Research Article

A Novel Index Method for K Nearest Object Query over Time-Dependent Road Networks

Yajun Yang 1,2, Hanxiao Li 1, Junhu Wang 3, Qinghua Hu 1, Xin Wang 1, and Muxi Leng 1

1 College of Intelligence and Computing, Tianjin University, China
2 State Key Laboratory of Digital Publishing Technology, China
3 School of Information and Communication Technology, Griffith University, Australia

Correspondence should be addressed to Xin Wang; wangx@tju.edu.cn

Received 30 November 2018; Accepted 28 January 2019; Published 24 February 2019

Abstract

K nearest neighbor (kNN) search is an important problem in location-based services (LBS) and has been well studied on static road networks. However, in real world, road networks are often time-dependent, i.e., the time for traveling through a road always changes over time. Most existing methods for kNN query build various indexes maintaining the shortest distances for some pairs of vertices on static road networks. Unfortunately, these methods cannot be used for the time-dependent road networks because the shortest distances always change over time. To address the problem of kNN query on time-dependent road networks, we propose a novel voronoi-based index in this paper. Furthermore, we propose a novel balanced tree, named V-tree, which is a secondary level index on voronoi-based index to make our querying algorithm more efficient. Moreover, we propose an algorithm for preprocessing time-dependent road networks such that the waiting time is not necessary to be considered. We confirm the efficiency of our method through experiments on real-life datasets.

1. Introduction

With the rapid development of mobile devices, k nearest neighbor (kNN) search on road networks has become more and more important in location-based services (LBS) [1–4]. Given a query location and a set of objects (e.g., restaurants) on a road network, it is to find k nearest objects to the query location. KNN search problem has been well studied on static road networks. However, road networks are essentially dynamic complex networks but not static in real world [5–8]. The “dynamic” means that the traveling time on road network is always time-dependent. For example, the Vehicle Information and Communication System (VICS) and the European Traffic Message Channel (TMC) are two transportation systems, which provide real-time traffic information to users. Such road networks are time-dependent; i.e., travel time for a road varies with taking “rush hour” into account.

The existing works propose various index techniques for answering k nearest object query on road networks. The main idea behind these indexes is to partition the vertices into several clusters, and then the clusters are organized as a voronoi diagram or a tree (e.g., R-tree, G-tree). All these methods precompute and maintain the shortest distances for some pairs of vertices to facilitate kNN query. Unfortunately, these indexes cannot be used for time-dependent road networks. The reason is that the minimum travel time between two vertices often varies with time. For example, u and v are in the same cluster for one time period but they may be in two distinct clusters for another time period because of the minimum travel time varying with time. Therefore, the existing index techniques based on the static shortest distance cannot handle the case that the minimum travel time is time-dependent. Moreover, the waiting time is allowed on time-dependent road networks; i.e., someone can wait a time period to find another faster path. When the waiting time is considered, it is more difficult to build an index for kNN query by existing methods because it is difficult to estimate an appropriate waiting time for precomputing the minimum travel time between two vertices.
Recently, there are some works about kNN query on time-dependent graphs [9–12]. Most of these works utilize A* algorithm to expand the road networks by estimating an upper or lower bound of travel time. There are two main drawbacks of these methods. First, in these works, the FIFO (first in first out) property is required for the networks and the waiting time is not allowed. Second, the indexes proposed by these works are based on the estimated value of travel time. However, these indexes cannot facilitate query effectively for large networks because the deviations are always too large between the estimated and the actual travel time.

In this paper, we study k nearest object query on time-dependent road networks. A time-dependent road network is modeled as a graph with time information. The weight of every edge is a time function \( w_{i,j}(t) \) which specifies how much time it takes to travel through the edge \((v_i, v_j)\) if departing at time point \( t \). The main idea of our method is to precompute minimum travel time functions (or mtt-function for short) instead of concrete values for some pairs of vertices and then design a “dynamic” voronoi-based index based on such functions. Here “dynamic” means that in a time-dependent network it can be easily decided which cluster a vertex should be in for any given time point \( t \). Furthermore, a secondary level index is built on voronoi-based index, which makes our querying algorithm more “smart” for \( k \) nearest objects searching; that is, the \( k \) nearest objects will be searched and the others will be filtered as early as possible. Different to previous works, our index can facilitate query effectively for large networks. Moreover, our method does not require the FIFO property for networks and we allow waiting time on every vertex.

The main contributions of this paper are summarized as below. First, we propose an algorithm to process \( w_{i,j}(t) \) for every edge such that the waiting time is not necessary to be considered. Let \( G_T \) and \( G^*_T \) be the original graph and the graph after processing \( w_{i,j}(t) \). We can prove that a shortest path with consideration of waiting time on \( G_T \) is one-to-one mapped to shortest path without waiting time on \( G^*_T \). Furthermore, we show how to compute the mtt-function for two vertices. Second, we propose a novel voronoi-based index for time-dependent road networks and an algorithm to answer kNN query using our index. Third, we propose a novel balanced tree structure, named V-tree, which can be considered as the secondary level index on voronoi-based index to make \( k \) nearest object query more efficient. Finally, we confirm the efficiency of our method through extensive experiments on real-life datasets.

The rest of this paper is organized as follows. Section 2 gives the problem statement. Section 3 describes how to process \( w_{i,j}(t) \) and compute the mtt-function. Section 4 explains how to build the voronoi-based index for time-dependent networks. Section 5 introduces the V-tree and how to answer the \( k \) nearest object query using it. The experimental results are presented in Section 6. The related work is in Section 7. Finally, we conclude this paper in Section 8.

2. Problem Statement

Definition 1 (time-dependent road network). A time-dependent road network is a simple directed graph, denoted as \( G_T(V, E, W) \) (or \( G_T \) for short), where \( V \) is the set of vertices; \( E \subseteq V \times V \) is the set of edges; and \( W \) is a set of nonnegative value function. For every edge \((v_i, v_j) \in E\), there is a time function \( w_{i,j}(t) \in W \), where \( t \) is a time variable. A time function \( w_{i,j}(t) \) specifies how much time it takes to travel from \( v_i \) to \( v_j \), if one departs from \( v_i \) at time point \( t \).

In this paper, we assume that \( w_{i,j}(t) \geq 0 \). The assumption is reasonable, because the travel time cannot be less than zero in real applications. Our work can be easily extended to handle undirected graphs. An undirected edge \((v_i, v_j)\) is equivalent to two directed edges \((v_i, v_j)\) and \((v_j, v_i)\), where \( w_{i,j}(t) = w_{j,i}(t) \).

There are several works that study how to construct time function \( w_{i,j}(t) \), which is always modeled as a piecewise linear function [6, 7, 13] and it can be formalized as follows:

\[
    w_{i,j}(t) = \begin{cases} 
    a_1t + b_1, & t_0 \leq t < t_1 \\
    a_2t + b_2, & t_1 \leq t < t_2 \\
    \vdots \\
    a_pt + b_p, & t_{p-1} \leq t \leq t_p 
    \end{cases} \tag{1}
\]

Given a path \( p \), the travel time of \( p \) is time-dependent. In order to minimize the travel time, some waiting time \( \omega_t \) is allowed at every vertex \( v_i \) in \( p \). That is, when arriving at \( v_i \), one can wait a time period \( \omega_t \) if the travel time of \( p \) can be minimized. We use \( \text{arrive}(v_i) \) and \( \text{depart}(v_i) \) to denote the arrival time at \( v_i \) and departure time from \( v_i \), respectively.

For each \( v_i \) in \( p \), we have

\[
\text{depart}(v_i) = \text{arrive}(v_i) + \omega_t, \tag{2}
\]

Let \( p = v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_n \) be a given path with the departure time \( t \) and the waiting time \( \omega_t \) for each vertex \( v_i \); then we have

\[
    \begin{align*}
    \text{arrive}(v_1) &= t \\
    \text{arrive}(v_2) &= \text{depart}(v_1) + w_{1,2}(\text{depart}(v_1)) \\
    \vdots \\
    \text{arrive}(v_n) &= \text{depart}(v_{n-1}) + w_{n-1,n}(\text{depart}(v_{n-1}))
    \end{align*} \tag{3}
\]

Thus the travel time of path \( p \) is \( w(p) = \text{arrive}(v_n) - t \).

Given two vertices \( v_i \) and \( v_j \) in \( G_T \), the minimum travel time from \( v_i \) to \( v_j \) with departure time \( t \) is defined as \( m_{i,j}(t) = \min\{w(p) \mid p \in P_{i,j}\} \), where \( P_{i,j} \) is the set of all the paths from \( v_i \) to \( v_j \) in \( G_T \). Obviously, \( m_{i,j}(t) \) is also a function related to the departure time \( t \). We call \( m_{i,j}(t) \) the minimum travel time function (or mtt-function shortly) from \( v_i \) to \( v_j \). Letting \( |V| \) be \( n \), in the following, we use \( m_{i,j}(t) \) to represent mtt-function from a vertex \( v_i \) to an object \( o_j \), in order to distinguish from \( m_{i,j}(t) \) from \( v_i \) to a vertex \( v_j \). Note that an object \( o_j \) is also a vertex regarded as \( v_{n+j} \) in the network.
There is a special case that the waiting time is only allowed on a subset $V_w$ of $V$; that is, people are allowed to wait a time period $\omega$ only when they are at the vertex $v_i \in V_w$. For this case, we only need to consider that the waiting time is always zero, i.e., $\omega_i \equiv 0$, for every vertex $v_i \in V \setminus V_w$. Therefore, the method proposed in this paper can be extended to handle this case easily.

Next, we give the definition of kNN query over time-dependent road networks.

