V-Tree: Efficient kNN Search on Moving Objects with Road-Network Constraints

Bilong Shen[1], Ying Zhao[1], Guoliang Li,[1] Weimin Zheng[1], Yue Qin[1], Bo Yuan[2], Yongming Rao[3]
Department of Computer Science and Technology[1], Department of Automation[2], Department of Electronic Engineering[3]
Tsinghua Uinversity, Beijing, China
shenbilong@gmail.com;{yingz, liguoliang, zwm-dcs}@tsinghua.edu.cn; {qinyuethu, tsinghuayuanb14, raoyongming95}@gmail.com

***Abstract*—Intelligent transportation systems, e.g., Uber, have become an important tool for urban transportation. An important problem is k nearest neighbor (kNN) search on moving objects with road-network constraints, which, given moving objects on the road networks and a query, finds k nearest objects to the query location. Existing studies focus on either kNN search on static objects or continuous kNN search with Euclidean-distance constraints. The former cannot support dynamic updates of moving objects while the latter cannot support road networks. Since the objects are dynamically moving on the road networks, there are two main challenges. The first is how to index the moving objects on road networks and the second is how to find the k nearest moving objects. To address these challenges, in this paper we proposes a new index, `V-Tree`, which has two salient features. Firstly, it is a balanced search tree and can support efficient kNN search. Secondly, it can support dynamical updates of moving objects. To build a `V-Tree`, we iteratively partition the road network into sub-networks and build a tree structure on top of the sub-networks. Then we associate the moving objects on their nearest vertices in the `V-Tree`. When the location of an object is updated, we only need to update the tree nodes on the path from the corresponding leaf node to the root. We design a novel kNN search algorithm using `V-Tree` by pruning large numbers of irrelevant vertices in the road network. Experimental results on real datasets show that our method significantly outperforms baseline approaches by 2-3 orders of magnitude.**

I. Introduction

Intelligent transportation systems, e.g., Uber, have been emerged as an important transportation tool. For drivers, Uber represents a flexible new way to earn money. For cities, Uber helps strengthen local economies, improves access to transportation, and makes streets safer. For passengers, Uber helps them easy to get a cab. Thus Uber has been widely used in our daily life.

An important problem in Uber is K nearest neighbor (kNN) search on moving objects on road networks, which finds the k nearest objects to a given query location. Existing studies focus on either kNN search on static objects [22], [16], [25], [10], [31], [30], [22] or continuous kNN search with Euclidean distance constraint [26], [11], [23], [27], [5], [9], [6], [17], [28], [29], [8]. The former cannot support dynamic updates of moving objects, because it is rather expensive to update the index. The latter cannot efficiently compute the distance on road networks. Thus they cannot efficiently address this problem. For example, Uber in China took more than 180 seconds to find the kNN results for each query.

Two factors make the problem more challenging. Firstly, the objects are dynamically moving on the road networks. For example, there are more than 60K taxies in Beijing and the locations of cars are updated every second. Thus one challenge is how to index the moving objects on road networks. Secondly, there are lots of queries. For example, in Beijing there are 1 million queries each day and in the peak time there are 100K queries in each second. Thus another challenge is how to find the kNN moving objects efficiently.

To address these challenges, in this paper we proposes a new index, `V-Tree`, which has two salient features. Firstly, it is a balanced search tree and can support efficient kNN search. Secondly, it can support dynamical updates of moving objects. To achieve this goal, we iteratively partition the road network into sub-networks and build a tree structure on top of the sub-networks, where the tree nodes are sub-networks. Then we associate the moving objects to their closet vertices on the road networks. To facilitate the kNN computation, we also keep the shortest distances from some important vertices (called borders) to the vertices with associated objects. When the location of an object is updated, we only need to update the tree nodes on the path from the corresponding leaf node to the root. We also design a novel kNN search algorithm using the borders to efficiently compute k nearest objects, which adopts a best-first method and can prune many irrelevant objects.

To summarize, we make the following contributions.

1) We devise an efficient and scalable tree index for moving objects on road network, called `V-Tree`. The space complexity of `V-Tree` is $\mathcal{O}(\log |V| \cdot |V|)$, where $|V|$ is the number of vertices in the road network.
2) We propose an efficient update strategy to support updates of moving objects. The average time complexity is $O(\frac{|V| \min(\log |\mathcal{M}|, \log |V|)}{|\mathcal{M}|})$, where $|\mathcal{M}|$ is the number of objects moving on the road network.
3) We devise a novel kNN search method using `V-Tree` to compute k nearest objects. The average time complexity is $O(\frac{k \cdot |V| \min(\log |\mathcal{M}|, \log |V|)}{|\mathcal{M}|})$.
4) We have conducted extensive experiments to evaluate our method on real datasets. Experimental results show that our method significantly outperforms baseline approaches by 2 orders of magnitude. We also publicize our source code at https://github.com/TsinghuaDatabaseGroup/VTree.

The structure of this paper is organized as follows. We first formulate the problem in Section II. The `V-Tree` is proposed in Section III, and we devise an efficient kNN search algorithm in Section IV. Experimental results are reported in Section V. We review related work in Section VI and conclude the paper in Section VII.

II. Preliminaries

Road Network. We model a road network as a directed weighted graph $G = \langle V, E \rangle$, where V is a set of vertices and E is a set of edges. Each edge $(u, v) \in E$ $(u, v \in V)$

has a weight, which is the travel cost from u to v (e.g., distance, travel time.). Given a path from u to v, the *distance* of this path is the sum of weights of the edges along the path. Let $\texttt{SPath}(u, v)$ denote the shortest path from u to v and $\texttt{SPDist}(u, v)$ denote the shortest-path distance. We use graph and road network interchangeably for ease of presentation.

Moving Objects. Each object (e.g., vehicle) moving on the road network is represented by $m = \langle t, p \rangle$, where p is the geo-location of m at time t. Note the locations of objects are updated periodically (e.g., every second).

We suppose moving object is driving on the road. We can utilize existing techniques [4], [24], [19] to map an object to an edge on the road. Suppose m is driving on edge $e = (u, v)$ and its distance to v is $\delta = \texttt{SPDist}(m, v)$. We use $m = (t, (u, v), \delta)$ to denote the object. The shortest-path distance from a moving object m to a vertex x, denoted by $\texttt{SPDist}(m, x)$, can be computed by $\delta + \texttt{SPDist}(v, x)$. In Figure 1, the distance from m_1 to v_7 is computed by summing up the δ from m_1 to v_5, and $\texttt{SPDist}(v_5, v_7)$, i.e., $\texttt{SPDist}(m_1, v_7) = 80 + 180 = 260$.

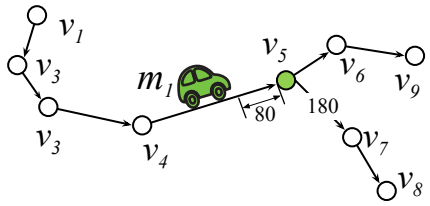


Fig. 1. Example of Objects.

kNN Query. Given a graph DAG, a moving object set $\mathcal{M}$, a kNN query $q = \langle v, k \rangle$, where v is a query location, and k is an integer. The answer of q 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., $\mathcal{R} \subseteq \mathcal{M}$.
(3) $\forall x \in \mathcal{R}, \forall y \in \mathcal{M} - \mathcal{R}, \texttt{SPDist}(v, x) \leq \texttt{SPDist}(v, y)$.

Similar to existing works [4], [24], [19], we assume that the query location of q is at a vertex. If the query location is not at a vertex, then (i) if it is on an edge, we find the top-k answers to the two vertices of edge e, and then select the top-k answers from these $2k$ candidates; (ii) if it is not on an edge, we find the closest edge e to the query location using existing technique [4], [24], [19] and then utilize the method in case (i) to compute the top-k answers. Thus in the paper, we focus on the case that the query location is at a vertex.

III. The **V-Tree** Index

We propose a tree index to support kNN search on moving objects, called V-Tree. We first formally define V-Tree (Section III-A) and then discuss how to construct V-Tree (Section III-B). Next we present utilizing V-Tree to compute shortest-path distance (Section III-C). We close this section by discussing how to update V-Tree (Section III-D).

A. V-Tree

Before we introduce the V-Tree structure, we first define some concepts.

Definition 1: (Graph Partition). Given a graph $G = \langle V, E \rangle$, where V is the vertex set and E is the edge set of G, f is the fanout, we partition G into f subgraphs, i.e., $G_1 = \langle V_1, E_1 \rangle, G_2 = \langle V_2, E_2 \rangle, ..., G_f = \langle V_f, E_f \rangle$, such that
(1) Completeness on vertices: $\cup_{1 \leq i \leq f} V_i = V$;
(2) Disjoint on vertices: For $i \neq j, V_i \cap V_j = \phi$; and
(3) Completeness on edges for vertices in the same subgraph: $\forall u, v \in V_i$, if $(u, v) \in E$, $(u, v) \in E_i$.

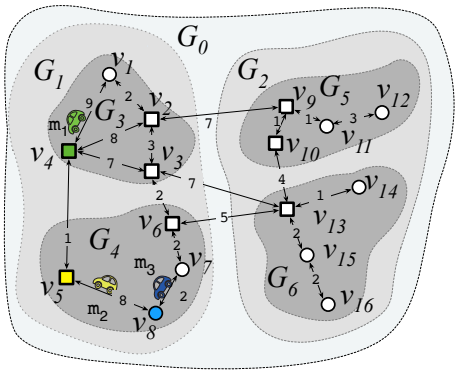


Fig. 2. Road Network Partition.

We build a tree hierarchy based on the graph partition. Initially the root is the graph G. Then we iteratively partition the graph G as follows. We partition G into f subgraphs G_1, G_2,..., G_f, which are taken as G's children. These f subgraphs are called *child subgraphs* of G. The *child-subgraph* set of G is denoted by $\mathcal{C}(G)$. G is called the *parent graph* of its *child subgraphs*. The *parent graph* of G_i is denoted by G_i^p. Next we iteratively partition G_i, and take its child subgraphs as its children. We terminate if G_i has less than τ vertices, where τ is a threshold. Figure 2 shows an example of this iterative graph partition process, with an original road network graph $G_0 = G$, $f = 2$ and $\tau = 4$. G_0 is partitioned into two child subgraphs G_1 and G_2, so $\mathcal{C}(G_0) = (G_1, G_2)$ and $G_1^p = G_0$.

