Information Sciences 301 (2015) 241–261

Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

Network Voronoi Diagram on uncertain objects for nearest neighbor queries

Guohui Li [a], Li Li [a], Jianjun Li [a,*], Yanhong Li [b]

[a] School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China
[b] Department of Computer Science, South-Central University for Nationalities, Wuhan, China

ARTICLE INFO

Article history:
Received 1 July 2013
Received in revised form 30 October 2014
Accepted 31 December 2014
Available online 8 January 2015

Keywords:
Nearest neighbor
Uncertain data
Road networks
Voronoi diagram

ABSTRACT

In the past decade, probabilistic nearest neighbor (PNN) query processing has received significant research attention due to the development of mobile smart terminals and the advances in wireless communication technologies. However, to the best of our knowledge, most of the existing PNN-oriented studies are aimed at the Euclidean space and cannot be readily extended to road networks. This paper takes the first step toward processing PNN queries in road Networks (NPNN). We first present an efficient method to construct Network Voronoi Diagram on Uncertain objects (UNVD), in which we first find the possible NNs of all the vertices, and then compute the u-edges as well as their corresponding possible NNs. Next, to process NPNN queries, we first present a computational method to calculate the probabilities for each possible nearest object, and then propose two data structures, namely *gIndex* and *qIndex*, to index the u-edges in UNVD. Finally, we evaluate the performance of our NPNN query processing methods via extensive experiments on both real road networks and synthetic 2-dimensional grid networks. Experimental results demonstrate the effectiveness and efficiency of our methods in terms of I/O cost and query time.

1. Introduction

Thanks to the rapid growth in popularity of smart mobile terminals like mobile phones and the advances in wireless communication technologies, location-based service (LBS) has become popular over the past few years. Being one of the enabling technologies for LBS, location based queries (LBQs) have been a hot research topic. An important class of LBQs is the nearest neighbor (NN) query, which answers query based on where the query is issued, and returns the object that is closest to the query point. Generally speaking, existing NN studies can be classified into two types: NN in the Euclidean space and NN in the network space, both assume that the object locations are precise. However, in practice, the locations of objects can be uncertain due to various reasons, such as the impreciseness of positioning technologies, where errors may originate from the satellite, the receiver, or the signal propagation. Moreover, location uncertainty is desirable for better privacy preservation, especially when one needs to reveal personal location information to the public for some particular purposes, like research. In such cases, to reduce the risk of being identified at a particular site, a user's position should be represented

* Corresponding author. Tel.: +86 027 87543104.
E-mail address: jianjunli@hust.edu.cn (J. Li).

http://dx.doi.org/10.1016/j.ins.2014.12.050
0020-0255/

as a larger region rather than a single point [1,2]. All these factors have led to a flurry of activity [9,3] on query processing over uncertain data, which is also known as probabilistic nearest neighbor (PNN) query processing.

In PNN, due to location uncertainty, an object may have multiple nearest neighbors, each having the probability value of being the nearest neighbor. A common uncertainty model used in PNN is to assume an uncertain object o has an uncertainty region where o possibly resides and a probability distribution function (*pdf*) which describes the distribution of o's precise position within the uncertainty region. To our best knowledge, most of the existing work on PNN deals with objects embedded in Euclidean space, where the distance between any two points is determined by their coordinates (Euclidean distance). However, in real life situations, objects are usually limited to move in road networks such as the railway networks and the highway networks, where the distance between objects is determined by the connectivity of the network, rather than the objects' coordinates in Euclidean space. Hence, existing PNN processing methods cannot be applied in road networks directly, and it is therefore essential to design PNN query methods suitable for use in real road Networks (abbreviated as NPNN in the following).

As a well known space partitioning technique, the Voronoi diagram (VD) [36] has been widely applied for processing spatial queries, especially the Nearest-Neighbor queries [41,15,48,35]. A general VD partitions the space into disjoint regions so that every point in the same region has the same nearest neighbor. Fig. 1 illustrates an example of VD, where every point located in polygon *abcdef* takes p as its NN, and every point located in polygon *abcdef* is the reverse NN of p. Such good properties make VD a promising tool for spatial query processing. Specifically, for NN queries, by representing the solution space as VD, the queries can be reduced to and processed as simple point queries.

In general, VD is generated by a set of space objects, termed as generators. In terms of the types of generators (certain or uncertain object) and which space (Euclidean or network) is assumed, the family of VDs can be classified into four types: (1) VDs for certain objects in Euclidean space; (2) VDs for certain objects in networks; (3) VDs for uncertain objects in Euclidean space; and (4) VDs for uncertain objects in networks. The first three types have been well studied in previous works, as shown in Table 1. On the contrary, there is no work reported on studying the network VD based on uncertain objects (UNVD). Constructing UNVD is of great importance not only because it can be used to efficiently answer NPNN queries, but also because it has great potential to be extended to facilitate other spatial queries such as reverse and range NN queries. Moreover, although NVD has been used in previous studies [26,27] to support NN queries in road networks, the VD designed for certain objects are not appropriate to be applied for uncertain objects, since multiple objects are possible to be the NNs of a point in UNVD, while only one NN exists for a point in NVD.

In this paper, we focus on two major issues: constructing UNVD and processing NPNN by using the constructed UNVD, both are fresh attempts. In our UNVD-constructing method, we first compute the possible NNs of all the network vertices by proposing a novel algorithm MarkV. MarkV traverses the network vertices one by one and records the potential possible NNs of each vertex, until the real possible NNs are obtained. In this way, the entire network only needs to be scanned once. Then, by proposing another algorithm MarkE, we partition the network edges into u-edges and compute the possible NNs of every u-edge. All points on the u-edge have the same possible NNs. Depending on whether the two endpoints of an edge have the same possible NNs, MarkE directly computes the possible NNs of the edge, or further splits the edge and repeats the examination on the sub-edges derived. By executing MarkV and MarkE sequentially, the UNVD can be constructed efficiently. Based on the constructed UNVD, we first show how to calculate the probabilities for each possible nearest object, and then present two data structures, namely *gIndex* and *qIndex*, to index the UNVD. Specifically, *gIndex* is implemented as a hash table, while *qIndex* adopts a PMR quad tree framework. In both indexes, the entire UNVD space is partitioned into a number of grid cells, each corresponding to a set of linked disk pages. Moreover, since NPNN queries, especially continuous NPNN queries, usually search for u-edges that are closely located, the grid cells are arranged in the order of Hilbert curve in *gIndex*. Overall, the main contributions of this paper can be summarized as follows:

- We address the UNVD construction problem by first proposing a novel method MarkV to compute the possible NNs for all the vertices in a road network, and then introducing another method MarkE to partition the network edges until all points on each sub-edge have the same possible NNs. MarkV exhibits the location-uncertainty property of uncertain objects and requires to scan the entire network only once, and thus is of high efficiency.