**Definition 2** (k nearest objects on time-dependent road networks). Given a time-dependent road network $G_T(V, E, W)$, a set of the objects $O = \{o_1, o_2, \cdots \}$, a query point $v_q \in V$, and a departure time $t_d$, $k$ nearest object query of $v_q$ is to find a $k$-size subset $O(v_q) \subseteq O$, such that $m_{x,y}^T(t_d) \geq \max \{m_{x,y}^T(t_d) \mid o_i \in O(v_q)\}$ for every object $o_i \in O \setminus O(v_q)$.

### 3. Minimum Travel Time Function

We precompute mtt-functions for some pairs of vertices and then build the index to facilitate kNN query over time-dependent road networks. In this section, we first describe how to process the time function $w_{ij}(t)$ for every edge in $G_T$ such that the waiting time is not necessary to be considered when computing mtt-function and then we explain how to compute mtt-function without waiting time.

#### 3.1 Preprocessing Time Function for Every Edge

Given a path $p$, the waiting time $\omega_i$ is allowed for any vertex $v_i \in p$. However, it is not easy to find an appropriate value of $\omega$ for every $v_i \in p$ to minimize the travel time of $p$. In this section, we propose an algorithm to convert time function $w_{ij}(t)$ to a new function $w_{ij}^*(t)$ for every edge $(v_i, v_j) \in E$. We call $w_{ij}^*(t)$ the “no waiting time function” of edge $(v_i, v_j)$ (or nwt-function for short). The waiting time can be considered as zero when nwt-function is used to compute the minimum travel time of path $p$. The nwt-function $w_{ij}^*(t)$ is defined by the following:

$$w_{ij}^*(t) = \min_{\omega_i} \left( \omega_i + w_{ij}(t + \omega_i) \right) \tag{4}$$

Note that when the waiting time is only allowed on a vertex subset $V_w$, we only need to calculate nwt-function $w_{ij}^*(t)$ for every vertex $v_i \in V_w$. For every vertex $v_i \in V \setminus V_w$, the nwt-function $w_{ij}^*(t)$ is exactly the time function $w_{ij}(t)$ because $\omega_i \equiv 0$; that is, anyone is not allowed to wait a time period $\omega_i$ at $v_i$.

The following theorem guarantees that the nwt-function $w_{ij}^*(t)$ can be used to compute the minimum travel time for any path $p$ in $G_T$ without waiting time.

**Theorem 3.** Given two time-dependent graphs $G_T(V, E, W)$ and $G_T^*(V, E, W^*)$, where $W^*$ is the set of nwt-functions of all edges in $E$, for any path $p$ in $G_T$, the minimum travel time of $p$ in $G_T$ with consideration of waiting time equals the minimum travel time of $p$ in $G_T^*$ without waiting time.

**Proof.** Let $p = v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_h$ be a given path with the departure time $t$. $\omega_i^*$ is the waiting time on vertex $v_i (1 \leq i \leq h)$ minimizing the travel time of $p$ in $G_T$. We have

$$\text{depart}(v_i) = \text{arrive}(v_i) + \omega_i^*$$

and $\text{arrive}(v_i) = \text{depart}(v_{i+1})$. Similarly, we have

$$\text{depart}^*(v_i) = \text{arrive}^*(v_i) + \omega_i^*$$

and $\text{arrive}^*(v_i) = \text{depart}^*(v_{i+1})$ for $G_T^*$. We only need to prove $\text{arrive}^*(v_i) = \text{arrive}^*(v_i)$. It can be easily proved by induction on $v_i$. We omit it due to the space limitation.

The algorithm to compute nwt-function is shown in Algorithm 1. For every $w_{ij}(t) \in W$, Algorithm 1 computes $w_{ij}^*(t)$ backward from $[t_{p+1}, t_p]$ to $[t_0, t_1]$ iteratively. In each iteration, $w_{ij}^*(t)$ for $t \in [t_{k-1}, t_k]$ is computed. Algorithm 1 first sets $w_{ij}^*(t_k)$ as $a^* + b^*$, where $a^* = -1$ and $b^* = t_k + \phi$, $\phi$ is the minimum value between $w_{ij}^*(t_k)$ and $w_{ij}^*(t_k)$. $w_{ij}^*(t_k)$ is the left limit value of $w_{ij}(t)$ on $t_k$. Note that $w_{ij}^*(t_k)$ and $\phi$ have been computed in the last iteration; i.e., the iteration for computing $w_{ij}^*(t)$ on $[t_{k-1}, t_k]$ is computed. Note that $w_{ij}^*(t_k)$ is defined by

$$w_{ij}^*(t_k) = \min \{w_{ij}(t_k), w_{ij}^*(t_k), w_{ij}^*(t_k-1)\}$$

$w_{ij}(t)$ is computed for $t \in [t_0, t_1]$. Note that if $v_j \notin V_w$, $w_{ij}^*(t)$ is computed as $w_{ij}^*(t)$ immediately.

The time and space complexities analysis for Algorithm 1 are given below. Let $m$ and $n$ be the number of the vertices and edges in $G_T$, respectively. For every edge $(v_i, v_j)$, Algorithm 1 needs to compute $w_{ij}^*(t)$ on $[t_{k-1}, t_k]$ iteratively from $k = p$ to 1. For every time interval $[t_{k-1}, t_k]$, $w_{ij}^*(t)$ can be computed in constant time. Therefore, the time complexity of Algorithm 1 is $O(mp)$. Moreover, Algorithm 1 needs to maintain $w_{ij}^*(t)$ and then the space complexity is also $O(mp)$.

**Example 4.** We illustrate how to compute $w_{ij}^*(t)$ by an example in Figure 1. As the solid black line in Figure I(a), $w_{ij}(t)$ is a piecewise linear function:

$$w_{ij}(t) = \begin{cases} 15, & 10 \leq t < 20 \\ -2t + 55, & 20 \leq t \leq 25 \end{cases} \tag{5}$$

In the first iteration, $\phi$ is initialized as $w_{ij}(25) = 5$ and then $b^* = 25 + \phi = 30$. As the dashed red line in the right-side of Figure 1(a), we find that $a^* + b^* = -t + 30$ is always less than $w_{ij}(t)$ on $[20, 25]$, then $w_{ij}^*(t) = -t + 30$ for $t \in [20, 25]$, and $\phi$ is updated as 10. Similarly, in the second iteration, $w_{ij}^*(t)$ on $[10, 20]$ is computed as $\min[15, -t + 30]$, i.e., $w_{ij}^*(t) = 15$ for $t \in [10, 15]$ and $w_{ij}^*(t) = -t + 30$ for $t \in [15, 20]$. Then $\phi$ is updated as $\min[w_{ij}^*(10), w_{ij}^*(10)] = 15$. In the final iteration, as the dashed red line in the left-side of Figure 1(a), $a^* + b^* = -t + 25$ is always larger than $t + 5$ on $[0, 10]$, and we have $w_{ij}^*(t) = -t + 25$ for $t \in [0, 10]$.
Theorem 5. \( w_{i,j}^*(t) \) computed by Algorithm 1 is exactly the nwt-function \( w_{i,j}^*(t) \) given by (4).

Proof. We proved it by induction on \( p \).

Basis. We need to prove that \( w_{i,j}^*(t) \) on time interval \([t_{p-1}, t_p]\) can be correctly computed by Algorithm 1. First, \( \omega_i^* \) only be zero when \( t = t_p \), then we have \( w_{i,j}^*(t_p) = w^*_j(t_p) \) and \( \phi_p^* = u_j(t_p) \). Next, we consider the case of \( t \in [t_{p-1}, t_p) \).

By the definition of \( w_{i,j}^*(t) \), we have

\[
\begin{align*}
w_{i,j}^*(t) &= \min_{\omega_i^*} \left( \omega_i + u_{i,j}(t + \omega_i^*) \right) \\
&= \min_{\omega_i^*} \left( \omega_i + a_p(t + \omega_i^*) + b_p \right) \\
&= \min_{\omega_i^*} \left( (a_p + 1) \omega_i + a_p t + b_p \right) \\
&= \min_{\omega_i^*} \left( (a_p + 1) \omega_i + w_{i,j}(t) \right)
\end{align*}
\]

For \( t \in [t_{p-1}, t_p) \), if \( a_p \geq -1, w_{i,j}^*(t) \) cannot decrease with \( \omega_i^* \) increasing. It means that \((a_p + 1)\omega_i + w_{i,j}(t)\) is minimum when \( \omega_i = 0 \) and then \( w_{i,j}^*(t) = w_{i,j}(t) \). If \( a_p < -1, w_{i,j}^*(t) \) will decrease with \( \omega_i^* \) increasing and thus \((a_p + 1)\omega_i + w_{i,j}(t)\) is minimum when \( \omega_i = t_p - t \), which is the longest waiting time on \( v_i \) for \( t \in [t_{p-1}, t_p) \). Then we have

\[
w_{i,j}^*(t) = \left( a_p + 1 \right) \left( t_p - t \right) + a_p t + b_p = -t + t_p + \phi_p \tag{8}
\]

Obviously, \( w_{i,j}(t) \) decreases when \( a_p \geq -1 \) and \( \omega_i \geq t_k - t \) and \( w_{i,j}(t) \) increases when \( a_p < -1 \). Then we have \( w_{i,j}^*(t) = \min \{ w_{i,j}(t), -t + t_p + \phi_p \} \) for \( t \in [t_{p-1}, t_p] \).

Induction. Assuming that the correct \( w_{i,j}^*(t) \) can be computed by Algorithm 1 for \( t \in [t_{k}, t_{p}] \), then we need to prove it also can be correctly computed for \( t \in [t_{k-1}, t_k] \). We consider the following two cases: (1) \( \omega_i \geq t_k - t \) and (2) \( \omega_i < t_k - t \).