Definition 2 (Boundary Vertex): Given a graph $G = \langle V, E \rangle$, its subgraph G_i, a vertex β_j in G_i is called a boundary vertex if $\exists (\beta_j, v) \in E$ and v is not in G_i.The boundary vertex set of G_i is denoted by $B(G_i)$.

Given two vertices u and v in G, if $v \in G_i$ and $u \notin G_i$, then the shortest path from u to v must bypass boundary vertices of G_i, because if v connects to u, it must go out G_i (i.e., bypass a boundary vertex) as stated in Lemma 1 .

Lemma 1: Given a subgraph $G_i = \langle V_i, E_i \rangle$ and two vertices $v_i \in V_i$, $v_j \notin V_i$, the shortest path from v_j to v_i must contain a boundary vertex β in G_i such that $\texttt{SPDist}(v_j, v_i) = \texttt{SPDist}(v_j, \beta) + \texttt{SPDist}(\beta, v_i)$.

Proof: The proofs of Lemmas and Theorems can be found in our technical report [1]. ■

Accordingly, given two subgraphs G_i, G_j and two vertices $v_i \in G_i$ and $v_j \in G_j$, there must exist two boundary vertices $\beta_i \in G_i$ and $\beta_j \in G_j$ such that

$$\texttt{SPDist}(v_j, v_i) = \texttt{SPDist}(v_j, \beta_j) + \texttt{SPDist}(\beta_j, \beta_i) + \texttt{SPDist}(\beta_i, v_i).$$

To efficiently compute the shortest-path distance, we can precompute the distances between vertices and boundary vertices, and between boundary vertices and boundary vertices. However this involves huge storage space. To address this issue, we only precompute the shortest-path distances between vertices for leaf nodes, and the shortest-path distances between boundary vertices and boundary vertices for non-leaf nodes with the same parents. We will show that the shortest path between two vertices can be efficiently computed based on these precomputed distances in Section III-C.

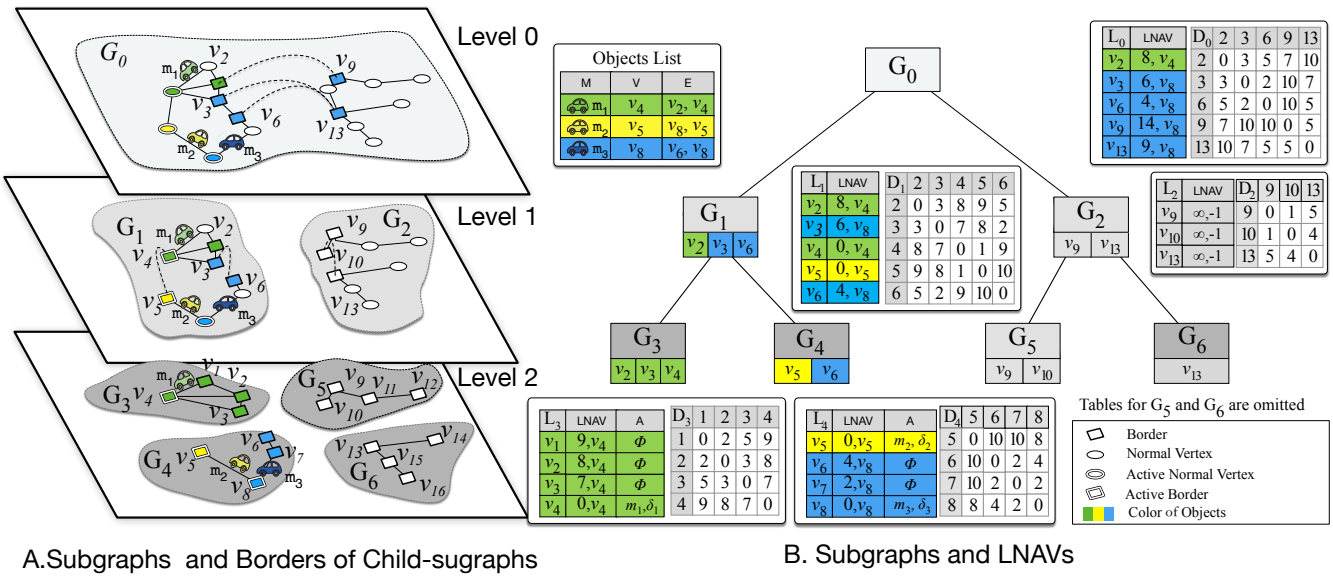


Fig. 3. V-Tree Structure (Square Vertices are Borders of Child-subgraphs in Different Levels).

Distance Matrix. Each leaf subgraph maintains a distance matrix $\mathcal{D}$ of all vertices in the subgraph, i.e., $\mathcal{D}_{i,j} = \texttt{SPDist}(v_i, v_j)$ for all v_i and v_j in the subgraph. Each non-leaf subgraph maintains a distance matrix $\mathcal{D}$ of all boundary vertices of its child subgraphs. i.e., $\mathcal{D}_{i,j} = \texttt{SPDist}(\beta_i, \beta_j)$, where β_i, β_j are boundary vertices of its child subgraphs. For ease of presentation, let $\mathbb{B}(G_i)$ denote the set of boundary vertices of G_i's child subgraphs, and we call each boundary vertex in $\mathbb{B}(G_i)$ is a *border*.

Definition 3 (Border): Given a non-leaf subgraph G_i, a vertex β_i in G_i is called a border if it is a boundary vertex of one of its child subgraph. Given a leaf subgraph, each of its boundary vertex is a border.

As shown in Figure 2, $B(G_3) = \{v_2, v_3, v_4\}$, $B(G_4) = \{v_5, v_6\}$, $\mathbb{B}(G_1) = \{v_2, v_3, v_4, v_5, v_6\}$. The shortest path from v_1 to v_8 bypasses borders v_3 of G_1 and v_6 of G_4.

Next, we discuss how to associate the moving objects into the vertices.

Definition 4 (Active and Inactive Vertex): Given an object m on an edge e which is moving to the vertex v, we call m an *active* object of vertex v and call v an active vertex. A vertex is *active* if it has active objects; *inactive* otherwise.

For example, in Figure 2, m_1 is on edge $e(v_1, v_4)$ and moving to v_4 with an offset of 0. m_1 is an active object of v_4 and v_4 is an active vertex. Vertex v_1 is inactive as it has no active object. The distance of the shortest path from an object m to any vertex $u \in G$, $\texttt{SPDist}(m, u)$, is the summation of the offset δ to vertex v that m is driving to and $\texttt{SPDist}(v, u)$, i.e., $\texttt{SPDist}(m, u) = \delta + \texttt{SPDist}(v, u)$.

*Definition 5 (Local Nearest Active Vertex-*LNAV*):* Given a vertex u in a subgraph G_i, an active vertex v in G_i is called the local nearest active vertex (LNAV) if v has the minimum distance to u among all the active vertices in G_i.

For example, in Figure 2, v_4, v_5, and v_8 are active vertices. The LNAV of v_3 in G_3 is v_4; in G_1, the LNAV of v_3 is v_8.

Note that the global nearest active vertex (GNAV) of u may be not in G_i. We use LNAV instead of GNAV because it is rather expensive to compute and update GNAV while LNAV is easy to maintain and update. More importantly, we can efficiently compute GNAV based on LNAV which will be discussed in Section IV.

LNAV **Table** $\mathcal{L}$**.** For each non-leaf subgraph G_i, we store the LNAV for all the borders in the subgraph in a NAV table, denoted by $\mathcal{L}_i$. The LNAV table has two columns: the LNAV and the distance to LNAV. For each border β in subgraph G_i, $\mathcal{L}_i[\beta].\gamma$ and $\mathcal{L}_i[\beta].\delta$ keep the NAV of β in G_i and the distance to β, respectively.

For example, in Figure 2, v_3 is a boundary vertex of G_3, the LNAV to v_3 in G_1 is v_8, so $\mathcal{L}_1[v_3].\delta = 6$ and $\mathcal{L}_1[v_3].\gamma = v_8$. We will discuss later that the LNAV table in each subgraph can be updated efficiently in Section III-D.

Active Object Table $\mathcal{A}$**.** For each leaf subgraph, we maintain an active object table $\mathcal{A}$. For each vertex v in the leaf subgraph G_i, we use $\mathcal{A}_i[v]$ to keep its active objects, where each entry is $\langle m, \delta \rangle$ to represent an active object and its offset to its active vertex.

Based on the above notations, next we are ready to define the V-Tree.

*Definition 6 (*V-Tree*):* A V-Tree of a road network G is a balanced search tree that has the graph partition hierarchy and satisfies the following properties.

(1) Each node in V-Tree corresponds to a subgraph. Each non-leaf node has f children. Each leaf node has less than τ vertices.

(2) Each node also maintains a distance matrix $\mathcal{D}$. Each leaf node maintains the distance matrix for its vertices while each non-leaf node maintains the distance matrix for its borders.

(3) Each node maintains a LNAV table $\mathcal{L}$. The leaf nodes maintain the LNAV for all vertices in the leaf nodes while the non-leaf nodes maintain the LNAV for its borders.

(4) Each leaf node maintains an active object table $\mathcal{A}$, which maintains the active objects for each vertex in the leaf node.

For example, Figure 3 shows the V-Tree of the road network and moving objects in Figure 2 with $f = 2$ and $\tau = 4$. Each non-leaf node stores a LNAV table (on the right or below the node). The rows in green are borders taking v_4 as the LNAV,

TABLE I. A LIST OF NOTATIONS USED IN THE PAPER

Notation	Definition
$G = \langle V, E \rangle$	Graph G with vertex set V and edge set E
$\mathcal{M}$	Set of Moving Objects
$B(G_i)$	Boundary vertex set of G_i
$\mathbb{B}(G_i)$	*Border* set of G_i
$\texttt{SPDist}(a, b)$	Shortest path distance from a to b
$\mathcal{D}$	Distance matrix
$\mathcal{D}_{i,j}$	$\texttt{SPDist}(\beta_i, \beta_j)$
$\mathcal{A}$	Active object table
LNAV	Local nearest active vertex
$\mathcal{L}_i$	LNAV table of G_i
$\mathcal{L}_i[\beta].\gamma$	NAV of β in G_i
$\mathcal{L}_i[\beta].\delta$	The distance from $\mathcal{L}_i[\beta].\gamma$ to β
G_v^p	Parent graph of G_v
f	Fanout of V-Tree
τ	Maximum number of vertices in a leaf node

to which m_1 is driving to. Similarly, the rows in yellow have v_5 as LNAV.

