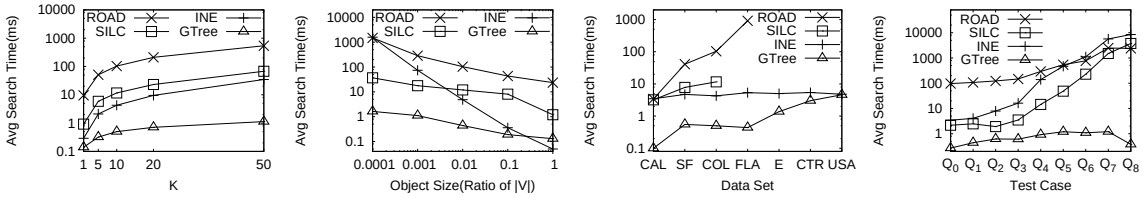
kNN Search by Varying k: The result is shown in Figure 12(a). We can see that `G-tree` outperformed the state-of-the-arts *ROAD* and *SILC* by 2-3 orders of magnitude, and outperformed *INE* by one order of magnitude. Since *SILC* had to search multiple quadtrees to find distances between query location and objects, this operation was very costly and inefficient for larger k. As *ROAD* employs an expansion-based method, it can only prune the nodes which have no objects and cannot use distance-based pruning. Note that, *INE* was faster than *ROAD* and *SILC* on this test - since objects were uniformly distributed *INE* could quickly come up with top-k answers with Dijkstra algorithm.

kNN Search by Varying Object Sizes: Figure 12(b) shows the result. We made three observations. First, the efficiency of these four methods increased for larger $|\mathcal{C}|$, as more objects implied less top-k finding overhead from query location. Second, `G-tree` outperformed three baseline methods significantly for different $|\mathcal{C}|$ settings. Third, when $|\mathcal{C}| = |\mathcal{V}|$, `G-tree` achieved comparable performance with *INE* by using identical leaf-node Dijkstra search. However, *SILC* and *ROAD* still performed poorly, as *SILC* used morton block to compress/decompress path information and *ROAD* needed to traverse numbers of tree-like entries on RO index for every vertex expansion.

kNN Search by Varying Datasets: Since *ROAD* and *SILC*

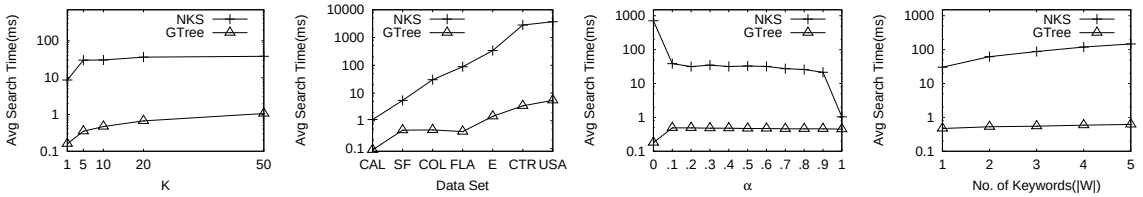
1041-4347 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2015.2399306, IEEE Transactions on Knowledge and Data Engineering



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

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



(a) Varying k. (b) Varying Datasets. (c) Varying α. (d) Varying $|\mathcal{W}|$.

Fig. 13. Performance Comparison on k^2N^2 Search (Default: *COL* dataset, $k = 10$, $|\mathcal{C}| = 0.01|\mathcal{V}|$, $|\mathcal{W}| = 1$ and $\alpha = 0.5$).

took a mass of pre-processing time and consumed large amount of memory[5], we only tested *ROAD* on *CAL*, *SF*, *COL* and *FLA*, and *SILC* on *CAL*, *SF* and *COL*. Figure 12(c) shows the result. `G-tree` outperformed *ROAD* and *SILC* on every dataset by 1-2 orders of magnitude. However, *INE*'s performance degenerated slowly with the increase of the data size. This is because that the efficiency of *INE* actually depends on the minimum distances to top-k objects. Since we had assumed that objects are uniform distributed and $|\mathcal{C}| = 0.01|\mathcal{V}|$, such distance did not change tremendously as the data size become larger. However, we will see that *INE* performed very poorly when objects were far away from the query location in the next test.

kNN Search by Varying Object Distances: We adopted the same setting which was used for SPSP queries in Figure 11(b) to guarantee the objects of Q_i were at least 2^{i-1} grids away. Figure 12(d) shows the result. `G-tree` significantly outperformed *ROAD*, *SILC* and *INE*, even by 3-4 orders of magnitude. All baseline methods performed poorly since they had to traverse long distance paths before they accessed all top-k answers. In contrast, `G-tree` can quickly locate top-k objects within fewer tree nodes, no matter how far away objects were. This test proves `G-tree`'s superiority and efficiency for kNN search.

Evaluation on k^2N^2 Search: We obtained 14 million POIs from a popular directory website[6], extracted the location and keywords, and attached each of them to the nearest vertex on the road network. Table 2 shows the statistics.