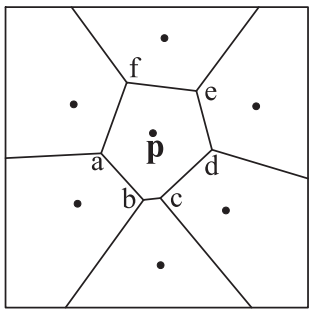


Fig. 1. An example of VD.

Table 1
Voronoi diagrams.

	Euclidean space	Network
Certain objects	Ordinary VD [36]	NVD [37]
Uncertain objects	UV-diagram [9]	UNVD

- To efficiently process static and continuous NPNN queries, we propose two data structures: *gIndex* and *qIndex*, to index the constructed UNVD. In particular, by utilizing Hilbert curve to organize the grid cells and allowing different buckets pointing to the same disk pages in *gIndex*, and adopting a PMR quad tree framework in *qIndex*, the two indexes enable efficient retrievals when dealing with static and continuous NPNN queries, respectively.
- We evaluate and comment on the performance of the proposed methods via extensive simulation experiments on both synthetic and real datasets. The experimental study shows that our NPNN processing methods are quite efficient in terms of I/O performance and query time.

The remainder of this paper is organized as follows. Section 2 reviews related work. Section 3 details our UNVD constructing method. Section 4 presents how to answer NPNN queries by using UNVD. Section 5 evaluates the performance of the proposed methods. Finally, Section 6 draws a conclusion.

2. Related work

The NN problem has been extensively studied during the last decade and is still an attractive topic currently [24,32,46,11,22]. In this section, we briefly review some existing work on NN search in Euclidean space and road networks with certain objects, and PNN search in Euclidean space with uncertain objects. We also discuss some existing work on Voronoi diagram.

2.1. NN search in Euclidean space and road networks

NN search in Euclidean space is the earliest type of the NN problem. Roussopoulos et al. [39] pioneered the work on NN processing. They provided a branch-and-bound R-tree traversal algorithm as well as metrics for ordering and pruning. However, it is difficult to achieve high efficiency in multi-user environments due to the depth-first fashion. Chen and Chin [6] overcomed this disadvantage by considering the concurrency feature. As an important variant of NN search, continuous nearest neighbor (CNN) queries in Euclidean space [47,45,13] has also been studied since the first solution was proposed by Song and Roussopoulos in [43].

NN processing in road networks has received significant attention ever since it is first addressed by Jensen et al. in [21]. Papadias et al. [38] used an architecture integrating the network and Euclidean information to answer static queries in networks. To avoid on-line distance computation, Kolahdouzan and Shahabi [26] studied the *k*NN processing problem based on the Network Voronoi Diagram. Huang et al. [18] used pre-computation and on-line network expansion to offer flexibility in balancing the amount of pre-computation, the updating of the pre-computed data and the query processing efficiency. An architecture called MOVNet which utilizes an on-disk R-tree to store the network structure information and an in-memory index to maintain object position updates was presented by Wang and Zimmermann in [44]. Two C*k*NN processing methods called IE and UBA were proposed by Kolahdouzan and Shahabi in [27]. In IE, *k*NN of all vertices on a given path are examined. Cho and Chung [10] combined the pre-computed *k*NN lists with Dijkstra algorithm to answer C*k*NN queries efficiently. A progressive incremental network expansion strategy which adopts a modified version of the IE algorithm was also adopted to address the CNN problems [40]. To answer concurrent C*k*NN, Mouratidis et al. [33] utilized an in-memory data structure to store network connectivity and an expansion strategy, while Huang et al. [19] utilized the pruning and refining techniques. Recently, continuous nearest neighbors in road networks on the air is studied in [30].

2.2. PNN in Euclidean space

As one kind of attribute uncertainty, location uncertainty is represented as a range of possible values and a *pdf* is bounded within the range [42]. Several algorithms have been proposed to process PNN over attribute uncertainty. Cheng et al. [8] utilized numerical integration techniques to address the PNN problems based on R-tree. Later, Cheng et al. [7] discussed the constrained NN problem, which returns objects whose probabilities are higher than some threshold. Kriegel et al. [28] computed the probabilities of answer objects based on sampling. Cheng et al. [9] answered PNN based on UV-diagram, which outperforms the R-tree based methods. Ali et al. [3] provided a pre-computation approach and an incremental approach to handle moving nearest neighbor queries on uncertain data in Euclidean space.

2.3. The Voronoi diagram

As an important technique for facilitating NN related queries, VD [36] has been extensively studied and many variants and construction algorithms have been proposed. The two most important variants related to our proposed UNVD are NVD [37] and UV-diagram [9]. Okabe et al. [37] constructed the Voronoi diagram in road networks based on objects having precise locations. Cheng et al. [9] constructed the Voronoi diagram in Euclidean space based on objects whose locations are imprecise. Evans and Sember [12] and Jooyandeh et al. [23] set similar assumptions to [9], but their goal is to find the set of points that are guaranteed to be closer to a particular site than to any others, while [9] returns objects that may have a chance to be the nearest neighbor of the query point.

3. UNVD

We consider a set of uncertain objects $O = \{o_1, o_2, \ldots\}$ in a road network which is modeled as a connected weighted graph $NG(V, E)$, where $V = \{v_1, v_2, \ldots\}$ and $E = \{e_1, e_2, \ldots\}$ are the sets of vertices and edges, respectively. We use $|O|$, $|V|$ and $|E|$ to denote the number of elements in O, V and E, respectively. The distance between two objects in NG is determined by the shortest-path distance. Specifically, the shortest-path distance between two points p_1 and p_2 in NG is denoted by $d(p_1, p_2)$. For simplicity, we assume the weight of an edge in NG is equal to the length of the edge, and any edge $e \in E$ can be represented by $\overline{v_s v_e}$ or $\overline{v_e v_s}$, where v_s and v_e are the two endpoints of e. We use $nn\_dom(p)$ to denote the set of possible NNs of a point p in NG, and use $nn\_dom(l)$ to denote the set of possible NNs of a line segment l in NG. Note here all the points on l have the same possible NNs. Since practical road networks (e.g., city street networks) generally have a fixed structure in a long period, which means the generator objects of UNVD are stationary uncertain objects, we do not consider the update of UNVD in this paper. Formal definitions of the symbols used in this paper are presented in Table 2.