Space Complexity of V-Tree. Given a graph G with $|V|$ vertices and $|\mathcal{M}|$ objects, the space complexity of V-Tree is $\mathcal{O}(|\mathcal{M}| + \log|V| \cdot |V|)$.

Table 1 summarizes a list of essential notations used in this paper. (Some notations will be introduced later.)

B. V-Tree *Construction*

Tree Hierarchy. It aims not only to partition the graph to equal-size subgraphs, but more important to ensure the subgraphs have small size of borders in each level. Based on the planar separator theorem [18], given a planar graph with $|V|$ vertices, if we partition it into f subgraphs, the number of boundary vertices of the subgraphs is $\mathcal{O}(\sqrt{|V|})$. We divide the full graph into f equal-sized subgraphs by a famous multilevel algorithm [15], which can make each subgraph have almost the same size and a small number of borders. Specifically, we first partition the full-graph to f equal-sized subgraphs. And for each subgraph, we divide it to f equal-sized subgraphs. The partition terminates until the number of vertices contained in the subgraph is less than or equal to τ.

Distance Matrix $\mathcal{D}$. It computes the shortest-path distances between all the borders for each non-leaf subgraphs and the shortest-path distances between all the vertices of the leaf subgraphs. The naive method is to compute the distance of vertices pair by pair. The construction complexity of this method is too high to scale to large road networks. To address this issue, we use the bottom-up method to compute the matrix by sharing the computations [31], [30], and the time complexity is $\mathcal{O}(|V|^{1.5})$.

Active Object Table $\mathcal{A}$. For a new object m, it adds m into corresponding subgraph's active object table. The complexity is $\mathcal{O}(1)$.

LNAV **Table $\mathcal{L}$.** For a new object, we discuss how to add it into the LNAV table later.

For a V-Tree which does not contain any object, each LNAV entry in V-Tree is set to $\langle \infty, \phi \rangle$, and the active object of each vertex is ϕ. The $\mathcal{A}$ and $\mathcal{L}$ tables will be updated when objects are added.

C. Implementing SPDist Function on V-Tree

Given two vertices u and v on the road network, SPDist (u, v) computes the shortest-path distance from u to v. We consider the following two cases.

Algorithm 1: $\texttt{Add}(m, v)$

```
Input: m, v                                   // adding m to vertex v
if v.status = inactive then
    L_{G_v}[v] = ⟨0, v⟩;
    for each vertex u in G_v do
        if L_{G_v}[u].δ > SPDist(v, u) then
            L_{G_v}[u] ← ⟨SPDist(v, u), v⟩;
    propagation ← true;
    while G_v ≠ G_0 and propagation do
        propagation ← false ;
        for each border β in 𝔹(G_v^p) ∩ 𝔹(G_v) do
            if L_{G_v}[β].δ < L_{G_v^p}[β].δ then
                L_{G_v^p}[β] ← L_{G_v}[β];
                propagation ← true;
        for each border β' in 𝔹(G_v^p) − 𝔹(G_v) do
            d ← min_{β ∈ 𝔹(G_v), L_{G_v^p}[β].γ = v} L_{G_v^p}[β].δ + SPDist(β, β');
            if L_{G_v^p}[β'].δ > d then
                L_{G_v^p}[β'] ← ⟨d, v⟩;
        G_v ← parent of G_v;
    v.status ← active;
add m to A[v] ;
```

u, v **in the same leaf subgraph**. SPDist (u, v) is directly gotten from the leaf distance matrix $\mathcal{D}[u][v]$. The time complexity of this case is $\mathcal{O}(1)$.

u, v **in different leaf subgraphs**. We utilize a dynamic-programming algorithm [31] to compute SPDist (u, v). The basic idea is as follows. We first location the leaf subgraphs G_v and G_u of v and u respectively, which can be implemented by a hash table. Then we compute the least common ancestor LCA of G_v and G_u, and the nodes on the paths from LCA to G_v and G_u. Next we enumerate the combinations of borders in these nodes to compute the shortest-path distance. We can utilize the dynamic-programming algorithm [31] to share the computations.

D. Updates on V-Tree

As the road network will not change frequently, we focus on how to update the locations of active objects on V-Tree.

There are four cases for updating an active object m.

Case 1. Adding a new object m on edge (u, v) and driving to v. For example, an object, e.g., a taxi, becomes free from busy. We use algorithm $\texttt{Add}(m, v)$ to add m to v.

Case 2. Deleting an object m from edge (u, v). For example, an object, e.g., a taxi, becomes busy from free. We use algorithm $\texttt{Del}(m, v)$ to delete m from v.

Case 3. Object m is still on edge (u, v) and driving to v, but the offset from m to v is changed. Vertex v is still an active vertex of m. We only need to update the offset of m in $\mathcal{A}[v]$. The update complexity of this case is $\mathcal{O}$ (1).

Case 4. Object m is moving to edge (v, w) from edge (u, v). m is not an active object of v and becomes an active object of w. We require to delete m from v and add m to w. Thus we call the functions $\texttt{Del}(m, v)$ and $\texttt{Add}(m, w)$.

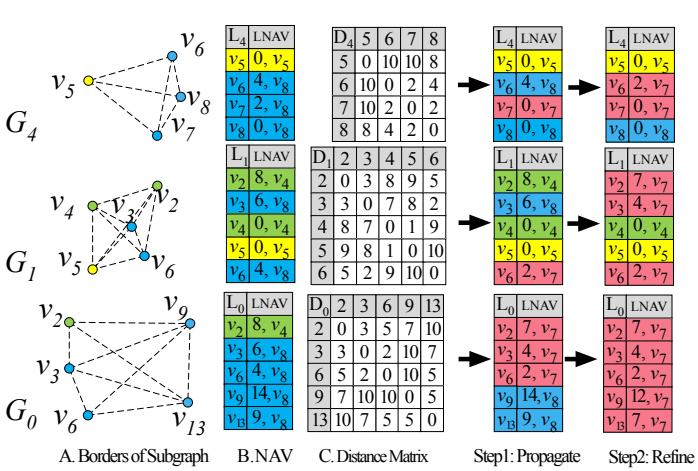


Fig. 4. Adding m_4 to v_7

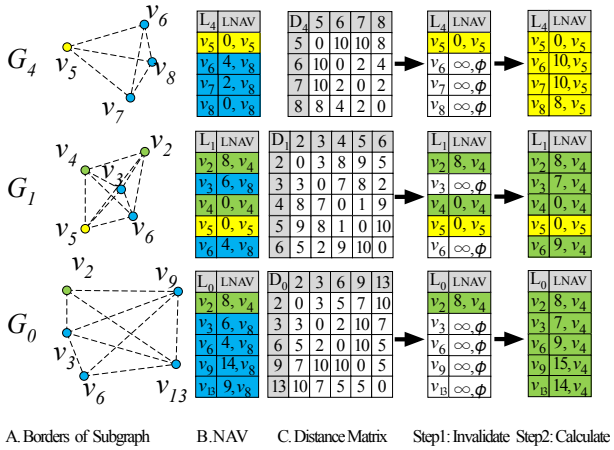


Fig. 5. Deleting m_3 from v_8

Thus we only need to consider two functions $\texttt{Del}(m, v)$ and $\texttt{Add}(m, v)$. Next we discuss how to implement these two functions.

1) Adding m to Vertex v, $\texttt{Add}(m, v)$: There are two cases for v. (i) Before adding m to v, v is already an active vertex, which already contains at least one object. The status of v will not change. Thus all LNAV entries on V-Tree will not change. We just add m into $\mathcal{A}[v]$. (ii) Before adding m to v, v is an inactive vertex. The state of v will change from inactive to active. We need to add m into $\mathcal{A}[v]$. As v may become the LNAV for some borders in the subgraphs containing v, we need to update the LNAV entries for such borders. Note v will not become the LNAV for borders in the subgraphs that do not contain v. In this way, we only need to check the leaf node containing v, G_v, and its ancestors. To this end, we propose a bottom-up method to update the LNAV and the pseudo code is shown in Algorithm 1.

Updating Leaf Graph G_v. Consider a vertex u in G_v whose LNAV is not v, i.e., $\mathcal{L}_{G_v}[u].\gamma = w$ and $w \neq v$. If $\texttt{SPDist}(v, u) < \texttt{SPDist}(w, u)$, v should be the LNAV of u and we should update $\mathcal{L}[u]_{G_v}.\gamma$ to v (lines 5-6 in Algorithm 1). Note that both $\texttt{SPDist}(v, u)$ and $\texttt{SPDist}(w, u)$ can be found from the pre-calculated distance matrix $\mathcal{D}$ in G_v.

Updating The Ancestors of G_v. If the LNAV of a border β in G_v is changed and v becomes the LNAV in G_v, i.e., $\mathcal{L}_{G_v}[\beta].\gamma = v$, v also becomes the LNAV of β in G_v's parent G_v^p, and thus we need to update the LNAV for its parent. The update may also influence other borders that are not in G_v, because it may shorten the distance from v to such border. Next we propose a three-step method to update G_v^p.

Step 1. For each border β in $\mathbb{B}(G_v^p) \cap \mathbb{B}(G_v)$, if $\mathcal{L}_{G_v}[\beta].\delta < \mathcal{L}_{G_v^p}[\beta].\delta$, we update $\mathcal{L}_{G_v^p}[\beta]$ as $\mathcal{L}_{G_v}[\beta]$.

Step 2. For each border β' in $\mathbb{B}(G_v^p) - \mathbb{B}(G_v)$, if $\exists \beta \in \mathbb{B}(G_v^p) \cap \mathbb{B}(G_v)$, $\mathcal{L}_{G_v^p}[\beta].\delta + \texttt{SPDist}(\delta, \delta') < \mathcal{L}_{G_v^p}[\beta'].\delta$, we update $\mathcal{L}_{G_v^p}[\beta']$ as $\langle \mathcal{L}_{G_v^p}[\beta].\delta + \texttt{SPDist}(\delta, \delta'), \mathcal{L}_{G_v^p}[\beta].\gamma \rangle$.

Step 3. If the LNAV of some borders in G_v^p are updated, we require to update the parent of G_v^p; otherwise the algorithm terminates.

Iteratively, we perform these steps on the parent node of G_v^p until reaching the root node.