For case (1), the departure time \( t + \omega_i \in [t_k, t_p] \) because \( \omega_i \geq t_k - t \). By the assumption, nwt-function \( w_{i,j}^*(t) \) has been correctly computed for \( t \in [t_{k-1}, t_k] \); then \( w_{i,j}^*(t_{k-1}) \) is the minimum travel time for edge \((v_i, v_j)\) with departure time \( t_k \). Therefore, \( w_{i,j}^*(t) \) for \( t \in [t_{k-1}, t_k] \) can be computed by the following:

\[
w_{i,j}^*(t) = t_k - t + w_{i,j}^*(t_k) \tag{9}
\]

For case (2), because \( \omega_i < t_k - t \), then \( t + \omega_i \in [t_{k-1}, t_k) \). Similar to the proof of basis, we have

\[
w_{i,j}^*(t) = \min \{ w_{i,j}(t), -t + t_k + w_{i,j}^*(t_k) \} \tag{10}
\]
Note that when \( a_k < -1 \), \( w_{ij}^*(t) = -t + t_k + w_{ij}^-(t_k) \) because \( w_{ij}(t) \) may be noncontinuous at \( t_k \). Therefore, we have

\[
   w_{ij}^*(t) = \min \left\{ w_{ij}(t), -t + t_k + w_{ij}^-(t_k) \right\}
\]

(11)
The proof is completed.

3.2. Computing Minimum Travel Time Function. We adopt a Dijkstra-based algorithm proposed in [13] to compute \( mtt \)-function for two vertices \( v_i \) and \( v_j \) in \( G_T \). This algorithm is only used in case the waiting time is not allowed. After converting \( w_{ij}(t) \) to \( mtt \)-function \( w_{ij}^*(t) \) for every edge in \( G_T \) by Algorithm 1, this algorithm can be used for time-dependent graphs with waiting time.

The main idea of this Dijkstra-based algorithm is to refine a function \( g_{ij}(t) \) iteratively for every \( v_i \in V \), where \( g_{ij}(t) \) represents the earliest arrival time on \( v_j \) if departing from \( v_i \) at time point \( t \). In every iteration, algorithm selects a vertex \( v_k \in V \) and then refines \( g_{kj}(t) \) by extending a time domain \( I_s \) to a larger \( I'_s \), where \( I_s = [t_0, t_s] \) is a subinterval of the whole time domain \( I \). \( g_{ij},(t) \) is regarded as well refined in \( I_s \) if it specifies the earliest arrival time at \( v_j \) for every vertex \( v_j \) for any departure time \( t \in I_s \). The algorithm repeats time-refinement process till \( g_{ij}(t) \) of destination \( v_j \) has been well refined in whole time domain \( I \) and then \( mtt \)-function \( m_{ij}(t) \) can be computed as \( m_{ij}(t) = g_{ij}(t) - t \). The more details about this Dijkstra-based algorithm is given in [13]. As shown in [13], the time and space complexities are \( O(n \log n + m \alpha(T)) \) and \( O((n + m)\alpha(T)) \), respectively, where \( \alpha(T) \) is the cost required for each function (defined in interval \( T \) operation).

4. The Novel Voronoi-Based Index

We propose a novel voronoi-based index for \( k \)NN query over time-dependent road networks. In static road networks, the voronoi diagram divides the network (or space) into a group of disjoint subgraphs (or subspace) where the nearest object of any vertex inside a subgraph is the object generating this subgraph. However, in time-dependent road networks, the nearest object of a vertex may be dynamic. The nearest object of a vertex \( v \) may be \( o_j \) for departure time \( t \in [t_1, t_2] \) but it may be \( o_i \) for \( t \in [t_3, t_4] \). The main idea of our novel voronoi-based index is also to divide the vertex set \( V \) into some vertex subsets \( V_i \) and every subset \( V_i \) is associated with one object \( o_i \in O \). Different from static road networks, our voronoi-based index is time-dependent; that is, every vertex \( v \) inside a subset is with a time interval indicating when the object \( o_i \) is nearest to \( v \). Next, we describe what the novel voronoi-based index is and how to construct it.

4.1. What Is the Voronoi-Based Index? Given a vertex \( v \) and an object \( o_i, I_i(v) \) is called \( v 's \) maximum time interval about \( o_i \) if it satisfies the following two conditions: (1) \( o_i \) is the nearest object of \( v \) for any departure time \( t \in I_i(v) \) but it does not exist another \( I_i'(v) \supseteq I_i(v) \) satisfying the condition (1). Note that \( I_i(v) \) may not be a continuous time interval; that is, if \( o_i \) is nearest to \( v \) for two disjoint departure time intervals \( [t_1, t_2] \) and \( [t_3, t_4] \), then \( [t_1, t_2] \cup [t_3, t_4] \subseteq I_i(v) \). The voronoi-based index maintains a set \( C_i \) for every object \( o_i \in O \), where \( C_i \) is a set of the tuples \( (v, I_i(v)) \) for all the vertices \( v \) with nonempty \( I_i(v) \), i.e.,

\[
   C_i = \{ (v, I_i(v)) \mid v \in V \land I_i(v) \neq \emptyset \}
\]

(12)

We call \( C_i \) the closest vertex-time pair set (or closest pair set shortly) of \( o_i \). For simplicity, we say \( v \) is a vertex in \( C_i \) if \( (v, I_i(v)) \in C_i \). Next, we give the definition of the border vertex.

**Definition 6** (border vertex). A vertex \( v_x \) in \( C_i \) is called a border vertex of \( C_i \) if there exist \( v_y \in N^+(v_x) \) such that \( (v_y, I_y) \notin C_i \) for any \( I_y \supseteq f_{x,y}(I_i(v_x)) \), where \( N^+(v_x) \) is the outgoing neighbor set of \( v_x \) and \( f_{x,y}(I_i(v_x)) \) is the time interval mapped from \( I_i(v_x) \) by the function \( f_{x,y}(t) = t + w^*_{x,y}(t) \).

The border vertex \( v_x \) of \( C_i \) indicates that there exists a time point \( t \in f_{x,y}(I_i(v_x)) \) such that \( o_i \) is not the nearest object of \( v_y \) if one departs at time point \( t \).

We use \( B_i \) to denote the set of all the border vertices of \( C_i \). For every \( C_i, B_i \) is the set of \( mtt \)-functions \( m_{x,y}(t) \) for all vertices \( v_x \) in \( C_i \), that is,

\[
   D_i = \{ m_{x,y}(t) \mid v_x \text{ is a vertex in } C_i \}
\]

(13)

and \( M_i \) is a matrix of size \( |C_i| \times |B_i| \) to maintain \( mtt \)-function \( m_{x,y}(t) \) for all pairs of vertex \( v_x \) and border vertex \( v_y \) in \( C_i \), i.e.,

\[
   M_i = \{ m_{x,y}(t) \mid v_x \in C_i \land v_y \in B_i \}
\]

(14)

The voronoi-based index is \( [C, B, D, M] \), where \( C, B, D, M \) are the collections of all \( C_i, B_i, D_i, M_i \), respectively.

4.2. How to Construct The Voronoi-Based Index? We have explained how to compute \( mtt \)-function in Section 3. Next, we describe how to compute \( C_i \) and \( B_i \) for every \( o_i \in O \).

For every vertex \( v_x \in V \), \( I_i(v_x) \) is initialized as the whole time domain \( T \). We refine \( I_i(v_x) \) iteratively by removing the subintervals on which \( m_{x,y}(t) \) is larger than \( m_{x,y+1}(t) \) for another object \( o_j \). It means \( o_i \) is not the nearest object of \( v_x \) when departure time is in these subintervals. For every \( o_j \in O \), let \( T_j(v_x) \) denote the maximum time interval on which \( m_{x,y+1}(t) < m_{x,y}(t) \) and \( I_i(v_x) \) is updated as \( I_i(v_x)' = T_j(v_x) \). After removing \( T_j(v_x) \) for every object \( o_j \), if \( I_i(v_x) \) is not empty, then the pair \( (v_x, I_i(v_x)) \) is inserted into \( C_i \).

For every vertex \( v_x \in C_i \), if there exists an outgoing neighbor \( v_y \) of \( v_x \), such that \( v_y \) is not in \( C_i \) or \( f_{x,y}(I_i(v_x)) \notin I_i(v_y) \), then \( v_x \) must be a border vertex of \( C_i \) and it is inserted into \( B_i \).

4.3. Query Processing by Voronoi-Based Index. Algorithm 2 describes how to find the \( k \) nearest objects for a query vertex \( v_q \) with departure time \( t_d \). In Algorithm 2, \( O(v_q) \) is a set to
Input: time-dependent graph $G^*_t$, query vertex $v_q$, departure time $t_d$ and $k$

Output: the $k$ nearest neighbor set $O(v_q)$

1: $O(v_q) \leftarrow \emptyset$, $Q \leftarrow \{C_1\}$; $E_q \leftarrow \{v_q\}$
2: while $|O(v_q)| < k$ do
3: \quad $C_i \leftarrow \text{DEQUEUE}(Q)$, $O(v_q) \leftarrow O(v_q) \cup \{o_i\}$
4: \quad foreach $v_j \in B_i$ do
5: \quad \quad foreach $v_k \in E_i$ do
6: \quad \quad \quad $m_{q,y} \leftarrow \min(m_{q,x} + m_{x,y}(t_d + m_{q,z}))$;
7: \quad \quad \quad $O(v_q) \leftarrow \text{Enqueue}(Q, C_j)$;
8: \quad \quad \quad \text{while } |O(v_q)| < k do
9: \quad \quad \quad \quad $m_{q,z} \leftarrow \min(m_{q,z} + m_{y,z}(t_d + m_{q,y}))$;
10: \quad \quad \quad \quad \quad \text{if } m_{q,z} > m_{y,z} + \omega_{y,z}\star(t_d + m_{q,y}) \text{ then }
11: \quad \quad \quad \quad \quad \quad m_{q,z} \leftarrow m_{y,z} + \omega_{y,z}\star(t_d + m_{q,y})$;
12: \quad \quad \quad \quad \quad \text{if } C_j \notin Q \text{ then }
13: \quad \quad \quad \quad \quad \quad \text{Update}(Q, C_j)$;
14: \quad \quad \quad \quad \quad \text{end while}
15: \quad \quad \quad \quad \quad \text{ENQUEUE}(Q, C_j)$;
16: \quad \quad \quad \quad \text{end while}
17: \quad \quad \quad \text{end foreach}
18: \quad \quad \text{end foreach}
19: \quad \text{return } O(v_q)$