TABLE 2
Textual Statistics on Seven Datasets

Size	*CAL*	*SF*	*COL*	*FLA*	*E-USA*	*C-USA*	*USA*
$\lvert$POIs$\rvert/\lvert\mathcal{V}\rvert$	1.01	2.62	0.53	0.84	1.03	0.42	0.54
Distinct Kwd	29439	165607	97165	230863	632823	806494	1443560
Average Kwd	6.58	6.43	6.71	6.93	6.56	6.83	6.73
Min-Max Kwd	[3,21]	[3,21]	[3,21]	[3,27]	[2,25]	[3,34]	[1,36]
Inv List(MB)	5.3	46.3	44.7	147.5	527.3	993.8	2035.5

We proposed a baseline method, namely *NKS*, which adopted the framework of *INE* and used the k-th obtained answer v^k to prune unpromising Dijkstra expansion, i.e. stop the expansion at v_x when $\texttt{SPDist}(q, v_x) + \max_{v_i \in \mathcal{C}} \texttt{TXDist}(q, v_i) < \mathcal{F}(q, v^k)$. We compared `G-tree` with *NKS* by varying the number of answers k, datasets, the weight factor α and the number of keywords $|\mathcal{W}|$. As default, we tested on *COL* dataset, and set $k = 10$, $|\mathcal{C}| = 0.01|\mathcal{V}|$, $\alpha = 0.5$ and $|\mathcal{W}| = 1$.

k^2N^2 Search by Varying k: Figure 13(a) shows the result. We can see that `G-tree` outperformed *NKS* by one order of magnitude. This is because that `G-tree` can fully utilize spatial and textual pruning at the query time, thus, even for $k = 50$, `G-tree` can answer a query within 1ms on average.

k^2N^2 Search by Varying Datasets: Figure 13(b) shows that `G-tree` achieved excellent efficiency and scalability as the data size increased. Compared with *NKS*, `G-tree` was nearly 3 orders of magnitude faster on *USA*.

k^2N^2 Search by Varying α: According to Equation 1, $\alpha = 0$ considers only the textual part, while setting $\alpha = 1$ considers only the spatial part. Figure 13(c) shows the result. We can see that *NKS* improved its performance as α increased. For $\alpha = 0$, *NKS* was extremely inefficient, while it was the opposite when $\alpha = 1$. The reason is that *NKS* was simply based on Dijkstra algorithm, thus, it was incapable to handle purely textual queries effectively. Note that `G-tree` is more efficient when $\alpha = 0$, as the spatial distance calculation was not required in this case.

k^2N^2 Search by Varying $|\mathcal{W}|$: We can see that `G-tree` still outperformed *NKS* by at least 1 order of magnitude, which is shown in Figure 13(d). Note that, the variation on $|\mathcal{W}|$ did not affect `G-tree`'s performance significantly, and had minor impact on *NKS*. The reason is that, according to Equations 1 and 2, the increase of $|\mathcal{W}|$ is actually augmenting the weight of the textual part. Hence, this is equal to decreasing the α to some extent.

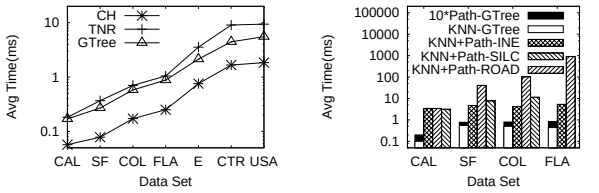
7.4 Evaluation on Path Recovery

For SPSP query, we compared `G-tree` with *TNR* and *CH*. For kNN query, we compared with *ROAD*, *SILC* and *INE*. We used the same setting as the existing experiments. The result is shown in Figure 14.

5. We estimated 4.8 days for *ROAD*, and 36.5GB for *SILC* on *USA*.
6. http://www.factual.com

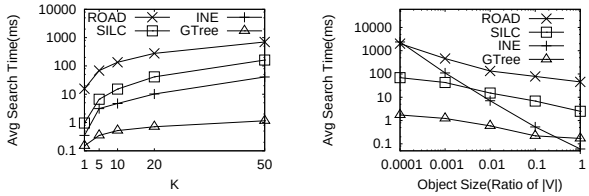
1041-4347 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TKDE.2015.2399306, IEEE Transactions on Knowledge and Data Engineering



(a) On SPSP Path Recovery. (b) On kNN Path Recovery.

Fig. 14. Evaluation on Path Recovery.



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

Fig. 15. Evaluation on Directed Graphs.

For SPSP, G-tree outperformed *TNR* but 5 times slower than *CH*. However, G-tree was 5 times faster than *CH* for distance query in Figure 11(a). For kNN, G-tree still beats *ROAD*, *SILC* and *INE* even by comparing the total time(1*kNN +10*Path Recovery).

7.5 Scalability

Figure 10(b) indicated that the index size of G-tree increased linearly with the increases of the data size. Figure 11(a), 12(c) and 13(b) had displayed G-tree's superior scalability for three different types of queries.