For example, in Figure 4, we show how to add object m_4 to vertex v_7 from the V-Tree shown in Figure 3. The LNAV tables and distance matrices before adding m_4 are shown in Figure 4 (B) and (C), respectively. We first change the status of v_7 to *active* and check whether v_7 becomes the LNAV of other vertices in G_4. $\mathcal{L}_{G_4}[v_6]$ is updated to $\langle 2, v_7 \rangle$. Next, because v_6 is a border in G_4, $\mathcal{L}_{G_1}[v_6]$ is updated to be $\langle 2, v_7 \rangle$ by lines 9-12 in Algorithm 1. The LNAV of v_2 and v_3 are refined by lines 13-16 in Algorithm 1. Iteratively, the $\mathcal{L}$ table of G_0 (the parent of G_1) needs to be updated too. $\mathcal{L}_{G_0}[v_2]$, $\mathcal{L}_{G_0}[v_3]$, and $\mathcal{L}_{G_0}[v_6]$ are directly propagated from G_1. $\mathcal{L}_{G_0}[v_9]$ and $\mathcal{L}_{G_0}[v_{13}]$ are refined accordingly. Since G_0 is the root, the process terminates.

2) Removing m from Vertex v, $\texttt{Del}(m, v)$: There are two cases for v. (i) After removing m from v, v is still an active vertex. The status of v will not change. Thus the LNAV will not change. We only need to remove m from $\mathcal{A}[v]$. (ii) After removing m from v, v become an inactive vertex. We remove m from $\mathcal{A}(v)$ and recalculate the LNAV where v was the LNAV for some borders in the subgraphs containing v.

Similar to Add (m,v), next we propose a bottom-up method to update LNAV, and the pseudo code is shown in Algorithm 2.

Updating Leaf Graph G_v. Consider a vertex u in G_v with $\mathcal{L}_{G_v}[u].\gamma = v$. We set $\mathcal{L}_{G_v}[u]$ as $\langle \infty, \phi \rangle$. We find the active vertex w with the minimum distance to u, and update $\mathcal{L}_{G_v}[u]$ as $\langle \texttt{SPDist}(w, u), w \rangle$.

Updating The Ancestor of G_v. Suppose G_v^p is the parent node of G_v. Consider a border β in G_v^p with $\mathcal{L}_{G_v^p}[\beta].\gamma = v$. We need to update it. We also adopt a 3 step method.

Step 1. For each border with $\mathcal{L}_{G_v^p}[\beta].\gamma = v$, we first set $\mathcal{L}_{G_v^p}[\beta] = \langle \infty, \phi \rangle$.

Step 2. We recalculate $\mathcal{L}_{G_v^p}[\beta]$ based on borders in G_v^p. For each border β' in G_v^p, say β' is the boundary vertex of $G_j, G_j \in \mathcal{C}(G_v^p)$, if $\mathcal{L}_{G_j}[\beta'].\delta + \texttt{SPDist}(\beta', \beta) < \mathcal{L}_{G_v^p}[\beta]$, we update $\mathcal{L}_{G_v^p}[\beta]$ by $\langle \mathcal{L}_{G_j}[\beta'].\delta + \texttt{SPDist}(\beta', \beta), \mathcal{L}_{G_j}[\beta'].\gamma \rangle$.

Step 3. If the LNAV of some borders in G_v^p are updated, we require to update the parent of G_v^p; otherwise the algorithm terminates.

Iteratively, we perform these steps on the parent node of G_v until reaching the root node.

In Figure 5, we show an example of deleting m_3 from v_8 on the V-Tree of Figure 2. The leaf subgraph containing v_8 is G_4. We update LNAV of v_6, v_7, and v_8 with $\langle \infty, \phi \rangle$. And then the LNAV of v_6, v_7, and v_8 are updated to $\langle 10, v_5 \rangle$,$\langle 10, v_5 \rangle$,and $\langle 8, v_5 \rangle$ by the active vertex v_5, respectively. We then move on to the parent graph of G_4, which is G_1. We first reset the LNAV of v_3 and v_6 to $\langle \infty, \phi \rangle$. Then we find v_4 as the LNAV

Algorithm 2: $\texttt{Del}(m, v)$

Input: m, v // deleting m from vertex v

1 **if** *v only has one active vehicle* **then**
2 $v.status \leftarrow$ inactive ;
3 **for** *each vertex u in G_v s.t. $\mathcal{L}_{G_v}[u].\gamma = v$* **do**
4 $\mathcal{L}_{G_v}[u] \leftarrow \langle \infty, \phi \rangle$;
5 **for** *each active vertex $w \in G_v$* **do**
6 **if** $\mathcal{L}_{G_v}[u].\delta > \texttt{SPDist}(w, u)$ **then**
7 $\mathcal{L}_{G_v}[u] \leftarrow \langle \texttt{SPDist}(w, u), w \rangle$;
8 $\texttt{propagation} \leftarrow$ true ;
9 **while** *$G_v \neq G_0$ and* $\texttt{propagation}$ **do**
10 $\texttt{propagation} \leftarrow$ false ;
11 **for** *each border β s.t. $\mathcal{L}^p_{G^p_v}[\beta].\gamma = v$* **do**
12 $\mathcal{L}_{G^p_v}[\beta] \leftarrow \langle \infty, \phi \rangle$;
13 $\texttt{propagation} \leftarrow$ true ;
14 $d \leftarrow \min_{\substack{\beta' \in B(G_j) \\ G_j \in \mathcal{C}(G^p_v)}} \texttt{SPDist}(\beta', \beta) + \mathcal{L}_{G_j}[\beta'].\delta$;
15 $\mathcal{L}_{G^p_v}[\beta] \leftarrow \langle d, \mathcal{L}_{G_j}[\beta'].\gamma \rangle$;
16 $G_v \leftarrow$ parent of G_v ;
17 remove m from $\mathcal{A}[v]$;

Algorithm 3: $\texttt{knn}(v, k)$

Input: v: query location; k: the number of nearest neighbors
Output: $\mathcal{R}$: k nearest objects to v

1 Priority queue $\mathcal{R} \leftarrow \Phi$;
2 ε=maximal distance of candidates in $\mathcal{R}$ to v(initialized as ∞);
3 $u = \texttt{gnav}(v)$;
4 **for** *each active object $\langle m, \delta \rangle \in \mathcal{A}[u]$* **do**
5 **if** $\texttt{SPDist}(u, v) + \delta < \varepsilon$ **then**
6 $\mathcal{R}.\texttt{Enqueue}(\langle m, \texttt{SPDist}(u, v) + \delta \rangle)$;
7 Update ε;
8 **while** *true* **do**
9 $u = \texttt{nnav}(v, u)$;
10 **if** $\texttt{SPDist}(u, v) \geq \varepsilon$ **then**
11 **break;**
12 **for** *each active object $\langle m, \delta \rangle \in \mathcal{A}[u]$* **do**
13 **if** $\texttt{SPDist}(u, v) + \delta < \varepsilon$ **then**
14 $\mathcal{R}.\texttt{Enqueue}(\langle m, \texttt{SPDist}(u, v) + \delta \rangle)$;
15 Update ε;
16 return $\mathcal{R}$;

for v_3 in G_1 through $\mathcal{L}_{G_4}[v_3]\delta = 7$, and v_4 for v_6 through $\mathcal{L}_{G_3}[v_4].\delta + \texttt{SPDist}(v_4, v_6) = 9$. Iteratively, the LNAV of G_0 (the parent of G_1) needs to be updated next.

Correctness of Updating Algorithms.

Theorem 1: The two updating algorithms, Add and Del, correctly maintain the LNAV tables on V-Tree.

Time Complexity of Updating Algorithms.

Lemma 2: If updating a moving object does not change the state of vertex, the complexity of an update is $\mathcal{O}(1)$. If the updating operation changes the state, the average complexity of an update on V-Tree is

$$\mathcal{O}(\frac{|V| \log \min(|V|, |\mathcal{M}|)}{|\mathcal{M}|}).$$

where $|V|$ is the number of vertices and $|\mathcal{M}|$ is the number of objects.

IV. kNN Algorithm

A. Overview of kNN Algorithm

Given a query $q = \langle v, k \rangle$, the kNN query returns top-k active objects ranked by shortest-path distance to v. For any active object m, we know that $\texttt{SPDist}(m, v) = \texttt{SPDist}(u, v) + \delta(m, u)$, where u is the active vertex of m and $\delta(m, u)$ is the distance from m to u. Note that $\delta(m, u)$ is usually much smaller than $\texttt{SPDist}(u, v)$. Based on this observation, we first find the global NAV of v, denoted by u, and take the active objects of u as the top-k candidates of v. Then we mark u as inactive, and find the next global NAV of v, u'. If $\texttt{SPDist}(u', v)$ is larger than the distance of the k-the candidate, denoted by ε (i.e., $\texttt{SPDist}(u', v) \geq \varepsilon$) we can safely terminate. Next we formally introduce the algorithm.

Suppose we have two functions $\texttt{gnav}(v)$ and $\texttt{nnav}(v, u)$ to find the global NAV and the next global NAV of v, respectively. The discovered top-k moving objects are stored in a priority queue $\mathcal{R}$ of size k, i.e., it only keeps k objects with the k shortest SPDist to v.

Then $\texttt{knn}(v, k)$ works in three steps. (i) It maintains a priority queue with k candidate objects to v. Let ε denote the maximal distances from the candidates to v in the priority queue, which is initialized as ∞ (lines 1-2). (ii) It finds the global NAV u of v and $d = \texttt{SPDist}(u, v)$ by calling $\texttt{gnav}(v)$. It adds the active objects into the priority queue (lines 3-7 in Algorithm 3). (iii) It marks u inactive and finds the next global NAV u of v and $d = \texttt{SPDist}(u, v)$ by calling $\texttt{nnav}(v, u)$. If $d > \varepsilon$, which means the active objects of u cannot be kNN result of v, and the algorithm terminates (lines 10-11); otherwise, it adds the active objects of u into the priority queue (lines 12-15). Iteratively, the algorithm finds the top-k results.

B. Function $\texttt{gnav}(v)$

There are two cases for the global NAV of v. (i) v and its global NAV are in the same leaf subgraph. We can find the global NAV by exploring all the active vertices in the leaf subgraph and find the nearest one. (ii) v and its global NAV are in two different leaf subgraphs, in which case the shortest path from the global NAV to v must pass-by a boundary vertex of the leaf subgraph or its ancestor subgraphs containing v by Lemma 1. Therefore, we only need to explore the boundary vertices of these subgraphs and find the one connecting the global NAV. In other words, the global NAV of v must be in the local NAV in the nodes from G_v to the root.