Algorithm 2: $k$NN-Query ($G^*_t$, $v_q$, $t_d$,$k$)

![Figure 2: Query processing.](image)

Example 7. We use the example in Figure 2 to illustrate the $k$NN querying process for $k = 3$. In this example, $v_q$ is the query vertex and it is in $C_1$ for the departure time $t_d$. Q and $O(v_q)$ are initialized as $\{C_1\}$ and $\emptyset$, respectively. In the first iteration, $C_1$ is dequeued from Q and then $o_1$ is inserted into $O(v_q)$. Because $v_1$ is a border vertex of $C_1$ and $v_1$ is an outgoing neighbor of $v_q$. Algorithm 2 computes $m_{q,y}$ as $\min(m_{q,x} + m_{x,y}(t_d + m_{q,z}))$ where $E_i$ is the entry set of $C_i$. The “entry” means any path entering into $C_i$ must go through a vertex in $E_i$. $E_i$ will be updated when Algorithm 2 runs. For every $v_j \in N^+(v_q)$, if $m_{q,z} > m_{q,y} + \omega_{y,z}\star(t_d + m_{q,x})$, then $m_{q,z}$ will be updated as $m_{q,z} + \omega_{y,z}\star(t_d + m_{q,x})$. Next, if $v_j$ is in $C_j$ ($C_j \notin C_i$ and $C_j \notin O(v_q)$) at the time point $t_d + m_{q,z}$, then $v_j$ will be inserted into $E_j$ as an entry of $C_j$. For the object $o_i$ of $C_j$, $m_{q,n_{i,j}}$ will be updated as $m_{q,z} + m_{n_{i,j}}(t_d + m_{q,z})$ when $m_{q,z} + m_{n_{i,j}}(t_d + m_{q,z})$. If $C_i$ is not in $Q$, then $C_i$ will be enqueued into $Q$. Otherwise, $C_i$ has been in $Q$ and will be updated by $C_i$ with new $m_{q,n_{i,j}}$. Algorithm 2 terminates when the size of $O(v_q)$ is $k$.

Theorem 8. In Algorithm 2, the object $o_i$ of $C_i$ dequeued from Q in the $k$-th iteration must be the $k$-th nearest object of query vertex $v_q$ for the departure time $t_d$.

Proof. We prove it by induction on $k$. 

Complexity
Basis. Obviously, \( C_q \) is dequeued from \( Q \) in the first iteration. By the definition of \( C_q \), \( o_q \) is the nearest object of \( v_q \) when the departure time is \( t_d \).

Induction. Assume that the \( i \)-th nearest neighbor of \( v_q \) is dequeued from \( Q \) in the \( i \)-th iteration for \( i < k \). We need to prove it also holds for \( i = k \). We prove it by contradiction. Let \( C_k \) be the closest pair set dequeued from \( Q \) in the \( k \)-th iteration and \( o_k \) is the object of \( C_k \). Suppose that the \( k \)-th nearest object of \( v_q \) is \( o_q \) and \( o_k \neq o_q \). Let \( p \) be the shortest path from \( v_q \) to \( o_q \) with the departure time \( t_d \). Because \( k > 1 \), then \( C_k \) is not \( C_q \) and there must exist an entry \( v_e \) of \( C_q \) in \( p \). Letting \( v_b \) be the predecessor of \( v_e \) in \( p \), then \( v_b \) must be a border vertex of \( C_k \) at time points \( t_d + m_{q,b} \) and \( b \neq o_k \). There are two cases for the object \( o_k \) of \( C_k \): (1) \( o_k \) is not in the \( k \) nearest object set of \( v_q \) and (2) \( o_k \) is in the \( k \) nearest object set of \( v_q \).

For case (1), by the definition of \( C_b \), \( o_b \) is the nearest neighbor of \( v_q \) at time point \( t_d + m_{q,b} \) and then we have

\[
m_{q,b} + m_{b,m} \leq t_d + m_{q,b} + m_{b,m+b} \leq t_d + m_{q,b} + m_{b,m+b}
\]

(15)

Thus \( o_k \) is nearer to \( v_q \) than \( o_b \) when the departure time is \( t_d \). It means \( o_k \) must be in the \( k \) nearest object set of \( v_q \) which is a contradiction.

For case (2), letting \( o_b \) be the \( i \)-th \((i < k)\) nearest object of \( v_q \) by the inductive assumption, \( C_b \) is dequeued from \( Q \) in \( i \)-th iteration. According to Algorithm 2, \( C_k \) is enqueued into \( Q \) in this iteration. Therefore, \( C_k \) will be dequeued from \( Q \) in \( k \)-th iteration instead of \( C_k \), which is a contradiction. The proof is completed.

Time and Space Complexities. The time and space complexities of Algorithm 2 are given below. Let \( b \) and \( e \) be the average size of \( B_i \) and \( E_i \), respectively. In every iteration, Algorithm 2 updates \( m_{q,y} \) as \( \min(m_{q,y}, m_{q,x} + m_{x,y}(t_d + m_{q,y})) \) for every border vertex \( v_y \) in \( C_i \). It will cost \( O(be) \) time. For every outgoing neighbor \( v_x \) of border vertex \( v_y \), Algorithm 2 needs to compute \( m_{xy} \) and then it will cost \( O(bd) \) time, where \( d \) is the average outdegree of the vertices in \( G_i \). Therefore, the time complexity of Algorithm 2 is \( O(kbd + e) \). On the other hand, because Algorithm 2 needs to maintain \( m_{q,y} \) and \( m_{q,z} \), then the space complexity is \( O(kb + e) \).

5. Optimization of Voronoi-Based Index

In Section 4, we introduce the voronoi-based index on time-dependent graphs and propose Algorithm 2 to find the \( k \) nearest objects with voronoi-based index. Algorithm 2 dequeues the objects iteratively until all \( k \) nearest objects have been searched. In each iteration, Algorithm 2 uses a prior queue \( Q \) to maintain a candidate set of \( C_i \), whose object \( o_j \) may be one of the \( k \) nearest objects. This candidate set is essentially the set of all the closest pair sets adjacent to all the objects which have been searched in \( O(v_j) \). However, for some closest pair set \( C_j \) of \( o_j \), even though \( o_j \) are not one of \( k \) nearest objects, Algorithm 2 also needs to maintain \( C_j \) in \( Q \) and update the traveling time to \( o_j \) and the border vertices in \( C_j \) because \( C_j \) is adjacent to an object \( o_j \) in \( O(v_j) \). It makes our algorithm not efficient enough for \( k \) nearest object query.

In this section, we propose a novel balanced tree structure on voronoi-based index, named V-tree, to organize all the closest pair sets \( C_j \). V-tree can be considered as a secondary level index for \( k \) nearest object query. By V-tree, our algorithm can avoid maintaining the closest pair sets \( C_j \), whose objects \( o_j \) are not in the set of \( k \) nearest objects. The main idea of V-tree comes from G-tree, which is a balanced tree structure, proposed in [14]. G-tree cannot be used for \( k \) nearest object query on time-dependent graphs because of the following reasons:

1. G-tree is built on the static graphs where the weight of every edge is a constant value. However, for time-dependent graphs, the weight on every edge is a time function \( w_{ij}(t) \) which is related to the departure time \( t \). The value of \( w_{ij}(t) \) always changes with the change of departure time. Therefore, it is unfeasible to construct G-tree for time-dependent graphs.

2. G-tree organizes all the vertices in a tree shape based on the distance of two vertices in a static graphs. Different from G-tree, V-tree is a second-level index based on voronoi-based index. It organizes all the closest pair sets \( C_i \) in a tree shape. Therefore, we need to define the distance between two closest pair sets \( C_i \) and \( C_j \) and then V-tree can be constructed.

3. Because of above two points, the querying algorithm by G-tree cannot work for our problem; then we propose a novel querying algorithm by V-tree.

In the following, we first introduce what is V-tree and how to construct it based on voronoi-based index. Then we propose a novel querying algorithm for \( k \) nearest object query by V-tree. Finally, we utilize an example to explain why V-tree is more efficient than voronoi-based index.

5.1. What Is V-Tree. V-tree is a balanced tree structure to organize all the closest pair sets \( C_i \) in voronoi-based index. Specifically, every node in V-tree is a collection of some closest pair sets. We first give the definition of \( \lambda \)-partition for voronoi-based index.

Definition 9 (\( \lambda \)-partition for voronoi-based index). Given a collection \( C \) of some closest pair sets, a \( \lambda \)-partition of \( C \) is a set \( \{C_1, \ldots, C_\lambda\} \) satisfying three following conditions: (1) \( \forall C_i, C_j \subseteq C \); (2) \( \forall C_i, C_j (i \neq j), C_i \cap C_j = \emptyset \); (3) \( C = \bigcup_{i=1}^\lambda C_i \).