In the following, we first briefly introduce some basic knowledge regarding UNVD in Section 3.1, and then present our UNVD construction method in Section 3.2.

3.1. Introduction of UNVD

Unlike previously studied VDs, UNVD is generated by uncertain objects and measured by shortest-path distances. An uncertain object is assumed to have an uncertain location, which is represented by an uncertainty region of possible locations (sub-graphs of NG) with *pdf*s assigned to the edges in the region.

Definition 1. The **uncertainty region** of uncertain object $o \in O$, denoted by $C(o)$, is defined as the area which includes all possible positions of o in NG. This means the precise position of o can only be located in $C(o)$. Unlike uncertainty region in a plane, $C(o)$ in NG is composed of several line segments.

Each uncertainty region has a *pdf* and a set of extreme points. The *pdf* describes the distribution of o's position within $C(o)$. In this paper, we assume that the complete *pdf* of o is composed of N (the number of line segments in $C(o)$) sub-functions, each of which is bounded in a line segment. The extreme point is defined as follows.

Definition 2. An **extreme point** of uncertain object $o \in O$, denoted by $E_i(o)$ $(i = 1, 2, \ldots, \lambda)$, is located on the boundary between the sub-graphs covered by $C(o)$ and the other parts of NG. Here λ is the total number of the extreme points in $C(o)$.

Table 2
Symbols and definitions.

Symbol	Definition
NG	A connected weighted graph modeling a road network
$O/V/E$	The set of uncertain objects/vertices/edges in NG
$\|O\|/\|V\|/\|E\|$	The number of uncertain objects/vertices/edges in NG
$d(p, q)$	The shortest-path distance between two points p and q
$\overline{v_s v_e}$	An edge where the endpoints are v_s and v_e
$nn\_dom(p)$	The set of possible NNs of a point p
$nn\_dom(l)$	The set of possible NNs of a line segment l, on which all the points have the same possible NNs
$C(o)$	The uncertainty region of uncertain object $o \in O$
pdf	The probability distribution function
$dmax(o, q)$	The possible maximal distance from $o \in O$ to a point q
$dmin(o, q)$	The possible minimal distance from $o \in O$ to a point q
$E_i(o)$	An extreme point of $o \in O$
λ	The total number of the extreme points in $C(o)$
$b_{(o_i, o_j)}$	A break point with respect to $o_i \in O$ and $o_j \in O$ $(o_i \neq o_j)$
$o : [a, b]$	A label allocated to vertices in V
LS_i	A label set allocated to vertex $v_i \in V$
α	The maximum number of the u-edges overlapped with a grid cell

As an illustration, Fig. 2(a) shows an example of $C(o)$ (depicted as bold lines), in which there are three extreme points $E_1(o)$, $E_2(o)$ and $E_3(o)$.

The UNVD built on *NG* can be regarded as an updated version of *NG*, where each edge is associated with a set of possible NNs. An edge in UNVD is referred to as a u-edge, which is a line segment corresponding to all or part of an original edge in *NG*. All the points in a u-edge have the same possible NNs. A UNVD is partitioned into a set of subnetworks, each specifying a part of *NG* on which any point has the chance to take one or more specified uncertain objects as its NNs. To sum up, we have the following definition regarding UNVD.

Definition 3. Given a set O of two or more but finite number of distinct uncertain objects in a road network, we associate all locations in the network space with the possible NN(s) with respect to the shortest-path distance. The result is a tessellation of the network into a set of the regions associated with subsets of O. This tessellation is called the **Network Voronoi Diagram on Uncertain objects** (**UNVD**) generated by O.

Fig. 2(b) gives an example of UNVD, which is generated by three uncertain objects o_1, o_2 and o_3. To clearly show the partitions, the u-edges are drawn with different types. Specifically, three types of line segments: densely dotted, solid and loosely dotted, are used to represent the sub-networks where o_1, o_2 and o_3 are taken as the possible NNs, respectively. Note in this example, any point on line segment $\overline{s_i s_j}$ ($i \in \{1,3,5,7,9\}$, $j = i+1$) has two possible NNs.

3.2. Construction of UNVD

In this section, we present our method for constructing UNVD. We first compute the possible NNs for each $v \in V$, and then compute the u-edges as well as their corresponding possible NNs. Due to location uncertainty, it is impossible to obtain a precise distance value $d(o,q)$ from an uncertain object o to a point q in *NG*. Hence, we use the *possible maximal/minimal distances*, which are defined in the following, to approximate the distances involving uncertain objects.

Definition 4. The **possible maximal distance** from an uncertain object o to a point q in *NG*, denoted by $dmax(o,q)$, is the longest shortest-path distance among the distances between q and any point r in $C(o)$, i.e., $dmax(o,q) = \max\{d(r,q), r \in C(o)\}$.

Definition 5. The **possible minimal distance** from an uncertain object o to a point q in *NG*, denoted by $dmin(o,q)$, is the shortest shortest-path distance among the distances between q and any point r in $C(o)$, i.e., $dmin(o,q) = \min\{d(r,q), r \in C(o)\}$.

Based on Definitions 4 and 5, we have the following observation:

Observation 1. For any uncertain object $o_j \in O$ and an arbitrary point q in *NG*, if there exists an uncertain object $o_k \in O (j \neq k)$ which satisfies $dmax(q,o_k) < dmin(q,o_j)$, then o_j is impossible to be the NN of q, i.e., $o_j \notin nn\_dom(q)$, and vice versa. In other words, for any uncertain object $o_k \in O (j \neq k)$, if $dmax(q,o_k) \geqslant dmin(q,o_j)$, then o_j is q's possible NN, i.e., $o_j \in nn\_dom(q)$, and vice versa.

3.2.1. Marking vertices

This section aims to compute $nn\_dom(v)$ for each $v \in V$. To achieve this goal, a straightforward method may work as follows: (1) utilize the shortest path algorithm (e.g., Dijkstra's algorithm) to find the shortest-path distances from each extreme point to each $v \in V$; (2) based on the obtained shortest-path distances, compute the possible maximal and minimal distances from each $o \in O$ to each $v \in V$; and (3) calculate the set of possible NNs $nn\_dom(v)$ of each $v \in V$ by comparing the possible maximal and minimal distances from each $o \in O$ to v.