Next we discuss how to compute the global NAV from the local NAV. Suppose G_v is the leaf node of v. Consider an ancestor of G_v, G^a_v. For any boundary vertex β in G_v, let

$$\texttt{LNAVDist}_{G^a_v}(\beta, v) = \texttt{SPDist}(\beta, v) + \mathcal{L}[\beta]_{G^a_v}.\delta$$

denote the local NAV distance from $w = \mathcal{L}_{G^a_v}[\beta].\gamma$ to v in G^a_v. The vertex w with the minimal local NAV distance is the global NAV as stated in Lemma 3.

Lemma 3: Given a vertex v, suppose G_v is its leaf node and let

$$\beta = \arg \min_{u \in \mathbb{B}(G^a_v), u \in B(G_v)} \texttt{LNAVDist}_{G^a_v}(u, v)$$

where G^a_v is an ancestor node of G_v. $w = \mathcal{L}_{G^a_v}[\beta].\gamma$ is the global NAV of v.

Function gnav(v)

Input: v: query location;
Output: the global NAV of v

1 $u = \arg\min_{\beta \in \mathbb{B}(G_v)} \texttt{LNAVDist}_{G_v}(\beta, v)$;
2 $\varepsilon \leftarrow \texttt{LNAVDist}_{G_v}(u, v)$;
3 **while** $G_v^p \neq$ NULL *and* $\texttt{SPDist}(G_v^p, v) < \varepsilon$ **do**
4 $\quad u' = \arg\min_{u' \in \mathbb{B}(G_v^p), u' \in B(G_v)} \texttt{LNAVDist}_{G_v^p}(u', v)$;
5 $\quad$ **if** $\texttt{LNAVDist}_{G_v^p}(u', v) < \varepsilon$ **then**
6 $\quad\quad u = u'$;
7 $\quad\quad \varepsilon \leftarrow \texttt{LNAVDist}_{G_v^p}(u, v)$;
8 $\quad G_v^p \leftarrow$ parent of G_v^p ;
9 return $\mathcal{L}_{G_v^p}[u].\gamma$;

Thus a simple method is to enumerate every local NAV in the nodes from G_v to the root. However this method will enumerate every ancestors of G_v, and next we propose an efficient method which can skip many unnecessary nodes. To achieve this goal, we have two observations.

Observation 1. Monotone Non-Decreasing. Given an ancestor of G_v, G_v^a, let

$$\texttt{SPDist}(G_v^a, v) = \min_{\beta \in \mathbb{B}(G_v^a)} \texttt{SPDist}(\beta, v).$$

We have $\texttt{SPDist}(G_v, v) \leq \texttt{SPDist}(G_v^a, v)$ as stated in Lemma 4.

Lemma 4: Given two nodes G_v and G_v^a, where G_v is the leaf node of v and G_v^a is an ancestor of v, we have

$$\texttt{SPDist}(G_v, v) \leq \texttt{SPDist}(G_v^a, v).$$

Observation 2. Early Termination. Let

$$\texttt{LNAVDist}(G_v^a, v) = \min_{\beta \in \mathbb{B}(G_v^a)} \texttt{LNAVDist}_{G_v^a}(\beta, v).$$

Consider two ancestors of G_v, G_v^p and G_v^a, where G_v^a is the parent of G_v^p.If the LNAV distance of G_v^p ($\texttt{LNAVDist}(G_v^p, v)$) is not larger than $\texttt{SPDist}(G_v^a, v)$, i.e.,

$$\texttt{LNAVDist}(G_v^p, v) \leq \texttt{SPDist}(G_v^a, v)$$

we can skip G_v^a and its ancestors, because the vertices in them have larger distance based on the monotone non-decreasing property and Lemma 3.

Based on these two observations, we propose a bottom-up method to explore the borders of subgraphs containing v, which is shown in Function gnav.

Exploring The Vertices in Leaf Graph G_v. Suppose G_v is the leaf subgraph containing v. Consider an active vertex u in G_v, u is also the NAV of itself in G_v, so we compute the vertex u in the leaf with the minimal distance (line 2 in Function gnav).

Exploring The Borders in Ancestors of G_v. Let G_v^p be the parent subgraph of G_v. ε holds the minimum of $\texttt{LNAVDist}_{G_v}(u, v)$. If $\texttt{SPDist}(G_v^p, v) \geq \varepsilon$; the algorithm terminates. On the contrary, if $\texttt{SPDist}(G_v^p, v) < \varepsilon$, we find GNAV in $G_v^p$and compute the vertex with the minimal LNAV in G_v^p. If there is a vertex in G_v^p with smaller distance, we select the vertex with the minimal distance (lines 4-7).

For example, in Figure 6 we illustrate how to search the global NAV of v_7 on the V-Tree shown in Figure 3. $G_4 =$ $\texttt{Leaf}(v_7)$ and v_8 is the NAV to v_7 in G_4 with $\texttt{SPDist}(v_8, v_7)$=2. The parent graph of G_4 is G_1. Since $\texttt{SPDist}(G_1, v_7) = 2$, any active vertex that passes borders in G_1 and its ancestor graphs to v_7 must have a distance greater or equal to 2. The algorithm terminates and returns v_8 as the global NAV for v_7.

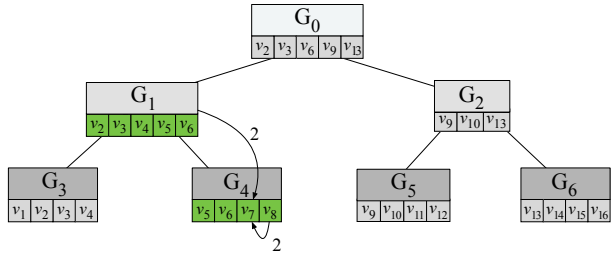


Fig. 6. Searching the global NAV for v_7

Function nnav(v, u)

Input: v: query location; u the previous returned global NAV of v
Output: the next global NAV of v

1 $\widetilde{G} \leftarrow \texttt{Inactivate}(u)$;
2 return $\texttt{gnav}(v)$;

Calculating $\texttt{SPDist}(\beta, v)$ efficiently. Given two ancestor graphs G_m and G_n, $G_n = PG(G_m)$, the SPDist from the boundary vertices of G_n to v are computed based on the SPDist from the boundary vertices of G_m to v, i.e., if $\beta \in B(G_n)$,

$$\texttt{SPDist}(\beta, v) = \min_{\beta_j \in B(G_m)} \texttt{SPDist}(\beta, \beta_j) + \texttt{SPDist}(\beta_j, v).$$

Hence, the SPDist from the borders of G_m to v are repeatedly used when computing the SPDist from the borders of G_n to v. We can utilize the dynamic programming to avoid duplicated computation.

C. *Function* $\texttt{nnav}(v, u)$

Suppose u is the global NAV of v found by the last gnav or nnav call. The function $\texttt{nnav}(v, u)$ finds the next GNAV of v by two steps: (i) inactivating u on V-Tree, and (ii) finding the next GNAV by calling $\texttt{gnav}(v)$, as shown in Function nnav.

Inactivating u on V-Tree. In order to find the next GNAV of v based on the LNAV tables maintained on V-Tree, we need to change the status of u from *active* to *inactive* and let the borders who has u as their local LNAV finds new ones. Note that, this procedure is identical to function Del as if we delete the last moving object of u. However, we cannot directly update the LNAV tables on V-Tree when answering a kNN query due to concurrency control issues. Instead, for each query we maintain a local buffer $\widetilde{G}$ to save the updated LNAV entries. Let $\texttt{Inactivate}(u)$ be the function to update LNAV tables as if u changes from *active* to *inactive*, and the updated entries are saved in $\widetilde{G}$ (line 1 in Function nnav). $\mathcal{L}_{\widetilde{G}}$ will first return values from the local buffer if the local buffer has them, and from the V-Tree otherwise.

Finding The Next GNAV. We directly call $\texttt{gnav}(v)$ on $\widetilde{G}$ with the updated LNAV tables after inactivating u (line 2). $\texttt{gnav}(v)$ returns the current GNAV of v.

$\texttt{nnav}(v)$ works on a slightly modified $\widetilde{G}$ and explores from the leaf node G_v containing v to its ancestor graphs until the global NAV of v is found. Hence, we may explore unchanged

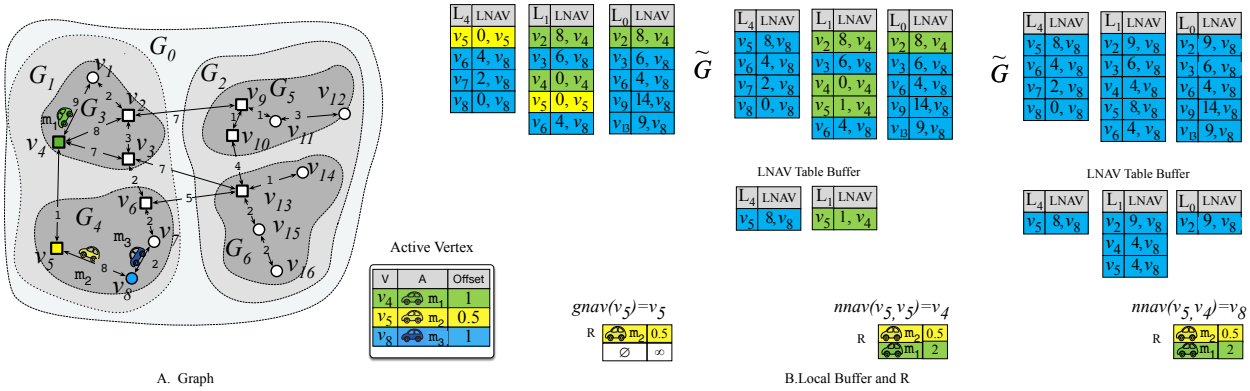


Fig. 7. Progressing of $\texttt{knn}(v_5, 2)$

subgraphs multiple times. To avoid such repeated calculation, we introduce a priority queue Q to record the explored borders and their LNAVDist in functions gnav and nnav. We keep the top entry of Q updated with the most recent $\widetilde{G}$ and the updates of other non-top entries can be delayed (other non-top entries in Q are irrelevant to the final result this time, because LNAVDist of a border can only be increased on the most recent $\widetilde{G}$). We also record the subgraph G_s where we stopped in the last gnav or nnav call. If the early termination condition does not hold anymore, i.e., the current shortest distance in Q becomes larger than $\texttt{SPDist}(G_s, v)$, we need to resume the exploration from G_s and continue with its ancestor graphs until the global NAV of v is found.