Note that every \( C_i \) is a subset of \( C \) and it is also a collection of closest pair sets. Every node in V-tree represents a collection \( C \) and has \( \lambda \) children by \( \lambda \)-partition on \( C \); i.e., every \( C_i \) \((1 \leq i \leq \lambda)\) is a child of \( C \). Next, we introduce how to get an appropriate \( \lambda \)-partition for a given closest pair set \( C \). An edge \((v_x, v_y)\) is called a crossing edge from the closest pair sets \( C_i \) to \( C_j \) if \( v_x \) and \( v_y \) are the border vertices of \( C_i \) and \( C_j \), respectively, i.e., \( v_x \in B_i \) and \( v_y \in B_j \). As our discussion in Section 2, the time function of a crossing edge is modeled as
as a piecewise linear function $w_{x,y}(t)$. We define the expected traveling time $\overline{w}_{x,y}$ for a crossing edge $(v_x, v_y)$ by the following:

$$\overline{w}_{x,y} = \frac{1}{T} \int_{T} w_{x,y}(t) \, dt$$

where $T$ is the whole time interval for time function $w_{x,y}(t)$, i.e., $T = t_p - t_0$. Because $w_{x,y}(t)$ is a piecewise linear function, $\overline{w}_{x,y}$ can be calculated as follows:

$$\overline{w}_{x,y} = \frac{1}{t_p - t_0} \sum_{i=1}^{p} \int_{t_{i-1}}^{t_i} w_{x,y}(t) \, dt$$

$$= \frac{1}{t_p - t_0} \sum_{i=1}^{p} \left( \frac{t_i - t_{i-1}}{2} \right) \left( w_1(t_{i-1}) + w_2(t_i) \right)$$

Note that the expected traveling time for a crossing edge is a constant value to estimate the traveling time for all the crossing edges from $C_i$ to $C_j$. Given two closest pair set $C_i$ and $C_j$, we use $\overline{w}_{x,y}$ to denote the weight from $C_i$ to $C_j$. Thus, the optimal $\lambda$-partition {C_1, ..., C_λ} for a given collection of some closest pair sets can be converted into a graph partition problem. For a pair of $(C_i, C_j)$ with $E_{ij} \neq \emptyset$, if $C_i$ and $C_j$ are in the same collection $C_x$ ($1 \leq x \leq \lambda$), then it is called an inner edge of $C_x$. Otherwise, $C_i$ and $C_j$ are in different collections $C_x$ and $C_y$, respectively, and then it is called an interedge from $C_x$ to $C_y$. Correspondingly, the weight $a_{ij}$ is called inner weight or interweight, respectively.

In V-tree, a parent node $C$ is partitioned into $\lambda$ children nodes $|C_1|, ..., |C_\lambda|$. To facilitate $k$ nearest object query, an appropriate partition is expected such that the inner weights are less and the interweights are more. Thus the optimal $\lambda$-partition is to find a group of $|C_1|, ..., |C_\lambda|$ that minimizes the following objective function:

$$f(|C_1|, ..., |C_\lambda|) = \sum_{C_x} a(C_x)$$

where $a(C_x) = \sum_{C_x \subseteq C_x} a_{ij}$ is the summation of all the inner weights $a_{ij}$ in $C_x$ and $f(|C_1|, ..., |C_\lambda|)$ is the summation of $a(C_x)$ for all $C_x$. The optimal $\lambda$-partition on $C$ is similar to the traditional graph partition problem which has been well studied [15–17]. However, the objective function is to minimize the sum of the interweights for the traditional graph partition. Therefore, for any pair $(C_i, C_j)$ in $C$, if $E_{ij} \neq \emptyset$, we assign a new weight $b_{ij} = a_{\max} - a_{ij}$ on it, where $a_{\max}$ represents the maximum weight for all the inner edges in $C$. Consider the following objective function:

$$g(|C_1|, ..., |C_\lambda|) = \sum_{C_x} b(C_x, C_y)$$

where $b(C_x, C_y) = \sum_{C_x \subseteq C_x} \sum_{C_y} b_{ij}$ is the summation of $b_{ij}$ for all the inter edges from $C_x$ to $C_y$ and $g(|C_1|, ..., |C_\lambda|)$ is the summation of $b(C_x, C_y)$ for all the pairs of $C_x$ and $C_y$. The following theorem guarantees that the optimal $\lambda$-partition under objective functions $f(\cdot)$ and $g(\cdot)$ is the same.

**Theorem 10.** The $\lambda$-partition minimizing $f(\cdot)$ is exactly the $\lambda$-partition minimizing $g(\cdot)$.

**Proof.** Given a $\lambda$-partition $\{C_1, ..., C_\lambda\}$ on $C$, let $a(C_x, C_y)$ denote the summation of all the inter weights $a_{ij}$ from $C_x$ to $C_y$, that is, $a(C_x, C_y) = \sum_{C_x \subseteq C_x} a_{ij}$, and then we have

$$a(C) = \sum_{C_x \subseteq C_x} a(C_x, C_y) + \sum_{C_x} a(C_x)$$

Because $a(C)$ is independent of $\lambda$-partition on $C$, then minimizing $\sum_{C_x} a(C_x)$ is equivalent to maximizing $\sum_{C_x \subseteq C_x} a(C_x, C_y)$. On the other hand, because $b_{ij} = a_{\max} - a_{ij}$ and $b_{ij} \geq 0$, then maximizing $\sum_{C_x \subseteq C_x} a(C_x, C_y)$ is equivalent to minimizing $\sum_{C_x \subseteq C_x} b(C_x, C_y)$. Therefore, the $\lambda$-partition minimizing the objective function $f(\cdot)$ also minimizes the objective function $g(\cdot)$.

Thus the optimal $\lambda$-partition under objective function $g(\cdot)$ is a traditional graph partition problem. In this paper, we adopt the multilevel graph partitioning technique (METIS) proposed by Metis et al. in [15], which is a classic graph partition algorithm.

Next, we introduce the concept of the minimum distance, which is to be maintained in V-tree. For two closest pair set $C_i$ and $C_j$, if $E_{ij} \neq \emptyset$, we defined the minimum weight from $C_i$ to $C_j$, denoted as $w_{ij}^{\text{min}}$, by the following:

$$w_{ij}^{\text{min}} = \min_{(v_x, v_y) \in E_{ij}} \left\{ \min_{t} \{ w_{x,y}(t) \} \right\}$$

where $(v_x, v_y) \in E_{ij}$ is a crossing edge from $C_i$ to $C_j$ and $\min_{t} \{ w_{x,y}(t) \}$ is the minimum value of time function $w_{x,y}(t)$. Thus $w_{ij}^{\text{min}}$ is the minimum value of $\min_{t} \{ w_{x,y}(t) \}$ for all the crossing edge in $E_{ij}$. A new graph can be constructed from the voronoi-based index; every closest pair set $C_i$ can be regarded as a super vertex. If $E_{ij} \neq \emptyset$, there is an edge from $C_i$ to $C_j$ and its weight is $w_{ij}^{\text{min}}$. Therefore, for any two $C_i$ and $C_j$, the shortest distance from $C_i$ to $C_j$ in this graph is called the minimum distance from $C_i$ to $C_j$ and we denote it by $d_{ij}$. Especially, for a closest pair set $C_i$ and a collection $C$, the minimum distance from $C_i$ to $C$ is denoted by $d(C_i, C)$ and is the minimum value of $d_{ij}$ for all $C_j$ in $C$, i.e., $d(C_i, C) = \min_{C_j \subseteq C} d_{ij}$. The value of $d(C_i, C)$ can be calculated easily by existing single-source shortest distance algorithm.
\(d(C_i, C) = 0\) when \(C_i \in C\). Next, we give the definition of \(V\)-tree below.

**Definition II (V-tree).** A \(V\)-tree, denoted as \(T_V\), is a balanced tree \(T_V\) that satisfies the following conditions:

1. Every node in \(V\)-tree represents a collection \(C\) of some closest pair sets. Specially, the root node represents the collection \(C_{\text{root}}\) of all the closest pair sets in voronoi-based index;

2. Every nonleaf node \(C\) has \(\lambda\) children \(\{C_1, \ldots, C_\lambda\}\). All these children consist of the optimal \(\lambda\)-partition on \(C\).

3. Every leaf node \(C\) has at most \(\sigma\) closest pair sets.

In \(V\)-tree, every leaf node \(C\) is associated with a matrix to maintain the minimum distance from every closest pair set \(C_i\) in leaf node \(C\) to every node \(C'\) in \(V\)-tree \(T_V\), i.e.,

\[
\left\{d(C_i, C') \mid C_i \in C \land C' \in T_V\right\},
\]

where \(C'\) and \(C\) are different nodes in \(T_V\). It is because \(C_i \in C\) and then \(d(C_i, C) = 0\). Note that if \(C'\) is an ancestor \(C\) in \(T_V\), we also have \(d(C, C') = 0\) directly.

**Example 12.** Figure 3 illustrates an example of 2-partition on a voronoi-based index. The corresponding \(V\)-tree is shown in Figure 4. In this example, \(C_{\text{root}}\) is the collection of all the closest pair set and it is partitioned into \(\{C_1, C_2\}\). Thus \(C_1\) and \(C_2\) are two children of \(C_{\text{root}}\) in \(V\)-tree. Similarly, \(C_3\) and \(C_4\) are two children of \(C_1\), \(C_5\) and \(C_6\) are two children of \(C_2\). Because \(C_3, C_4, C_5,\) and \(C_6\) are four leaf nodes of \(V\)-tree, then
every one of them is associated with a minimum distance matrix. For example, $C_1$ is a closest pair set in $C_2$ and the minimum distance from $C_1$ to every node $C_i$ in $\mathcal{T}_V$ is stored in the matrix associated with $C_i$. In Figure 4, every matrix maintaining the minimum distance is presented next to every node in V-tree.