The above method, though effective, is not cost-efficient due to that the Dijkstra's algorithm has to be executed many times, especially when $|O|$ is large. To address this problem, we propose a novel two-step method called MarkV, which only needs to scan *NG* once. Below we introduce the two steps of MarkV in detail.

Step 1. This step aims to compute the possible maximal and minimal distances from each uncertain object $o \in O$ to the vertices directly connected to it.

It is clear that the possible minimal distance $dmin(o,v)$ must be equal to one of the distances from $\{d(E_1(o),v), \ldots, d(E_\lambda(o),v)\}$. Hence, we mainly focus on computing the possible maximal distance $dmax(o,v)$ from o to v, and the challenge lies in how to compute it without scanning every point in $C(o)$. Suppose $e_1, e_2, \ldots, e_k$ are k line segments in $C(o)$, we have,

$$dmax(o,v) = \max_{1 \leqslant i \leqslant k} \{dmax(o,v)|e_i\} \tag{1}$$

where $dmax(o,v)|e_i$ denotes the possible maximal distance from o to v when o is located on e_i. Further, suppose v_s and v_e are the two endpoints of e_i and p is an arbitrary point on e_i, $dmax(o,v)|e_i$ can be computed by

$$dmax(o,v)|e_i = \min\{d(v_s,v) + d(v_s,p), d(v_e,v) + d(v_e,p)\} \tag{2}$$

Regarding Eq. (2), we have the following three cases:

- If $d(v_e, v) + d(v_e, v_s) - d(v_s, v) > 0$, then $dmax(o, v)|e_i = d(v_e, v)$;
- If $d(v_s, v) + d(v_e, v_s) - d(v_e, v) > 0$, then $dmax(o, v)|e_i = d(v_s, v)$;
- Otherwise, we find p which satisfies the following formula:

$$\begin{cases} d(v_s, v) + d(v_s, p) = d(v_e, v) + d(v_e, p) \\ d(v_s, p) + d(v_e, p) = d(v_s, v_e) \end{cases} \tag{3}$$

where $d(v_s, v) = \min_{1 \leqslant i \leqslant \lambda}\{d(v_s, E_i(o)) + d(E_i(o), v)\}$ and $d(v_e, v) = \min_{1 \leqslant i \leqslant \lambda}\{d(v_e, E_i(o)) + d(E_i(o), v)\}$, and then we can get that $dmax(o, v)|e_i = d(v_s, v) + d(v_s, p) = d(v_e, v) + d(v_e, p)$.

Step 2. This step is responsible for computing $nn\_dom(v)$ for each $v \in V$. To facilitate the computation, we allocate every vertex a set of labels, each of which is defined as follows.

Definition 6. A **label**, denoted by $o : [a, b]$, is composed of an uncertain object $o \in O$ and a distance range $[a, b]$, where a and b represent the upper and lower bounds of the range, respectively. For presentation convenience, we also use $\tilde{o}$ to represent $o : [a, b]$ in the following discussion.

For any two distance ranges $[a_i, b_i]$ and $[a_j, b_j]$, we say $[a_i, b_i] > [a_j, b_j]$, if and only if $a_i > b_j$, and say $[a_i, b_i] \approx [a_j, b_j]$, if and only if $[a_i, b_i]$ and $[a_j, b_j]$ overlap, i.e., $a_j < a_i \leqslant b_j$ or $a_i < a_j \leqslant b_i$. Based on this claim, we define the relationship between any two labels $o_i : [a_i, b_i]$ and $o_j : [a_j, b_j]$ as follows.

Definition 7. We say label $\tilde{o_i}$ is larger than label $\tilde{o_j}$, denoted by $\tilde{o_i} > \tilde{o_j}$, if and only if $[a_i, b_i] > [a_j, b_j]$. And, we say $\tilde{o_i} \approx \tilde{o_j}$, if and only if $[a_i, b_i] \approx [a_j, b_j]$.

By the above definition, we have $o_i : [2, 4] < o_j : [5, 6]$ and $o_i : [2, 4] \approx o_k : [3, 5]$. For each vertex $v_i \in V$, there is a **label set** $LS_i = \{\tilde{o_{i1}}, \tilde{o_{i2}}, \ldots\}$ associated with it. LS_i contains at most $|O|$ labels, and is initialized to be empty. Similar to Definition 7, we define the relationship between two label sets as follows.

Definition 8. For any two label sets LS_i and LS_j, we say $LS_i > LS_j$, if and only if any label in LS_i is larger than all the labels in LS_j, and say $LS_i < LS_j$, if and only if any label in LS_i is smaller than all the labels in LS_j. Otherwise, we say $LS_i \approx LS_j$.

The value of the labels would change during the execution of our algorithm. If the value of a label does not change any more, then it is considered to be permanent. Specifically, let $o : [a, b]$ be a label that is allocated to a vertex $v \in V$ at a time instant, if $o : [a, b]$ is permanent, we have $a = dmax(o, v)$ and $b = dmin(o, v)$. Otherwise, we have $a \geqslant dmax(o, v)$ and $b \geqslant dmin(o, v)$. An uncertain object contained in a permanent label allocated to vertex v is a possible NN of v. A label set consisting of all permanent labels is called a permanent label set, and a vertex which is allocated a permanent label set is termed as a visited vertex.