For example, we show $\texttt{knn}(v_5, 2)$ in Figure 7. We show the graph and the active object table in Figure 7 (A), and the LNAV tables and R after each iteration in Figure 7 (B), where the local LNAV buffer only keeps the updated entries. In the first iteration, $\texttt{gnav}(v_5) = v_5$. We add $\langle m_2, 0.5\rangle$ to R and $\varepsilon = \infty$. In the second iteration, $\texttt{nnav}(v_5, v_5) = v_4$. We add $\langle m_1, 2\rangle$ to R and $\varepsilon = 2$. In the third iteration, $\texttt{nnav}(v_5, v_4) = v_8$. Since $\texttt{SPDist}(v_8, v_5) > \varepsilon$, the algorithm terminates.

D. Correctness of knn

Theorem 2: Functions gnav, nnav, and algorithm knn correctly find the global NAV, the next global NAV, and the k nearest moving objects of v, respectively.

E. Time Complexity of knn

Lemma 5: Given a graph G and V-Tree, which contains $|V|$ vertices and $|\mathcal{M}|$ moving objects on the road network. Assume moving objects are uniformly distributed on the road network, the average time complexity of kNN search is $\mathcal{O}(\frac{k \cdot |V| \cdot \log min(|V|, |\mathcal{M}|)}{|\mathcal{M}|})$.

V. Experiment

We evaluated the performance of V-Tree and compared with baseline approaches, including index construction, kNN queries, updates, and scalability of V-Tree.

Datasets: Road Network. We used seven real-world road networks, which were widely used in previous studies [31], [16], [25]. The size of the datasets varied from 21,048 to 24 million vertices, as illustrated in Table II.

Datasets: Query and Object. We used two real-world datasets BJTaxi and SpecialCar. BJTaxi was obtained from real taxi trajectories in Beijing, which contained 28,000 taxi trajectories from 8:00am to 9:00am on Sept. 30, 2015. SpecialCar contained trajectories of 16,000 cars from 9:30pm to 10:30pm on July 15, 2015, which was gotten from a company like Uber. We first mapped the positions of moving objects and queries onto the road network using existing methods [4], [24], [19]. The moving objects updated locations in every second. We showed the number of moving objects, queries per hour, and queries per second of BJTaxi and SpecialCar in Table III.

For other six datasets, we synthetically generated moving objects and queries. We randomly selected one percent of vertices as the initial positions of moving objects, and moving objects were moving following the same distribution draw from BJTaxi and SpecialCar. For queries, we generated the queries similar to BJTaxi and SpecialCar (i.e., the number of generated queries per second and the number of active objects). The positions of these queries were randomly selected from the vertices of each dataset, and they evenly arrived during each hour. The detailed statistics were illustrated in Table III.

Baseline. We compared our V-Tree with three state-of-the-art methods, SILC[25], ROAD[16], and G-Tree[31]. The implementations of ROAD and G-Tree were provided by the authors, SILC was implemented by ourselves. As the SILC, ROAD, G-Tree do not support kNN query on moving objects directly, we extended them to support our problem as follows. We first built indexing structures for a (static) road network. Then we created an occurrence index, which keeps a map from a vertex to a list of moving objects that belong to this vertex. If a vertex has some moving objects, we call it an active vertex. Next, given a query we computed kNN active vertices using existing techniques, and added the moving objects belonging to such vertices into the results. Note that we may not need to find k active vertices because an active vertex may contain multiple moving objects and the algorithm will stop when finding k moving objects. For example, G-Tree used a best-first algorithm to find kNN active vertices by traversing the tree from the root. We kept an occurrence index for each tree node. If a tree node had no moving object, we pruned the node; otherwise we accessed its children to compute the active vertices. SILC also used a best-first method to compute kNN results. We also utilized the occurrence index to keep the active vertices, computed top-k active vertices and added moving objects of active vertices into the result set. In ROAD, we also only considered active vertices based on the occurrence index and prune other vertices. When the moving objects were updated, we only updated the occurrence index and recomputed the top-k results using the static index and the updated occurrence index. For G-Tree, we set the leaf capacity τ=32 and the fanout f=4. For ROAD, we set the hierarchy level $l = 8$ and fanout f=4.

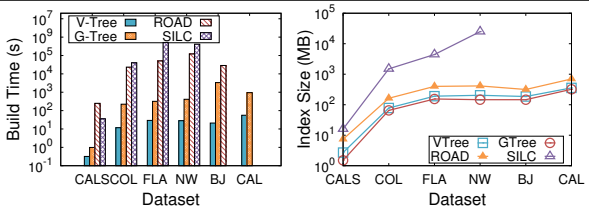
Metrics. Suppose there were n queries $q_1 \cdots q_n$ in a period. Before executing q_i, we needed to update objects. We used the

TABLE II. CHARACTERISTICS OF ROAD NETWORKS

Database	Description	Vertices	Edges
BJ	Beijing	1,278,984	2,677,984
CALS	California	21,048	43,386
COL	Colorado	435,666	1,057,066
FLA	Florida	1,070,376	2,712,798
NW	Northwest USA	1,207,945	2,840,208
CAL	California and Nevada	1,890,815	4,657,742
USA	Full USA	23,947,347	58,333,344

TABLE III. CHARACTERISTICS OF QUERIES AND OBJECTS

Database	Source	Objects	Queries per Second	Updates per Second
BJTaxi	real-world	28000	6	75
SpecialCar	real-world	16000	4	60
CALS	synthetic	210	1	1
COL	synthetic	4357	1	27
FLA	synthetic	10704	2	66
NW	synthetic	12079	3	174
CAL	synthetic	18908	2	116
USA	synthetic	239473	94	1469



(a) Construction Time (b) Index Size

Fig. 8. Index Comparison

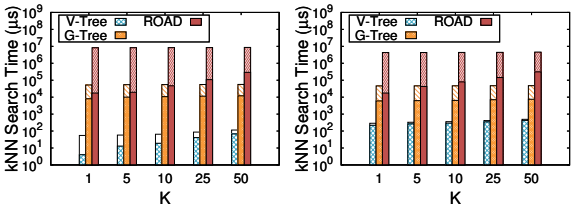
amortized time $T_a = \frac{T_u + \sum_{i=1}^{n} T_{q_i}}{n}$, where T_u and T_q were the update and query time, respectively.

Environment. We conducted experiments on a 64-bit Linux computer with Intel 3.10GHz i5-3450 CPU and 32GB RAM. Our program was compiled with G++ 4.9 using the O3 flag. We did not parallelize the program and used one core.

A. Evaluation on Index Construction

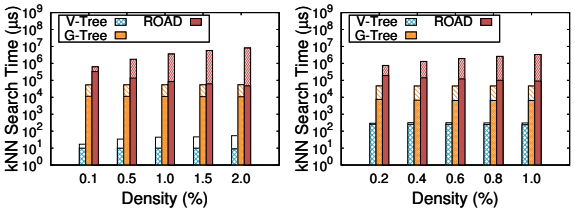
We conducted experiments by varying two parameters: fanout f and the maximum number of vertices in a leaf node τ, and the details are in our technical report [1].

We evaluated the V-Tree index building time and space overhead with G-Tree, SILC, and ROAD on the datasets CALS, COL, FLA, NW, BJ, and CAL. V-Tree consists of tree structure, distance matrices, LNAV tables, and active object tables. Note that the index time of SILC and ROAD was too long for BJ and CAL, and the results were not reported. As can be seen from Figure 8, the time of V-Tree was better than that of SILC and ROAD by almost 3 orders of magnitude, and 1 order of magnitude faster than G-Tree in all the 6 test datasets. For example, on FLA, the construction time of V-Tree was only 29 s, G-Tree consumed 320 s, SILC required 142 hours, and ROAD cost 14 hours. The main reason was that (1) SILC required to compute the shortest path of every two vertices which was rather time consuming, (2) ROAD had to compute and store shortest-path distances of all border pairs, (3) our bottom-up construction method can reduce many unnecessary computations than G-tree. For index size, V-Tree outperformed SILC and ROAD by 1 order of magnitude, and achieved almost the same result as G-Tree. This is because (1) the space overhead of SILC is $\mathcal{O}(|V|^{1.5})$ and it was rather expensive to compute all-pair shortest paths, and (2) ROAD computed larger numbers of borders than our methods and needed to store shortest-path distances of all border pairs. Thus SILC and ROAD took more space and time than ours.



(a) BJTaxi Dataset (b) SpecialCar Dataset

Fig. 9. Varying K on BJTaxi and SpecialCar Datasets



(a) BJTaxi Dataset (b) SpecialCar Dataset

Fig. 10. Varying Density on BJTaxi and SpecialCar Datasets

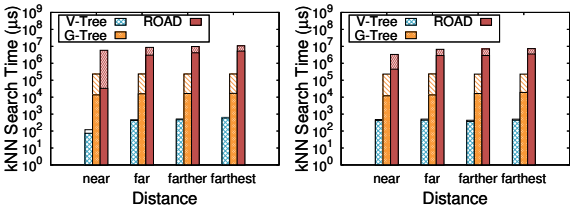
B. kNN Query

We compared our proposed method with state-of-the-art approaches. The performance of kNN was evaluated in two ways. The first is the performance on real-world datasets, where we used the datasets BJTaxi and SpecialCar containing real trajectories of moving objects and request queries on road network of BeiJing. As SILC is too expensive and not able to generate the road index for BJ, we omit this algorithm on real-world datasets. The other is to evaluate the performance on varying datasets of synthetic data. As mentioned, we used the ROAD method with the best performance parameter $t = 4$, and $l = 8$ to generate road index. For V-Tree and G-Tree, we set $f = 4$, and $\tau = 32$.

We evaluated the kNN search efficiency of V-Tree, and used SILC, ROAD, G-Tree as baselines. We varied k, object distribution, datasets, object distance, and updating interval on real-world road networks.