5.2. How to Construct V-Tree. Algorithm 3 describes how to construct a V-tree $\mathcal{T}_V$ for a given collection $C$ of all the closest pair sets in a voronoi-based index. To construct V-tree $\mathcal{T}_V$, Algorithm 3 utilizes a queue $Q$ to dequeue the nodes in $\mathcal{T}_V$ iteratively and expand the children for every node until V-tree has been constructed. Both $Q$ and $\mathcal{T}_V$ are initialized as $\{C_{root}\}$, where $C_{root}$ is the collection of all the closest pair set in voronoi-based index. In each iteration, a node $C$ is dequeued from $Q$. If $|C| > \sigma$, $C$ is a nonleaf node and then Algorithm 3 calls function PARTITION ($C$) to get the optimal $\lambda$-partition on $C$. Note that $|C|$ is the number of the closest pair sets in $C$ and PARTITION ($C$) is the implementation of METIS algorithm. Assume that the output of PARTITION ($C$) is $\{C_1, \ldots, C_k\}$, which will be inserted into $\mathcal{T}_V$ as the children set Children($C$) of $C$. Next, Algorithm 3 enqueues every child $C_i$ of $C$ into $Q$. Note that if $|C| \leq \sigma$, node $C$ is a leaf node in $\mathcal{T}_V$ and Algorithm 3 will dequeue next top element in $Q$. When $Q$ is empty, the structure of V-tree $\mathcal{T}_V$ has been constructed. Finally, for each leaf node $C$ in $\mathcal{T}_V$, Algorithm 3 calculates the minimum distance matrix $\{d(C_j, C'_i) \mid C_j \in C \land C'_i \in \mathcal{T}_V\}$ and associates it with the leaf node $C$. Therefore, V-tree is constructed and returned.

Example 13. We use the example in Figure 3 to show how to construct V-tree in Figure 4. In this example, $\lambda = 2$ and $\sigma = 3$. At the beginning, $Q$ and $\mathcal{T}_V$ are initialized as $\{C_{root}\}$; $C_{root}$ is the collection of all the closest pair set in Figure 3. In the first iteration, the root node $C_{root}$ is dequeued from $Q$, because $|C_{root}| = 11 \geq \sigma$ and $C_{root}$ is partitioned into $C_1$ and $C_2$. Then $C_1$ and $C_2$ are inserted into $\mathcal{T}_V$ as two children of $C_{root}$. Next, $C_1$ and $C_2$ are enqueued into $Q$. In second and third iteration, $C_1$ and $C_2$ are dequeued from $Q$, respectively. $\{C_1, C_2\}$ and $\{C_3, C_4\}$ are inserted into $\mathcal{T}_V$ and enqueued into $Q$. Algorithm 3 dequeues $C_3, C_4, C_5$, and $C_6$ one by one. Finally, the minimum distance matrices on $C_1, C_2, C_4$, and $C_6$ are, respectively, computed and maintained in V-tree $\mathcal{T}_V$. Thus the V-tree shown in Figure 4 has been constructed.

Space Complexity of V-Tree. The space complexities of V-tree are given below. Let $h$ denote the height of a V-tree $\mathcal{T}_V$. Obviously, $O(h) = O(\log_\lambda(|O|/\sigma) + 1)$, where $|O|$ is the number of objects in $\mathcal{T}_V$. The $i$-th level of $\mathcal{T}_V$ there are at most $\lambda^i$ nodes. Thus the number of nodes in a V-tree $\mathcal{T}_V$ is $O \left( \frac{1 - \lambda \log_\lambda(|O|/\sigma) + 1}{1 - \lambda} \right) = O \left( \frac{\lambda(|O|/\sigma) - 1}{\lambda - 1} \right)$ (24)

For each leaf node $C$ in $\mathcal{T}_V$, a matrix is used to maintain the minimum distance from every closest pair set $C_i \in C$ to every node $C' \in \mathcal{T}_V$. The size of every matrix is $O(\sigma \times |O|/\sigma) = O(|O|)$. Because there are at most $O(|O|/\sigma)$ leaf nodes in $\mathcal{T}_V$, then the space complexity of $\mathcal{T}_V$ is $O(|O|^2/\sigma + |O|/\sigma) = O(|O|^2/\sigma)$.

For the time complexity of V-tree construction, Algorithm 3 needs to compute $\lambda$-partition using existing partition algorithm for every nonleaf nodes in $\mathcal{T}_V$. We use $O(P)$ to denote the time complexity of $\lambda$-partition on all the closest pair sets in voronoi-based index. Therefore, constructing the structure of $\mathcal{T}_V$ needs $O(|O|/\sigma \times P)$ time. For every leaf node $C$ in $\mathcal{T}_V$, the minimum distance needs to be calculated. The minimum distances from one $C_i \in C$ to all other $C_j$ can be calculated by single-source and multidestination shortest path algorithm, which can be done in $O(|O|^2)$ time. Then the time complexity for every matrix on leaf node is $O(|O|^2)$. Therefore, the time complexity of V-tree construction is $O(|O|/\sigma \times P + |O|/\sigma \times |O|) = O(P + |O|/\sigma + |O|^3)$.

5.3. Query Processing with V-Tree. In this section, we propose an efficient algorithm for searching $k$ nearest objects by V-tree. The pseudocode is shown in Algorithm 4. The same as
Algorithm 4: kNN-Query \((v_q,t_d,k)\).

For case (2), \(C\) is dequeued from \(R\). If \(C\) is a non-leaf node in \(V\)-tree, the children \(C_i\) \((1 \leq x \leq \lambda)\) of \(C\) will be inserted into \(R\) with the minimum distance \(d(C_q,C_i)\). Note that \(d(C_q,C_i)\) is maintained in the leaf node \(C_q\) containing \(C_i\) in \(V\)-tree. If \(C\) is a leaf node in \(V\)-tree, every closest pair set \(C_i\) in \(C\) will be enqueued into \(Q\) with \(m_{q,n+i}\). Algorithm 4 calculates \(m_{q,n+i}\) by the following:

\[
\min_{v_q \in B(O(v_q)), v_j \in C, (v_q,v_j) \in E} \left\{ m_{q,j} + w_{x,y} (t_d + m_{q,x}) \right\} + m_{y,n+i} \tag{25}
\]

Note that the process to calculate \(m_{q,n+i}\) is as the same as that in Algorithm 2.

Algorithm 4 terminates when the size \(O(v_q)\) is \(k\). It means that \(k\) nearest objects of query vertex \(v_q\) have been found.

Theorem 14 guarantees the correctness of Algorithm 4.

**Theorem 14.** In Algorithm 4, the object node \(o_i\) inserted into \(O(v_q)\) at the \(k\)-th time must be the \(k\)-th nearest object of query vertex \(v_q\) for the departure time \(t_d\).

**Proof.** Similar to the proof of Theorem 8, we prove it by induction on \(k\).

**Basis.** Obviously, the object node \(o_q\) in \(C_q\) is the first object node inserted into \(O(v_q)\). By the definition of \(C_q\) and \(o_q\), we know that \(o_q\) is indeed the nearest object of \(v_q\) for the departure time \(t_d\).

**Induction.** Assume that the \(i\)-th nearest neighbor of \(v_q\) is inserted into the result set \(O(v_q)\) at the \(i\)-th time for \(i < k\).
We need to prove it also hold for $i = k$. We prove it by contradiction. It can be known that the object node inserted into $O(v_q)$ at the $i$-th time is the object node that dequeued from the priority queue $Q$ at the $i$-th time. Let $C_k$ be the closest pair set dequeued from $Q$ at the $k$-th time. Suppose that the $k$-th nearest object of $v_q$ is $o_{k'}$ and $o_{k'} \neq o_k$. Let $p$ be the shortest path from $v_q$ to $o_{k'}$ with the departure time $t_p$. Because $k > 1$, $C_k$ is not $C_q$ and there must exist an entry $v_c$ of $C_k'$ in $p$. Let $v_b$ be the predecessor of $v_c$ in $p$; then $v_b$ must be a border vertex of $C_i$ at time points $t_k + m_{gb}$ and $C_b \neq C_k'$. Obviously, for the object node $o_b$ of $C_b$, $(1) o_b$ is not in the $k$-nearest object set of $v_q$ and $(2) o_b$ is in the $k$-nearest object set of $v_q$.

For case (1), it is similar to this case in Theorem 8;

For case (2), it is assumed that $o_b$ is the $i$-th ($i < k$) nearest object of $v_q$. According to Algorithm 4, we know that $B(O(v_q))$ will be updated, when $o_b$ is inserted into $O(v_q)$. At this point, $m_{gb}$ will be updated by $B(O(v_q))$ if $C_k'$ is in $Q$. Otherwise, $m_{gb}$ will be updated by $B(O(v_q))$ when $C_k'$ is enqueued in $Q$. So $C_k'$ will be dequeued from $Q$ at $k$-th time with an accurate $m_{gb}$, which is a contradiction. The proof is completed.

**Time and Space Complexities.** We first analyze the time complexity of Algorithm 4. For processing priority queue $Q$, the time complexity is the same as Algorithm 2 in the worst case. For processing priority queue $R$, all nodes in $T_V$ will be dequeued in the worst case; then the time complexity is $O(|O|/\sigma)$. Therefore, the time complexity of Algorithm 4 is $O(k(b + e) + |O|/\sigma)$. Next, we analyze the space complexity, the same as Algorithm 2; the space complexity of Algorithm 4 is $O(k(b + e))$.

5.4. Effectiveness of V-Tree. We use the example in Figure 3 to show Algorithm 4 with V-tree being more efficient than Algorithm 2 without V-tree. For this example, we set $k = 3$, $\lambda = 2$, and $\sigma = 3$. The red five-pointed star represents the query vertex $v_q$ which is in $C_1$ for the departure time $t_q$. The minimum distance $m_{gb}$ from $v_q$ to every object $o_i$ is shown in the table in Figure 3.