Algorithm 1. MarkV

Input: O, $NG(V, E)$
Output: $nn\_dom(v)$ for each $v \in V$

```
1: for each v directly connected to o ∈ O do
2:    Compute dmax(o, v) and dmin(o, v);    Step 1
3: end for
4: Initialize CUR ← O, VI ← ∅;        //Step 2
5: for each uncertain object o ∈ O do
6:    for each vertex v_i ∈ V in an uncertainty region C(o) do
7:       Insert a permanent label õ into LS_i;
8:       VI ← v_i;
9:    end for
10: end for
11: while VI ≠ O do
12:    CUR = RETR(NG, CUR);
13:    VI = EXAM(NG, CUR, VI);
14: end while
15: for each v_k ∈ VI do
16:    for each label õ ∈ LS_k do
17:       Insert o into nn_dom(v_k);
18:    end for
19: end for
```

Algorithm 1 shows the pseudo-code of MarkV. Step 1 is first executed to compute the possible maximal and minimal distances (lines 1–3). Let *VI* and *CUR* be the set of visited vertices and current vertices, respectively. Step 2 starts by initializing *CUR* and *VI* to be *O* and $\emptyset$, respectively (line 4 of Algorithm MarkV). Then, for each $o \in O$, a permanent label $\tilde{o}$ is allocated to every vertex located in $C(o)$, and the corresponding vertices are inserted into *VI* (lines 5–10). Next, two functions RETR and EXAM, which will be detailed later, are executed repeatedly, until all the vertices are visited (lines 11–14). Finally, for each $v_k \in VI$, we can obtain the set of possible NNs by collecting the uncertain objects reside in LS_k (lines 15–19). Now we detail the two functions RETR and EXAM, respectively.

Algorithm 2. RETR

```
Input: NG(V, E), CUR
Output: CUR
  for each v_j ∈ (V − VI) do
    for each v_i ∈ CUR do
      for each label õ ∈ LS_i do
        Compute õ + w_ij;
        Add õ + w_ij to LS_j;
      end for
    end for
  end for
  Select the smallest label set(s);
  Empty out CUR and insert the vertices corresponding to the smallest label set(s) into CUR;
  Return CUR;
```

Algorithm 3. EXAM

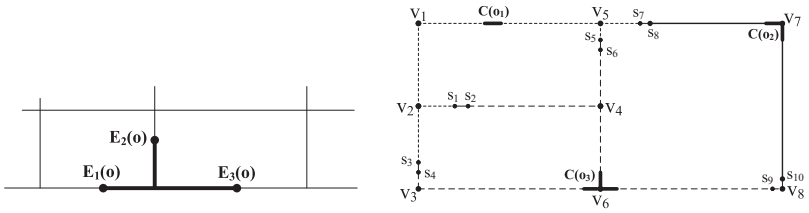
```
Input: NG(V, E), CUR, VI,
Output: VI
  for each vertex v_s ∈ CUR do
    Check (v_s, CUR);
  end for
  VI ← CUR;
  Return VI;
```

Algorithm 4. Check

```
Input: v_s, CUR
  for each vertex v_t ∈ CUR that is directly connected to v_s do
    for each label õ_i ∈ LS_s do
      Compute õ_i + w_st;
      Add õ_i + w_st to LS_t;
    end for
    if LS_t is changed then
      Check(v_t, CUR);
    end if
  end for
```

RETR (Algorithm 2) aims to select a set of new current vertices, and is implemented by sequentially executing the following two steps.

(1) For each unvisited vertex v_j and an arbitrary vertex v_i in *CUR*, we compute new labels for v_j and add them to the label set LS_j (lines 1–8). More specifically, for each label $\tilde{o} \in LS_i$, we compute a new label $\tilde{o} + w_{ij} = o : [a + w_{ij}, b + w_{ij}]$ (by Definition 6), where w_{ij} is the weight of the edge from v_i to v_j, and add it to LS_j (lines 3–6). When adding the new label $o : [a + w_{ij}, b + w_{ij}]$ to LS_j, depending on whether there exists a label $o : [a', b'] \in LS_j$ that contains the same uncertain object o, there are two cases to be considered.



(a) An example of $C(o)$ (b) An example of UNVD

Fig. 2. Examples.

- If $o:[a',b']$ exists, we first replace $o:[a',b']$ with a new label $o:[\min\{a+w_{ij},a'\},\min\{b+w_{ij},b'\}]$, and then check whether there exist labels in LS_j that are larger than this new label. If such labels exist, they will be removed from LS_j.
- If $o:[a',b']$ does not exist, we compare the new label $o:[a+w_{ij},b+w_{ij}]$ with all the labels in LS_j. If $o:[a+w_{ij},b+w_{ij}]$ is larger than every label in LS_j, $o:[a+w_{ij},b+w_{ij}]$ is directly discarded; If $o:[a+w_{ij},b+w_{ij}]$ equals approximately every label in LS_j, $o:[a+w_{ij},b+w_{ij}]$ is inserted into LS_j. Otherwise, $o:[a+w_{ij},b+w_{ij}]$ is inserted into LS_j and the labels that are lager than $o:[a+w_{ij},b+w_{ij}]$ are removed from LS_j.

(2) Determining the smallest label set(s) (by Definition 8) among all the label sets currently allocated to the unvisited vertices, and replacing the vertices in *CUR* with the vertex (vertices) corresponding to the smallest label set(s) (lines 9–11).

EXAM (Algorithm 3) is designed to compute the permanent labels for the current vertices returned in Function RETR. First, for each vertex $v_s \in CUR$, Algorithm 4 (Check) is executed (lines 1–3). Then, all the current vertices in *CUR* are inserted into *VI*, which is finally returned (lines 4–5). In Algorithm 4 (Check), for each current vertex $v_t \in CUR$ that is directly connected to a given current vertex $v_s \in CUR$, we first update the label set LS_t by adding new labels (lines 2–5), and then check whether LS_t keeps unchanged. If the answer is negative, then Algorithm 4 (Check) itself will be invoked, taking v_t and *CUR* as the inputs (lines 6–8).

It seems that endless loops may appear in EXAM, since the current vertices may form circles. But in fact, such endless loops do not exist, as illustrated in the following theorem.

Theorem 1. *Any current vertex can be allocated a permanent label set by being examined a limited number of times.*

Proof. We prove it by contradiction. Suppose a current vertex would be examined unlimited times, i.e., EXAM falls into endless loop, then according to the logic of Algorithm 3 (EXAM), there must exist at least one circle C that is composed of the current vertices, and the label set allocated to each current vertex in C is always changing. However, the label set LS_i of a vertex v_i cannot be always changing, since by RETR, LS_i keeps unchanged when the new label to be added is larger than every old label in LS_i, and a vertex v_i in C can always obtain a new label that is larger enough than every label in LS_i, due to that the edge weights are continuously accumulated to form new labels. We thus come to a contradiction and the theorem follows. □

Illustrative Example. To better understand Step 2 of MarkV, we now give an example, as shown in Table 3, to find the permanent labels (represented in bold) for all the vertices in Fig. 3, in which we assume the possible maximal and minimal distances from every uncertain object to the vertices that directly connected to it have been computed. In this example, we first set *CUR* and *VI* to be $\{o_1,o_2,o_3\}$ and $\{v_6\}$, respectively, since v_6 is located in $C(o_3)$. Meanwhile, v_6 is allocated a permanent label $\tilde{o_3}$, and the label sets of the other vertices are set to be empty. Then, in each of the next four iterations (see Table 3), the two functions RETR and EXAM are executed sequentially.