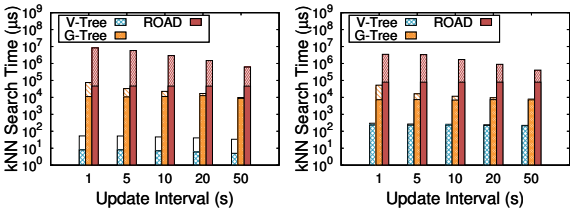
Varying k: We evaluated the performance of V-Tree, ROAD, and G-Tree for kNN query on real-world dataset of taxi and SpecialCar. We varied k in 1, 5, 10, 20, 50. We reported the average time on the real datasets, including kNN search and update time. As can be seen from Figure 9, the whole histogram is the amortized time, and the bottom part is the average single query time and the top part is the average update time. We had the following conclusions. Firstly, the average time of kNN query of V-Tree is the fastest one among the three algorithms. The amortized query time of V-Tree was 65 μs for $k = 10$, while ROAD took nearly 10^7 μs and G-Tree took 10^5 μs. V-Tree method outperformed ROAD by almost 5 orders of magnitude and outperformed G-tree by 2-3 orders of magnitude. Secondly, the single query time of V-Tree was also the best one in the three algorithms. When k was 10, the average query time of V-Tree was 19 μs while ROAD took 10^5 μs and G-Tree took 10^4 μs. This was attributed to the lower updating cost and the efficient query method. ROAD took much time because it involved many times of Dijkstra algorithms to compute the shortest-path distance. G-tree was expensive for updates on moving objects.

Varying Density: We evaluated the performance on varying object densities in Figure 10. There are 28000 taxis and 16000 Special cars, and the number of vertices was 1,278,984. We select 0.1%, 0.5%, 1.0%, 1.5% and 2.0% portion of objects for BJTaxi; 0.2%, 0.4%, 0.6%, 0.8% and 1.0% portion of



(a) BJTaxi Dataset (b) SpecialCar Dataset

Fig. 11. Varying Distance on BJTaxi and SpecialCar Datasets



(a) BJTaxi Dataset (b) SpecialCar Dataset

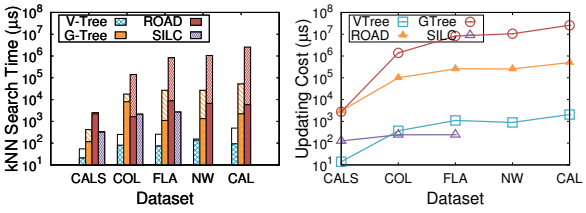
Fig. 12. Varying Updating Interval on BJTaxi and SpecialCar Datasets

objects for SpecialCar. The experiment was conducted by 3,600 queries, and we set k = 10 and evaluated the average time. We made the following three observations. Firstly, V-Tree outperformed state-of-the-art G-Tree and ROAD by 2-3 orders of magnitude for the search time. Secondly, with the increasing of searched objects, the amortized search time of the three algorithms all increases. This is because the overhead of updating increased as the number of objects grew. Thirdly, the single query time of three algorithms all decreased. Because of small number of objects and the sparse distribution of the vertices in the graph, the searching space for the k nearest objects was increasing by the density decreasing. The cost of computation will increase by the farther vertices. The updating cost of V-Tree increased very slightly, which was more suitable for moving objects indexing.

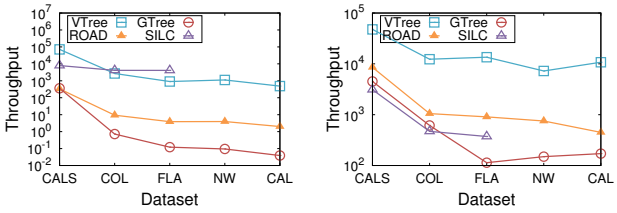
Varying Distance: We evaluated the performance on varying object distances in Figure 11. To generate the objects with varying distances, we first extracted the query position from the real queries. Then we sorted the objects by their distances to the query position. The objects were partitioned into 4 equally sized datasets, by their distance, denoted as near, far, farther, and farthest. We can see that the V-Tree was almost stable when the distance changed, and outperformed ROAD and SILC and G-Tree by 2-6 orders of magnitude. ROAD and SILC needed to compute more vertices for the long distance objects; while on the contrary V-Tree structure can acquire the kNN vertices by accessing fewer tree nodes, no matter how far away the objects were. This means that the method of V-Tree kNN method is suitable for the different distance of query.

Varying Update Interval: We evaluated the amortized query time of the algorithms on two real-world datasets BJTaxi and SpecialCar. We set k = 10 and. We used the positions of moving objects and queries in this period to evaluate the updating cost. As illustrated in Figure 12, we selected the update interval 1 s, 5 s, 10 s, 20 s, and 50 s. We know that the higher frequency of updating, the fewer updating objects in the updating period. We can see that with the update frequency increasing, the cost of amortized query time declined. The reason is the updating cost declines as the longer updating period amortized updating interval. Meanwhile, when updating interval is 1 s, the amortized update time of V-Tree on BJTaxi and SpecialCar is 44μs and 263 μs respectively.

Various Datasets: We evaluated the performance of the algorithm on five datasets CALS, COL, FLA, NW and CAL



(a) kNN Performance (b) Updating Cost



(c) Updating Throughput (d) Query Throughput

Fig. 13. Performance on Varying Datasets

as illustrated in Table III and set $k = 10$. The number of vertices varied from $21,048$ to $1,890,815$. As can be seen from Figure 13, The amortized time of V-Tree significantly outperformed G-Tree, ROAD, by more than 3 orders of magnitude, and more than one order of magnitude compared with SILC. The single query time of V-Tree outperformed other algorithms by at least one order of magnitude. V-Tree not only had small kNN search time, but also had small preprocessing time. Note that although the SILC algorithm was the fastest one of the other algorithms, the index size was too huge, e.g. 24GB for NW. In contrast, the size of V-Tree was only 220MB. Since SILC consumed too large amount of memory and pre-processing time, we did not evaluate it on BJ and CAL datasets. The superiority of throughput of V-Tree is shown in Figure 13(c) and Figure 13(d). In addition, the throughput (including search and update) was slightly decreased with the increase of dataset.

C. Scalability

We evaluated the scalability of V-Tree from three aspects, indexing, search and update. Figure 14 showed the result. Firstly, we evaluated the time and space scalability of V-Tree. We set $k = 10$ and $|\mathcal{M}| = 0.01|V|$, and the moving objects was uniformly distributed. We calculated the average overhead of 10K random kNN queries. We can see from 14(a) that V-Tree had very good scalability as the data size increased from 10^4 to 10^8. The average search time on the US road network with 24 million vertices was only 330.96 μs. Secondly, we evaluated the scalability of the number of moving objects on road network. We randomly generated uniformly distributed vertices as the number of moving objects. We set k = 10, and used BJ. As can be seen form Figure 14(b), searching time of kNN query decreased by the number of the objects. This is for the more objects were indexed, the vertex will be more likely to not change its state between *active* and *inactive*, which means that the influence of moving objects will decrease. This means that our method was suitable for huge number of moving objects indexing on road networks. Thirdly, the throughput (including update and search) slightly decreased with increase of vertices. Fourthly, Table IV showed that the index size of V-Tree was scalable. We evaluated the index size for different datasets, and the number of vertices increased from 21 thousands to 24 millions. We can see that the index size of V-Tree increased almost linearly as the size

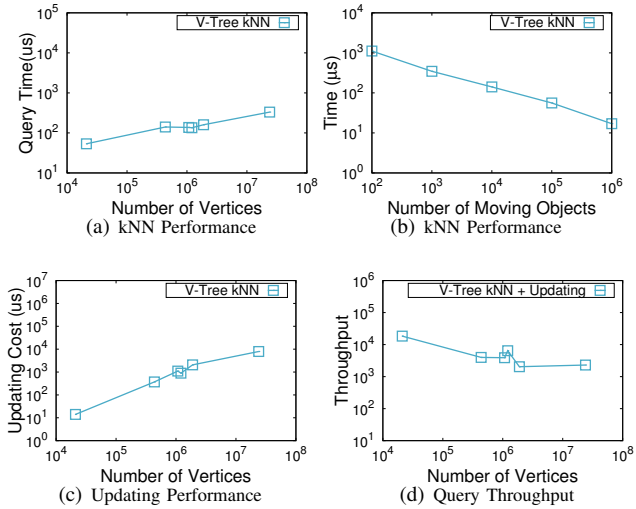


Fig. 14. Scalability of V-Tree

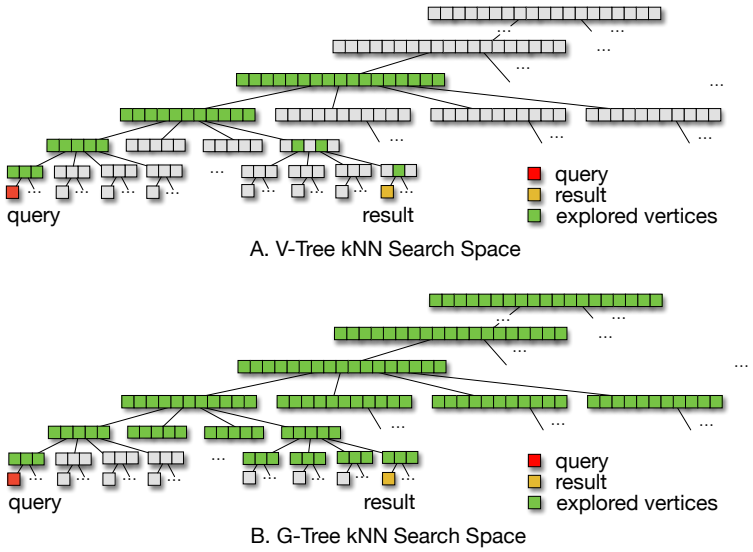


Fig. 15. V-Tree and G-Tree kNN Search Space Overview

of vertices and edges increased.