For Algorithm 2, $Q$ is initialized as $\{C_1\}$. In the first iteration, $C_1$ is dequeued from $Q$. Because $C_2$, $C_4$, and $C_9$ are three neighboring closest pair set of $C_1$, Algorithm 2 will update their $m_{gb}$ and then enqueue them into $Q$. Similarly, in the second iteration, $C_2$ is dequeued from $Q$. $C_3$ and $C_4$ are enqueued into $Q$. $C_3$ will be dequeued from $Q$ in the final iteration. Note that $m_{gb}$ for the object in $C_2$ will be updated twice in the first and second iteration; respectively; this is because $C_2$ is the neighboring closest pair set of both $C_1$ and $C_4$. Therefore, Algorithm 2 needs to update $m_{gb}$ for five times in the whole querying process.

For Algorithm 4 by V-tree, $R$ and $Q$ are initialized as $\{C_{root}\}$ and $\emptyset$, respectively. When $C_4$ is dequeued from $R$, $C_6$, $C_2$, and $C_3$ are enqueued into $Q$. Note that $B(O(v_q)) = \emptyset$; then Algorithm 4 does not need to update $m_{gb}$ for $C_1$, $C_2$, and $C_3$ in the following iterations, Algorithm 4 will dequeue $C_1$, $C_2$, and $C_3$ from $Q$ one by one. This is because for the top element $C_1$ in $R$, $d(C_1)$ is $7$ is always larger than $m_{gb}$ for $C_1$, $C_2$, and $C_3$ which are the top elements in $Q$. When $C_1$ is dequeued from $Q$, $m_{gb}$ is updated and $C_3$ is enqueued into $Q$. Similarly, $m_{gb}$ will be updated and $C_3$ will be enqueued into $Q$. When $C_3$ is dequeued from $Q$, Algorithm 4 terminates and it only needs to update $m_{gb}$ twice in the whole querying process, which is obviously more efficient than Algorithm 2.

Next, we give the following theorem to guarantee that the Algorithm 4 with V-tree is at least as good as Algorithm 2.

**Theorem 15.** The number of updating $m_{gb}$ for all the closest pair sets in Algorithm 4 is no more than that in Algorithm 2.

**Proof.** In each iteration, for Algorithm 2, when $C_2$ is dequeued, all $m_{gb}$ for its neighboring $C_i$ are needed to be updated. Obviously, for Algorithm 4, it only needs to update $m_{gb}$ for a small number of $C_i$ in $Q$ which are a subset of the closest pair sets to be updated in Algorithm 2. So the number of updating $m_{gb}$ for all the closest pair sets in Algorithm 4 is no more than that in Algorithm 2.

6. Experiments

We compare our V-tree method (marked as VT) and voronoi-based index method (marked as VI) with FTTI (Fast-Travel-Time Index) method [11] and TLNI (Tight-and-Loose-Network Index) method [9] on the real-life datasets. FTTI and TLNI are the state of the art index-based methods for kNN query over time-dependent road networks. Note that FTTI and TLNI are used on $G_t^\tau$, in which every edge is an nwt-function $w_{ij}(t)$ because FTTI and TLNI do not allow the waiting time. Although some algorithms are proposed in recent works [5, 10], they are only to find the nearest object (i.e., $k = 1$) and they cannot be used for general kNN query on time-dependent graphs. All the experiments are conducted on a 2.6GHz Intel Core i7 CPU PC with the 16GB main memory, running on Windows 7.

6.1. DataSets and Experiment Setup. We tested the voronoi-based index method on California road network (CARN) (http://snap.stanford.edu/data/roadNet-CA.html) with 196,5206 vertices and 553,3214 edges. We extracted five time-dependent graphs with different size using the CARN database. The number of vertices ranges from 100k to 500k. The time domain is set as $T = [0, 2000]$; i.e., the departure time $t$ can be selected from $[0, 2000]$ for any vertex. Here, 2000 means 2000 time units. For every $w_{ij}(t)$, we split the time domain $T$ to $p$ subintervals and assign a linear function randomly for every subinterval and then $w_{ij}(t)$ is a piecewise linear function.

6.2. Experimentnal Results

**Exp-1. Impact of Network Size.** In this group of experiments, we study the impact of time-dependent network size. The number of the vertices increases from 100k to 500k and the number of objects is fixed at 10k. We investigate the querying time for $k = 7$. The number of piecewise intervals of $w_{ij}(t)$ is set as 4. As shown in Figures 5(a) and 5(b), the querying
time of our method is always less than FTTI and TLNI. Note that our V-tree method has the minimum querying time. Specifically, the querying time of TLNI is always much more than our methods even though TLNI has the smallest index size. The reason TLNI index only maintains the vertices for an object $o_i$ is that the upper bound of travel time to $o_i$ is less than the lower bound to the other objects. It cannot facilitate query effectively in large networks.

Exp-2. Impact of Object Set Size. In this group of experiments, the number of the vertices is fixed at 100k and the number of objects ranges from 10k to 50k. As shown in Figures 6(a) and 6(b), the querying time of our methods is always less than FTTI and TLNI. The querying time decreases with the increasing of the object set size. There are two reasons as follows: (1) the average size of $C_i$ and $B_i$ decreases if the object set size increases; (2) the increasing of object size results in that the objects become nearer to $v_q$ and then querying time decreases. The index size for VI, FTTI, and TLNI decreases with increasing of the object size. But for V-tree, the index size increases with the object size increasing. This is because more objects result in more nodes in V-tree.

Exp-3. Impact of the Time Domain. In Figure 7, we study the impact of time domain. In this group of experiments, the number of vertices and objects is fixed at 100k and 10k, respectively. The time domain ranges from $[0, 1000]$ to $[0, 3000]$. We investigate the querying time for $k = 7$. As shown in Figures 7(a) and 7(b), the querying time and index size of our methods are not affected by the expanding of time domain. However, for FTTI and TLNI, the querying time increases with the the expanding of time domain. This is because they need to maintain the estimated value about travel time in index to facilitate $k$NN query. If the time
domain becomes larger, the deviation between the estimation and the actual travel time will become larger too. It cannot facilitate query effectively.

**Exp-4. Impact of the Number of Piecewise Intervals.** In Figure 8, we investigate the impact of the number of piecewise intervals of \( w_{i,j}(t) \). In this group of experiments, the number of piecewise intervals of \( w_{i,j}(t) \) increases from 2 to 10. The numbers of the vertices and objects are fixed at 100k and 10k, respectively. As shown in Figures 8(a) and 8(b), the querying time and index size always increase with the increasing of the number of piecewise intervals. The reason is that the more piecewise intervals of \( w_{i,j}(t) \), the more piecewise intervals of mtt-function and then the more border vertices maintained in the index.

**Exp-5. Impact of the Average Distance to Objects.** The impact of the average distance to objects is shown in Figure 9. In this group of experiments, the number of vertices is set as 100k and 500k, respectively. The number of objects is fixed at 10k and \( k = 7 \). The average distance to objects ranges from 4 to 12. Note that the distance means the hops from the query vertex to an object. As shown in Figures 9(a) and 9(b), the querying time increases marginally with the increasing of distance. Our methods are always better than FTTI and TLNI.

**Exp-6. Impact of \( k \).** In Figure 10, we study the querying time by varying \( k \) from 1 to 10 on two different networks with 10k vertices and 50k vertices, respectively. In this group of experiments, the number of objects is fixed at 10k and 50k for two different networks, respectively. As shown in Figures 10(a) and 10(b), the querying time always increases marginally with the increasing of \( k \) for our index method.

**Exp-7. Impact of \( \lambda \) and \( \sigma \).** In Figure 11, we investigate the impact of the \( \lambda \) and \( \sigma \) for V-tree. In this group of experiments, the numbers of the vertices and objects are fixed at 100k and 10k, respectively. The values of \( \lambda \) and \( \sigma \) are selected from [2, 4, 6, 8, 10] and [10, 20, 30, 40], respectively. The experimental results show the effectiveness of V-tree is the best when \( \lambda = 4 \). The querying time increases with the increasing of \( \sigma \). But the index size increases with the decreasing of \( \sigma \). The index size is not affected by \( \lambda \) varying.

7. Related Work

KNN query has been well studied on static road networks. Most of the existing works propose various index techniques. The main ideas of these methods are to partition the vertices into several clusters, and then the clusters are organized as a voronoi diagram or a tree (e.g., R-tree) [14, 18–25]. These methods precompute and maintain the shortest distances for some pairs of vertices to facilitate kNN query. Unfortunately, these index techniques cannot be used for the time-dependent road networks because the minimum travel time between two vertices always varies with time. kNN query has also been studied on time-dependent road networks [9–12, 26]. Most of these works are based on A* algorithm. The authors in [5, 10] study the problem to find nearest (i.e., \( k = 1 \)) object on time-dependent networks. In [10], a virtual node \( v \) is inserted into the graph \( G \) with the zero-cost edges connecting to all the objects. The nearest object can be found on the shortest path from the query vertex to \( v \). The authors in [27] study the problem of finding \( k \) POIs that minimize the aggregated travel time from a set of query points. The index-based methods are proposed in [9, 11]. In [9], a * algorithm is utilized to expand the road networks by estimating an upper or lower bound of travel time. An index is built to facilitate kNN query using these estimated bounds. In [11], time domain is divided to several subintervals. For every subinterval, \( C \) nearest objects of every vertex are found by an estimation of minimum travel time. There are two main drawbacks of these methods. First, in these works, the FIFO (first in first out) property is required for networks and waiting time is not allowed. Second, the indexes proposed by
Figure 8: Impact of the number of piecewise interval of time function.

Figure 9: Impact of the average distance to objects.

Figure 10: Impact of $k$. 
these works are based on the estimated value of travel time. However, these indexes cannot facilitate query effectively for the large networks because the deviations are always too large between the estimated and the actual travel time.