Take iteration 3 for example, in RETR, for each unvisited vertex $v_j \in V \setminus VI$, we add the weight of the edge $\overline{v_iv_j}$ to each label allocated to $v_i \in CUR$ in last iteration, i.e., v_{10}. Since v_{10} is connected to unvisited vertices v_2 and v_4, and the edge weights of $\overline{v_{10}v_2}$ and $\overline{v_{10}v_4}$ are 7 and 4, respectively, we can obtain new labels $o_3:[4,3]+7=o_3:[11,10]$ for v_2, and $o_3:[4,3]+4=o_3:[8,7]$ for v_4. Because the new label $o_3:[11,10]$ is larger than every label in $LS_2=\{o_1:[9,8]\}$, $o_3:[11,10]$ is discarded and LS_2 remains unchanged. Moreover, because the new label $o_3:[8,7]$ equals approximately every label in $LS_4=\{o_2:[8,7]\}$, we insert the new label $o_3:[8,7]$ into LS_4. Then, we compare the label sets $LS_2=\{o_1:[9,8]\}$, $LS_3=\{o_3:[8,6]\}$, $LS_4=\{o_2:[8,7],o_3:[8,7]\}$, $LS_7=\{o_2:[6,5]\}$, $LS_8=\{o_3:[11,9],o_2:[10,9]\}$ and $LS_9=\{o_1:[6,5]\}$ which are now allocated to the unvisited vertices, and select the smallest label sets LS_3, LS_7, and LS_9. Finally, we remove v_{10} from *CUR*, and insert v_3, v_7 and v_9 into *CUR*.

Next, in EXAM, since v_7 and v_9 are directly connected to each other, we add the weights of edges $\overline{v_9v_7}$ and $\overline{v_7v_9}$ to the labels in LS_7 and LS_9, respectively, and obtain two new labels $o_2:[16,15]$ and $o_1:[16,15]$ for v_7 and v_9, respectively. Because the two new labels $o_2:[16,15]$ and $o_1:[16,15]$ are respectively larger than every label in $LS_7=\{o_2:[6,5]\}$ and $LS_9=\{o_1:[6,5]\}$, $o_2:[16,15]$ and $o_1:[16,15]$ are discarded, and LS_7 and LS_9 keep unchanged. In this way, we can obtain three

Table 3
An example of Step 2 in MarkV.

Iteration	Operation	v_1	v_2	v_3	v_4	v_5	v_6 v_7	v_8	v_9	v_{10}	*CUR*	*VI*
0	Initial	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\boldsymbol{o_3}$ $\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	o_1, o_2, o_3	v_6
1	RTER	$o_1 : [2,1]$ $\emptyset$		$o_3 : [8,6]$	$o_2 : [8,7]$	$o_2 : [3,2]$	$\emptyset$	$o_3 : [11,9]$	$o_1 : [6,5]$	$o_3 : [4,3]$	v_1, v_5	v_6
	EXAM	$\boldsymbol{o_1} : [\mathbf{2},\mathbf{1}]$ $\emptyset$		$o_3 : [8,6]$	$o_2 : [8,7]$	$\boldsymbol{o_2} : [\mathbf{3},\mathbf{2}]$	$\emptyset$	$o_3 : [11,9]$	$o_1 : [6,5]$	$o_3 : [4,3]$	v_1, v_5	v_6, v_1
2	RTER		$o_1 : [9,8]$	$o_3 : [8,6]$	$o_2 : [8,7]$		$o_2 : [6,5]$	$o_3 : [11,9]$ $o_2 : [10,9]$	$o_1 : [6,5]$	$o_3 : [4,3]$	v_{10}	v_6, v_1, v_5
	EXAM		$o_1 : [9,8]$	$o_3 : [8,6]$	$o_2 : [8,7]$		$o_2 : [6,5]$	$o_3 : [11,9]$ $o_2 : [10,9]$	$o_1 : [6,5]$	$\boldsymbol{o_3} : [\mathbf{4},\mathbf{3}]$	v_{10}	v_6, v_1, v_5, v_{10}
3	RTER		$o_1 : [9,8]$	$o_3 : [8,6]$	$o_2 : [8,7]$ $o_3 : [8,7]$		$o_2 : [6,5]$	$o_3 : [11,9]$ $o_2 : [10,9]$	$o_1 : [6,5]$		v_3, v_7, v_9	v_6, v_1, v_5, v_{10}
	EXAM		$o_1 : [9,8]$	$\boldsymbol{o_3} : [\mathbf{8},\mathbf{6}]$	$o_2 : [8,7]$		$\boldsymbol{o_2} : [\mathbf{6},\mathbf{5}]$	$o_3 : [11,9]$	$\boldsymbol{o_1} : [\mathbf{6},\mathbf{5}]$		v_3, v_7, v_9	v_6, v_1, v_5, v_{10}, v_3, v_7, v_9
					$o_3 : [8,7]$			$o_2 : [10,9]$				
4	RTER		$o_1 : [9,8]$		$o_2 : [8,7]$			$o_3 : [11,9]$			v_2, v_4,	v_6, v_1, v_5, v_{10}, v_3, v_7, v_9
			$o_3 : [11,9]$		$o_3 : [8,7]$ $o_1 : [9,8]$			$o_2 : [10,9]$			v_8	
	EXAM		$bio_{\boldsymbol{1}} : [\mathbf{9},\mathbf{8}]$		$\boldsymbol{o_2} : [\mathbf{8},\mathbf{7}]$			$\boldsymbol{o_3} : [\mathbf{11},\mathbf{9}]$			v_2, v_4, v_8	v_6, v_1, v_5, v_{10}, v_3, v_7, v_9, v_2, v_4, v_8
			$\boldsymbol{o_3} : [\mathbf{11},\mathbf{9}]$		$\boldsymbol{o_3} : [\mathbf{8},\mathbf{7}]$ $\boldsymbol{o_1} : [\mathbf{9},\mathbf{8}]$			$\boldsymbol{o_2} : [\mathbf{10},\mathbf{9}]$				
	Possible NNs	o_1	o_1, o_3	o_3	o_1, o_2, o_3	o_2	o_3 o_2	o_2, o_3	o_1	o_3		