TABLE IV. INDEX TREE SIZE OF V-TREE

Size	CALS	COL	NW	FLA	BJ	CAL	USA
Vertices	21K	435K	1.2M	1.1M	1.2M	1.8M	24M
Edges	43K	1.1M	2.8M	2.7M	2.7M	4.7M	58M
Data	702K	19M	46M	48M	81M	87M	1270M
V-Tree	3.5M	91M	220M	224M	233M	416M	5639M

VI. RELATED WORKS

A. Single Pair Shortest Path Queries

The single pair shortest path (SPSP) queries, which find the shortest path for two vertices on the road networks, have been extensively studied, e.g.,G-Tree[31][30], HEPV[12][13], HiTi[14], TNR[3], CH[7] and PHL[2]. HEPV[12][13] partitions the graph by cutting vertices and pre-computes all the shortest paths between all border pairs. It is both time consuming and space consuming to store all such pairs, and cannot support large graphs. Furthermore, HEPV only considers three layers and it is not clear how to extend it to support multiple layers. HiTi[14] computes the shortest paths for objects in different subgraphs by using the A* algorithm. It utilizes the Euclidean distance to estimate a lower bound of the road-network distance and then uses the A* algorithm to prune subgraphs withlarger distances. Transit Node Routing (TNR)[3] calculates the distances of each vertex to a set of transit nodes and utilizes the transit nodes to compute the shortest-path distance. Contraction Hierarchies (CH)[7] first pre-computes the road network by appending additional edges. Then bidirectional shortest-path search is used to restrict the edges leading to more important nodes which reduces the search spaces. The Pruned Highway Labeling (PHL)[2] uses highway-based labeling and a preprocessing algorithm to compute the distance. G-Tree[31], [30] is a hierarchy structure to do kNN query, which also supports SPSP query. An assembly-based method is proposed to efficiently compute the shortest path between two vertices. These algorithms cannot support dynamical updates of moving objects either.

B. kNN Query on Road Networks

Recently, kNN query on road networks has been extensively studied, such as INE[22], IER[22], ROAD[16], SILC[25], G-Tree[31][30]. Incremental Euclidean Restriction(IER)[22] uses Euclidean distance as a pruning bound to acquire the kNN results. Incremental Network Expansion(INE)[22] improved IER Euclidean distances bound by expending searching space from the query location. SILC[25] precomputed the shortest paths between all possible vertices in the network and then made use of Quadtree-based encoding method to reduce the storage cost. Route Overlay and Association Directory(ROAD)[16] and G-Tree [31], [30] extend HiTi and HEPV to support kNN search. ROAD uses a hierarchical structure, which recursively partitions the whole graph to a hierarchy of interconnected regional sub-networks. The *shortcuts* between the partitioned vertices are precomputed. It uses the INE-like method to compute the kNN results by using shortcuts to compute a tighter bound. G-Tree is a hierarchy structure to compute kNN results on road networks. It also uses a hierarchical structure and adopts an assembly-based method to efficiently compute kNN results. Different from our method, these approaches assume that the locations of objects are not frequently changed. In our problem, obviously the locations of objects are dynamically changed and thus existing algorithms cannot efficiently support our problem.

V-Tree extends G-Tree to support moving objects. G-Tree is better than ROAD because G-Tree uses the dynamic-programing algorithm to compute the shortest-path distance for two objects across different subgraphs while ROAD uses the Dijkstra algorithm to compute the shortest paths based on the boundary nodes in subgraphs. SILC takes $|V|^{1.5}$ space and could not support large graphs. Next we explain why V-Tree is much better than G-Tree. Firstly, in G-Tree, it can only know which subtrees contain active vertices, and it needs to use the dynamic-programming algorithm to compute the distance for objects in different subgraphs. In V-Tree, we utilize the `LNAV` structure to keep the nearest active vertex for each border. If the query and the active vertex are in the same node, it can directly retrieve the active vertex. If the query and the active vertex are not in the same node, it can utilize the `LNAV` structure to efficiently find the active vertex. In this way, we can avoid duplicated computation using the `LNAV` structure. Secondly, G-Tree uses a top-down manner to traverse the tree structure and it may visit unnecessary nodes as shown in Figures 15. V-Tree uses a down-up manner, and it only visits the active vertices based on the `LNAV` structure. Thus the search space of V-Tree is much smaller than G-Tree.

C. Moving Objects Query

There are some studies on finding kNN moving objects with Euclidean distance, e.g., TPR-Tree[26], B^x-Tree[11], STR-Tree[23], TB-tree[23], DSI[29], V^*kNN[21],

MOVNet[27]. The Time-parameterized R-tree (TPR-tree) [26] extends R-tree to index moving point objects by Euclidean distance. The Spatio-Temporal R-tree (STR-tree) [23] extends R-tree to index spatio-temporal objects. Trajectory-Bundle tree (TB-tree) uses a hybrid structure to process the trajectories of moving objects. B^x-Tree[11] enables the B^+-Tree to support range query and kNN queries and continuous queries. Shortest-Distance-based Tree(SD-Tree)[28] reduces the continuous query update cost by precomputing some vertices distances. Dynamic Strip Index(DSI)[29] uses the strip index structure to support distributed processing on kNN queries of moving objects. MOVNet[27] uses an in-memory grid structure to index moving objects. These studies focus on identifying kNN results for continuous queries. However they cannot support kNN search on road-network distances.

D. Continuous Query

Some studies focus on continuous k-NN queries [20], [17], DLMTree[8], COMET[6], which study how to continuously answer a query, but they do not focus on efficient kNN search on moving objects in large-scale road networks. Specifically, [20] uses an incremental monitoring algorithm and a group monitoring algorithm to share the computation on moving objects. [17] utilizes the driving directions and speeds to reduce unnecessary computations. DLMTree[8] focuses on continuous reverse k nearest queries in road networks. COMET[6] proposes a collaborative framework that combines different techniques, e.g, safe segment and influence segment, to reduce the search space. Thus they focus on optimizing a continuous query and do not emphasize on optimizing kNN search for a large number of online queries in road networks.

VII. Conclusion

In this paper we have studied the kNN search on moving objects with road-network constraints. We proposed a novel tree structure `V-Tree`. We devised an efficient algorithm to construct the index. We developed efficient strategies to support updates of moving objects. We designed an efficient kNN search algorithm using `V-Tree`. Experimental results show that our method significantly outperforms baseline algorithms.

Acknowledgment

This work was supported by 973 Program of China (2015CB358700), NSF of China (61373024, 61422205, 61661166012, 61632016), Shenzhou, Tencent, FDCT/116/2013/A3, and FDCT/007/2016/AFJ.

References

[1] V-tree: Efficient knn search on moving objects with road-network constraints. In *Tsinghua Technical Report, http://dbgroup.cs.tsinghua.edu.cn/ligl/vtree.pdf*.

[2] T. Akiba, Y. Iwata, K. Kawarabayashi, and Y. Kawata. Fast shortest-path distance queries on road networks by pruned highway labeling. In *ALENEX*, pages 147–154, 2014.

[3] H. Bast, S. Funke, D. Matijevic, P. Sanders, and D. Schultes. In transit to constant time shortest-path queries in road networks. In *ALENEX*, 2007.

[4] L. Cao and J. Krumm. From gps traces to a routable road map. In *GIS*, pages 3–12, New York, NY, USA, 2009. ACM.

[5] H. J. Cho and R. Jin. Efficient processing of moving k-range nearest neighbor queries in directed and dynamic spatial networks. *Mobile Information Systems*, 2016, 2016.

[6] H. J. Cho, R. Jin, and T. S. Chung. A collaborative approach to moving -nearest neighbor queries in directed and dynamic road networks. *Pervasive and Mobile Computing*, 17, Part A:139 – 156, 2015.

[7] 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.

[8] L. Guohui, L. Yanhong, L. Jianjun, L. Shu, and Y. Fumin. Continuous reverse k nearest neighbor monitoring on moving objects in road networks. *Information Systems*, 35(8):860–883, 2010.

[9] H. Hu, J. Xu, and D. L. Lee. A generic framework for monitoring continuous spatial queries over moving objects. In *SIGMOD*, pages 479–490. ACM, 2005.

[10] W. Huang, G. Li, K. Tan, and J. Feng. Efficient safe-region construction for moving top-k spatial keyword queries. In *CIKM*, pages 932–941, 2012.

[11] C. S. Jensen, D. Lin, and B. C. Ooi. Query and update efficient b+-tree based indexing of moving objects. In *VLDB*, pages 768–779. VLDB Endowment, 2004.

[12] N. Jing, Y. Huang, and E. A. Rundensteiner. Hierarchical optimization of optimal path finding for transportation applications. In *CIKM '96, Proceedings of the Fifth International Conference on Information and Knowledge Management, November 12 - 16, 1996, Rockville, Maryland, USA*, pages 261–268, 1996.

[13] N. Jing, Y. 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.

[14] S. Jung and S. Pramanik. An efficient path computation model for hierarchically structured topographical road maps. *IEEE Trans. Knowl. Data Eng.*, 14(5):1029–1046, 2002.

[15] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998.

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

[17] G. Li, P. Fan, Y. Li, and J. Du. An efficient technique for continuous k-nearest neighbor query processing on moving objects in a road network. In *CIT*, pages 627–634. IEEE, 2010.

[18] R. J. Lipton and R. E. Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, 1979.

[19] Y. Lou, C. Zhang, Y. Zheng, X. Xie, W. Wang, and Y. Huang. Map-matching for low-sampling-rate GPS trajectories. In *GIS*, pages 352–361, 2009.

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

[21] S. Nutanong, R. Zhang, E. Tanin, and L. Kulik. V*-knn: An efficient algorithm for moving k nearest neighbor queries. In *ICDE*, pages 1519–1522, 2009.

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

[23] D. Pfoser, C. S. Jensen, Y. Theodoridis, et al. Novel approaches to the indexing of moving object trajectories. In *Proceedings of VLDB*, pages 395–406, 2000.

[24] S. Rogers, P. Langley, and C. Wilson. Mining gps data to augment road models. In *SIGKDD*, pages 104–113, New York, NY, USA, 1999. ACM.

[25] H. Samet, J. Sankaranarayanan, and H. Alborzi. Scalable network distance browsing in spatial databases. In *SIGMOD*, pages 43–54, New York, NY, USA, 2008. ACM.

[26] S. $\mathcal{V}$Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. *SIGMOD*, 29(2):331–342, May 2000.

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

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

[29] Z. Yu, Y. Liu, X. Yu, and K. Q. Pu. Scalable distributed processing of k nearest neighbor queries over moving objects. *TKDE*, 27(5):1383–1396, 2015.

[30] R. Zhong, G. Li, K. Tan, and L. Zhou. G-tree: an efficient index for KNN search on road networks. In *CIKM*, pages 39–48, 2013.

[31] R. Zhong, G. Li, K. L. Tan, L. Zhou, and Z. Gong. G-tree: An efficient and scalable index for spatial search on road networks. *TKDE*, 27(8):2175–2189, Aug 2015.