7.6 Evaluation on Directed Graph

We evaluated the search efficiency of G-tree on a directed graph *WA*. We set $f = 4$ and $\tau = 128$. The G-tree index was 52.13MB and the index building time was 16s. We compared the kNN performance with *SILC*, *ROAD* and *INE* by varying k and object size $|\mathcal{C}|$, as shown in Figure 15. G-tree still significantly outperformed existing methods. The results are consistent with those on undirected graphs.

8 Conclusion

In this paper we have proposed an efficient and scalable index on road networks. We devised an assembly-based method to efficiently calculate shortest-path distance. We proposed efficient search algorithms to answer SPSP, kNN and k^2N^2 queries. We also discussed path recovery issue. Experimental results show G-tree's theoretical and practical superiority over state-of-the-art methods.

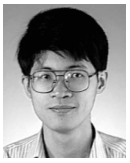
Acknowledgement. This work was partly supported by the 973 Program of China (2015CB358700 and 2011CB302206), and the NSFC project (61272090, 61373024 and 61422205), YETP0105, Tencent, Huawei, SAP, the "NExT Research Center" (WBS: R-252-300-001-490), and the FDCT/106/2012/A3.

References

[1] 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.
[2] 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.
[3] G. Cong, C. S. Jensen, and D. Wu. Efficient retrieval of the top-k most relevant spatial web objects. *PVLDB*, 2009.
[4] I. D. Felipe, V. Hristidis, and N. Rishe. Keyword search on spatial databases. In *ICDE*, 2008.
[5] 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.
[6] H. Hu, D. L. Lee, and V. C. S. Lee. Distance indexing on road networks. In *VLDB*, pages 894–905, 2006.
[7] H. Hu, D. L. Lee, and J. Xu. Fast nearest neighbor search on road networks. In *EDBT*, pages 186–203, 2006.
[8] C. S. Jensen, J. Kolárvr, T. B. Pedersen, and I. Timko. Nearest neighbor queries in road networks. In *GIS*, pages 1–8, 2003.
[9] 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.
[10] 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.
[11] G. Karypis and V. Kumar. Analysis of multilevel graph partitioning. In *SC*, 1995.
[12] M. R. Kolahdouzan and C. Shahabi. Voronoi-based k nearest neighbor search for spatial network databases. In *VLDB*, 2004.
[13] 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.
[14] S. Luo, Y. Luo, S. Zhou, G. Cong, and J. Guan. Disks: A system for distributed spatial group keyword search on road networks. *PVLDB*, 5(12):1966–1969, 2012.
[15] K. Mouratidis, M. Yiu, D. Papadias, and N. Mamoulis. Continuous nearest neighbor monitoring in road networks. In *VLDB*, 2006.
[16] D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao. Query processing in spatial network databases. In *VLDB*, pages 802–813, 2003.
[17] M. Qiao, L. Qin, H. Cheng, J. X. Yu, and W. Tian. Top-k nearest keyword search on large graphs. *PVLDB*, 6(10):901–912, 2013.
[18] J. B. Rocha-Junior and K. Nørvåg. Top-k spatial keyword queries on road networks. In *EDBT*, pages 168–179, 2012.
[19] H. Samet, J. Sankaranarayanan, and H. Alborzi. Scalable network distance browsing in spatial databases. In *SIGMOD*, 2008.
[20] J. Sankaranarayanan, H. Alborzi, and H. Samet. Efficient query processing on spatial networks. In *GIS*, pages 200–209, 2005.
[21] J. Sankaranarayanan, H. Samet, and H. Alborzi. Path oracles for spatial networks. *PVLDB*, 2(1):1210–1221, 2009.
[22] H. Wang and R. Zimmermann. Snapshot location-based query processing on moving objects in road networks. In *GIS*, 2008.
[23] 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.
[24] R. Zhong, G. Li, K.-L. Tan, and L. Zhou. G-tree: an efficient index for knn search on road networks. In *CIKM*, pages 39–48, 2013.

Ruicheng Zhong is currently a PhD candidate in the Department of Computer Science, Tsinghua University, Beijing, China. His research interests mainly include spatio-textual data query, query processing on road network and spatial database.

Guoliang Li received his PhD degree from Tsinghua University, in 2009. He is currently working as an associate professor in the Department of Computer Science, Tsinghua University, Beijing. His research interests mainly include data cleaning and integration, spatial databases, and crowdsourcing.



Kian-Lee Tan received the BSc (Hons.) and PhD degrees from the National University of Singapore(NUS) in 1989 and 1994. He is currently a professor in the Department of Computer Science, NUS. His major research interests include query processing and optimization, database security, and database performance. He is a member of ACM.

Lizhu Zhou received the master of science degree from the University of Toronto, in 1983. Currently, he is a full professor at the Tsinghua University, China. His major research interests include database systems, web data processing, and digital resource management. He is a member of ACM.

Zhiguo Gong is currently an Associate Professor in the Department of Computer and Information Science, University of Macau, Macau, China. His research interests include databases, Web information retrieval, and Web mining.

1041-4347 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.