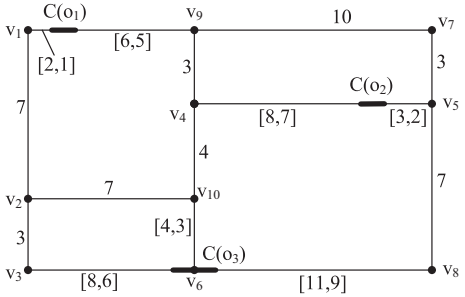


Fig. 3. An illustrative graph.

permanent label sets $LS_3 = \{o_3 : [8,6]\}$, $LS_7 = \{o_2 : [6,5]\}$ and $LS_9 = \{o_1 : [6,5]\}$ for v_3, v_7 and v_9, respectively. Meanwhile, v_3, v_7 and v_9 are inserted into *VI*.

Following a similar way, after conducting the four iterations as shown in Table 3, we can obtain the possible NN set for each vertex by collecting the uncertain objects in the permanent labels allocated to it.

3.2.2. Marking edges

In this section, we aim to find the u-edges as well as their corresponding possible NNs. If any two points on an edge $e \in E$ have the same possible NNs, then e is considered as a u-edge, and $nn\_dom(e)$ is equal to $nn\_dom(p)$, where p is an arbitrary point on e. Otherwise, we split e into sub-edges until any two points in a sub-edge have the same possible NNs, and each sub-edge is considered to be a u-edge.

For an edge on which there does not exist any object to be searched, a property of the relationship between the exact NN of a point on the edge and the NNs of the edge's endpoints has been used to process CNN queries [10,27]. In brief, the property can be described as follows: *the NN of any point on an edge must be one of the NNs of the two endpoints of the edge*. This property is only proposed for certain (with 100% probability) NNs. Below we show in Lemma 1 that a similar property also holds for possible NNs.

Lemma 1. *Suppose $\overline{v_s v_e} \in E$ is an edge on which there does not exist any extreme point, q is an arbitrary point on $\overline{v_s v_e}$ and o_j is a randomly selected uncertain object, if $o_j \in nn\_dom(v_s)$ and $o_j \in nn\_dom(v_e)$, then $o_j \in nn\_dom(q)$; Conversely, if $o_j \notin nn\_dom(v_s)$ and $o_j \notin nn\_dom(v_e)$, then $o_j \notin nn\_dom(q)$.*

Proof. By Observation 1, when $o_j \in nn\_dom(v_s)$ and $o_j \in nn\_dom(v_e)$, we have $dmax(v_s, o_k) \geqslant dmin(v_s, o_j)$ and $dmax(v_e, o_k) \geqslant dmin(v_e, o_j)$, where $o_k (k \neq j)$ is an arbitrary uncertain object. Moreover, for an arbitrary point p, there are,

$$\begin{cases} dmin(p,o_j) = \min\{dmin(v_s,o_j) + d(v_s,p), dmin(v_e,o_j) + d(v_e,p)\} \\ dmax(p,o_k) = \min\{dmax(v_s,o_k) + d(v_s,p), dmax(v_e,o_k) + d(v_e,p)\} \end{cases} \quad (4)$$

Hence, for an arbitrary uncertain object $o_k(k \neq j)$, we have $dmax(p,o_k) \geqslant dmin(p,o_j)$, which in turn means there does not exist any uncertain object o_k which satisfies $dmax(p,o_k) \geqslant dmin(p,o_j)$. Then according to Observation 1, we can get that $o_j \in nn\_dom(p)$.

Similarly, if $o_j \notin nn\_dom(v_s)$ and $o_j \notin nn\_dom(v_e)$, then by Observation 1, there must exist an uncertain object $o_k(k \neq j)$ such that $dmax(v_s,o_k) < dmin(v_s,o_j)$ and $dmax(v_e,o_k) < dmin(v_e,o_j)$. Moreover, Formula (4) always holds for any arbitrary point p. Hence, there must exist an uncertain object $o_k(k \neq j)$, such that $dmin(p,o_j) > dmax(p,o_k)$. Then, according to Observation 1, we can derive that $o_j \notin nn\_dom(p)$. □

For an edge $\overline{v_s v_e}$ on which no extreme points exist, if $nn\_dom(v_s) = nn\_dom(v_e)$, then by Lemma 1, we can get that $nn\_dom(\overline{v_s v_e}) = nn\_dom(v_s)$. Otherwise, $\overline{v_s v_e}$ needs to be further split into several sub-edges until all the points in a sub-edge have the same possible NNs, i.e., until all the u-edges derived from $\overline{v_s v_e}$ are obtained. A boundary point between adjacent u-edges derived from the same edge is called a split point. The concept of the split point has been used by CNN search in the literature [10,27]. But, the calculation method proposed in this paper is quite different from, and more complicated than that in traditional CNN search based on certain NNs.

In Fig. 4(a), where the NN of v_s and v_e are o_i and o_j, respectively, there exists only one split point S on edge $\overline{v_s v_e}$. However, in Fig. 4(b), where the sets of possible NNs of v_s and v_e are $\{o_i\}$ and $\{o_j\}$, respectively, there exist two split points S_1 and S_2 on edge $\overline{v_s v_e}$. In addition, $nn\_dom(\overline{v_s S_1}) = \{o_i\}$, $nn\_dom(\overline{S_1 S_2}) = \{o_i, o_j\}$ and $nn\_dom(\overline{S_2 v_e}) = \{o_j\}$. When the endpoints v_s and v_e have more possible NNs, the computation of the split points becomes more complicated. Before introducing our computation method, we first give a definition as follows.

Definition 9. A **break point** on a line segment l is a point where the possible NNs of the points on l may change.

A break point concerns a pair of uncertain objects o_i and o_j. Given an arbitrary edge $e = \overline{v_s v_e} \in E$ and two uncertain objects $o_i \in nn\_dom(v_s)$ and $o_j \in nn\_dom(v_e)$ $(o_i \neq o_j)$: (1) when $o_i \notin nn\_dom(v_e)$ and $o_j \notin nn\_dom(v_s)$, there exist two break points $\boldsymbol{b}_{(o_i,o_j)}$ and $\boldsymbol{b}_{(o_j,o_i)}$ on e; (2) when $o_i \in nn\_dom(v_e)$ and $o_j \notin nn\_dom(v_s)$, there exists only one break point $\boldsymbol{b}_{(o_i,o_j)}$ on e; (3) when $o_i \notin nn\_dom(v_e)$ and $o_j \in nn\_dom(v_s)$, there exists only one break point $\boldsymbol{b}_{(o_j,o_i)}$ on e.