Recently, there are some works about the shortest path query between two given vertices over time-dependent graphs [13, 28–33]. However, these works do not study any index that can be used in kNN query over time-dependent road networks. Note that, in [32, 33], the waiting time is incorporated in arrival function $g_i(t)$ which represents the earliest arrival time for every vertex $v_i$ when the departure time is $t$. Different to these works, we propose nwt-function $w^*_{ij}(t)$ to represent the minimum traveling time for every edge when the waiting time is allowed. The arrival function $g_i(t)$ cannot be utilized in our voronoi-based index for k nearest objects problem. Otherwise, the intuitive meaning behind $w^*_{ij}(t)$ is more easily comprehended than $g_i(t)$.

8. Conclusion

In this paper, we study the problem of $k$ nearest object query on time-dependent road networks. We first give an algorithm for processing time-dependent road networks such that the waiting time is not necessary to be considered and then propose a novel voronoi-based index to facilitate kNN query. We explain how to construct the index and complete the querying process using our index. Furthermore, we propose a novel balanced tree, named V-tree, which is a secondary level index on voronoi-based index to make our querying algorithm more efficient. We confirm the efficiency of our method through extensive experiments on real-life datasets.

Data Availability

The road network dataset used to support the findings of this study is included within the article. It can be downloaded from http://snap.stanford.edu/data/roadNet-CA.html.

Additional Points

Different to our previous work published on APWeb-WAIM 2018, in this paper, we propose a novel balanced tree structure, named V-tree, which can be considered as a secondary level index on the voronoi-based index to make k nearest object query more efficient.

Disclosure

An early version of this work has been published in the 2nd Asia-Pacific Web (APWeb) and Web-Age Information Management (WAIM) Joint International Conference on Web and Big Data (APWeb-WAIM 2018).

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work is supported by the National Natural Science Foundation of China Grant nos. 61402323, 61572353, and U1736103, the Opening Project of State Key Laboratory of Digital Publishing Technology, and the Australian Research Council Discovery Grant DP130103051.

References

[1] T. Abeywickrama, M. A. Cheema, and D. Taniar, "k-nearest neighbors on road networks: A journey in experimentation and in memory implementation," in Proceedings of the 42nd International Conference on Very Large Data Bases, VLDB 2016, vol. 9, no. 6, pp. 492–503, September 2016.

[2] K. C. Lee, W. Lee, and B. Zheng, "Fast object search on road networks," in Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology, p. 1018, Saint Petersburg, Russia, March 2009.
[3] S. Luo, B. Kao, G. Li, J. Hu, R. Cheng, and Y. Zheng, “TOAIN, a throughput optimizing adaptive index for answering dynamic knn queries on road networks,” The Proceedings of the VLDB Endowment (PVLDB), vol. 11, no. 5, pp. 594–606, 2018.

[4] F. Wei-Kleiner, “Finding nearest neighbors in road networks: a tree decomposition method,” in Proceedings of the the Joint EDBT/ICDT 2013 Workshops, pp. 233–240, Genoa, Italy, March 2013.

[5] M. R. R. B. Chucr, S. M. do Nascimento, J. A. de Macêdo et al., “Taxi, please! a nearest neighbor query in time-dependent road networks,” in Proceedings of the 2016 17th IEEE International Conference on Mobile Data Management (MDM), pp. 180–185, June 2016.

[6] B. George and S. Shekhar, “Time-aggregated graphs for modeling spatio-temporal networks,” Journal on Data Semantics, vol. 11, pp. 191–212, 2006.

[7] E. Kanoulas, Y. Du, T. Xia, and D. Zhang, “Finding fastest paths on a road network with speed patterns,” in Proceedings of the 22nd International Conference on Data Engineering, p. 10, April 2006.

[8] Y. Li, H. Caö, and Y. Tan, “Novel method of identifying time series based on network graphs,” Complexity, vol. 17, no. 1, pp. 13–34, 2011.

[9] U. Demiryurek, F. Banaei-Kashani, and C. Shahabi, “Efficient K-nearest neighbor search in time-dependent spatial networks,” in Database and Expert Systems Applications, vol. 6261 of Lecture Notes in Computer Science, pp. 432–449, Springer, Berlin, Germany, 2010.

[10] L. A. Cruz, F. Lettich, L. S. Júnior, R. P. Magalhães, and J. A. F. de Macêdo, “Finding the nearest service provider on time-dependent road networks,” in Proceedings of the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery, pp. 21–31, 2017.

[11] Y. Komai, D. H. Nguyen, T. Haru, and S. Nishio, “KNNS search utilizing index of the minimum road travel time in time-dependent road networks,” in Proceedings of the 2014 IEEE 33rd International Symposium on Reliable Distributed Systems Workshops, SRDSW 2014, pp. 131–137, October 2014.

[12] L. A. Cruz, M. A. Nascimento, and J. A. F. de Macêdo, “k-nearest neighbors queries in time-dependent road networks,” JDIM, vol. 3, no. 3, pp. 211–226, 2012.

[13] B. Ding, J. X. Yu, and L. Qin, “Finding time-dependent shortest paths over large graphs,” in Proceedings of the 11th International Conference on Extending Database Technology: Advances in Database Technology, pp. 205–216, Nantes, France, March 2008.

[14] R. Zhong, G. Li, K. Tan, and L. Zhou, “G-tree: an efficient index for KNNS search on road networks,” in Proceedings of the 22nd ACM international conference on Information & Knowledge Management, pp. 39–48, San Francisco, Calif, USA, October 2013.

[15] A. Abou-Rjeili and G. Karypis, “Multilevel algorithms for partitioning power-law graphs,” in Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2006, April 2006.

[16] I. S. Dhillon, Y. Guan, and B. Kulis, “Weighted graph cuts without eigenvectors a multilevel approach,” IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 29, no. 11, pp. 1944–1957, 2007.

[17] X. Xu, N. Yuruk, Z. Feng, and T. A. Schweiger, “SCAN: a structural clustering algorithm for networks,” in Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery And Data Mining, pp. 824–833, San Jose, Calif, USA, August 2007.

[18] H. Hu, D. L. Lee, and J. Xu, “Fast nearest neighbor search on road networks,” in Advances in Database Technology - EDBT 2006, vol. 3896 of Lecture Notes in Computer Science, pp. 186–203, Springer, Berlin, Germany, 2006.

[19] X. Huang, C. S. Jensen, and S. S. Ďenovík, “The islands approach to nearest neighbor querying in spatial networks,” in Advances in Spatial and Temporal Databases, vol. 3633 of Lecture Notes in Computer Science, pp. 73–90, Springer, Berlin, Germany, 2005.

[20] M. R. Kohl K保洁uan and C. Shahabi, “Voronoi-based K nearest neighbor search for spatial network databases,” in Proceedings of the Thirtieth International Conference on Very Large Data Bases, pp. 840–851, 2004.

[21] C.-L. Li, E. T. Wang, G.-J. Huang, and A. L. P. Chen, “Top-n query processing in spatial databases considering bi-chromatic reverse k-nearest neighbors,” Information Systems, vol. 42, pp. 123–138, 2014.

[22] Y. Liu, Z. Li, and X. Zhen, “Empirical study on indicators selection model based on nonparametric k-nearest neighbor identification and R clustering analysis,” Complexity, vol. 2018, Article ID 2067065, 9 pages, 2018.

[23] Y. Yang, M. A. Cheema, X. Lin, Y. Zhang, and W. Zhang, “Reverse k nearest neighbors queries and spatial reverse top-k queries,” The VLDB Journal, vol. 26, no. 2, pp. 151–176, 2017.

[24] Y. Zheng, Q. Guo, A. K. H. Tung, and S. Wu, “Lazysh: approximate nearest neighbor search for multiple distance functions with a single index,” in Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data, SIGMOD 2016, pp. 2023–2037, July 2016.

[25] H. Zhu, X. Yang, B. Wang, and W.-C. Lee, “Range-based obstructed nearest neighbor queries,” in Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data, SIGMOD 2016, pp. 2053–2068, July 2016.

[26] C. F. Costa, M. A. Nascimento, J. A. Macedo, and J. Machado, “A*-based solutions for KNNS queries with operating time constraints in time-dependent road networks,” in Proceedings of the 2014 15th IEEE International Conference on Mobile Data Management (MDM), pp. 23–32, Brisbane, Australia, July 2014.

[27] C. F. Costa, J. Machado, M. A. Nascimento, and J. A. Macêdo, “Aggregate k-nearest neighbors queries in time-dependent road networks,” in Proceedings of the the 4th ACM SIGSPATIAL International Workshop, pp. 3–12, November 2015.

[28] J. Kim, W.-S. Han, J. Oh, S. Kim, and H. Yu, “Processing time-dependent shortest path queries without pre-computed speed information on road networks,” Information Sciences, vol. 255, pp. 135–154, 2014.

[29] J. Ma, B. Yao, X. Gao, Y. Shen, and M. Guo, “Top-k critical vertices query on shortest path,” IEEE Transactions on Knowledge and Data Engineering, vol. 30, no. 10, pp. 1999–2012, 2018.

[30] Y. Yang, H. Gao, J. X. Yu, and J. Li, “Finding the cost-optimal path with time constraint over time-dependent graphs,” Proceedings of the VLDB Endowment, vol. 7, no. 9, pp. 673–684, 2014.

[31] D. Zhang, D. Yang, Y. Wang, K.-L. Tan, J. Cao, and H. T. Shen, “Distributed shortest path query processing on dynamic road networks,” The VLDB Journal, vol. 26, no. 3, pp. 399–419, 2017.

[32] L. Li, W. Hua, X. Du, and X. Zhou, “Minimal on-road time route scheduling on time-dependent graphs,” Proceedings of the VLDB Endowment, vol. 10, no. 11, pp. 1274–1285, 2017.
[33] L. Li, K. Zheng, S. Wang, W. Hua, and X. Zhou, "Go slow to go fast: minimal on-road time route scheduling with parking facilities using historical trajectory," *The VLDB Journal*, vol. 27, no. 3, pp. 321–345, 2018.