Our computation of the break points is inspired by the work in [37]. Specifically, we compute $b_{(o_i,o_j)}$ and $b_{(o_j,o_i)}$ by the following formula.

$$\begin{cases} dmax(o_i,v_s) + d(v_s,b_{(o_i,o_j)}) = dmin(o_j,v_e) + d(v_e,b_{(o_i,o_j)}) \\ dmax(o_j,v_e) + d(v_e,b_{(o_j,o_i)}) = dmin(o_i,v_s) + d(v_s,b_{(o_j,o_i)}) \\ d(v_s,b_{(o_i,o_j)}) + d(v_e,b_{(o_i,o_j)}) = d(v_s,v_e) \\ d(v_e,b_{(o_j,o_i)}) + d(v_s,b_{(o_j,o_i)}) = d(v_s,v_e) \end{cases} \quad (5)$$

The variables $dmax(o_i,v_s)$, $dmin(o_j,v_e)$, $dmax(o_j,v_e)$ and $dmin(o_i,v_s)$ in the above formula can be obtained during searching the possible NNs for the vertices, as introduced in Section 3.2.1.

Without considering other break points, $b_{(o_i,o_j)}$ partitions the edge into two parts, and o_j is not a possible NN of every point on $\overline{v_s b_{(o_i,o_j)}}$, but a possible NN of every point on $\overline{b_{(o_i,o_j)} v_e}$. Similarly, $b_{(o_j,o_i)}$ partitions the edge into two parts, and o_i is not a possible NN of every point on $\overline{b_{(o_j,o_i)} v_e}$, but a possible NN of every point on $\overline{v_s b_{(o_j,o_i)}}$. Nevertheless, there may exist multiple break points concerning different uncertain objects on an edge. All these break points should be analyzed together to compute the split points (u-edges) on the edge.

Algorithm 5. MarkE

Input: An edge $\overline{v_s v_e} \in E$
Output: The u-edges derived from $\overline{v_s v_e}$, as well as their possible NNs
1: **if** there exist extreme points on $\overline{v_s v_e}$ **then**
2: Partition $\overline{v_s v_e}$ using the extreme points, and insert the sub-edges into *Tmp*;
3: **else**
4: Insert $\overline{v_s v_e}$ into *Tmp*;
5: **end if**
6: **for** each sub-edge $\overline{v'_s v'_e}$ in *Tmp* **do**
7: **if** $nn\_dom(v'_s) = nn\_dom(v'_e)$ **then**
8: Return $\overline{v'_s v'_e}$ and $nn\_dom(\overline{v'_s v'_e}) = nn\_dom(v'_s)$;
9: **else**
10: Return SplitEdge($\overline{v'_s v'_e}$);
11: **end if**
12: **end for**

Algorithm 6. SplitEdge

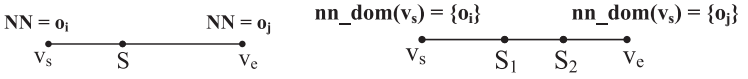
Input: $e = \overline{v'_s v'_e}$
Output: The u-edges derived from $\overline{v'_s v'_e}$, as well as their possible NNs
1: Initialize $TmpE \leftarrow e, TmpNN(e) \leftarrow \emptyset$;
2: **for** each pair of uncertain objects $o_i \in nn\_dom(v'_s)$ and $o_j \in nn\_dom(v'_e)$ **do**
3: **if** $o_i == o_j$ **then**
4: For each sub-edge $e' \in TmpE, TmpNN(e') \leftarrow o_i$;
5: **else if** $o_i \in nn\_dom(v'_e)$ and $o_j \notin nn\_dom(v'_s)$ **then**
6: Compute $b_{(o_i,o_j)}$ and update $TmpE$ with $b_{(o_i,o_j)}$;
7: For each sub-edge $e' \in TmpE$, if e' is contained in $\overline{v'_s b_{(o_i,o_j)}}, TmpNN(e') \leftarrow o_i$; Otherwise, $TmpNN(e') \leftarrow \{o_i, o_j\}$;
8: **else if** $o_j \in nn\_dom(v'_s)$ and $o_i \notin nn\_dom(v'_e)$ **then**
9: Compute $b_{(o_j,o_i)}$ and update $TmpE$ with $b_{(o_j,o_i)}$;
10: For each sub-edge $e' \in TmpE$, if e' is contained in $\overline{b_{(o_j,o_i)} v'_e}, TmpNN(e') \leftarrow o_j$; Otherwise, $TmpNN(e') \leftarrow \{o_i, o_j\}$;
11: **else**
12: Compute $b_{(o_i,o_j)}$ and $b_{(o_j,o_i)}$ and update $TmpE$ with $b_{(o_i,o_j)}$ and $b_{(o_j,o_i)}$;
13: For each sub-edge $e' \in TmpE$, if e' is contained in $\overline{v'_s b_{(o_i,o_j)}}, TmpNN(e') \leftarrow o_i$; If e' is contained in $\overline{b_{(o_i,o_j)} b_{(o_j,o_i)}}, TmpNN(e') \leftarrow \{o_i, o_j\}$; Otherwise, $TmpNN(e') \leftarrow o_j$;
14: **end if**
15: **end for**
16: Merge adjacent sub-edges that have the same possible NNs in $TmpE$ into a single edge.
17: Return each edge $e' \in TmpE$ and $TmpNN(e')$; // The points shared by two edges in $TmpE$ are the split points

Now we propose the whole algorithm MarkE (Algorithm 5) for calculating the u-edges derived from an edge, as well as their possible NNs. For a given edge $\overline{v_s v_e} \in E$, we first check whether there exist extreme points. If the answer is affirmative, we partition $\overline{v_s v_e}$ into sub-edges by using these extreme points, and insert all the sub-edges into a set Tmp (lines 1–2). Otherwise, we insert $\overline{v_s v_e}$ into Tmp directly (lines 3–4). Then, for each sub-edge $\overline{v'_s v'_e}$ in Tmp, we examine the possible NNs of its end points. Specifically, if the two endpoints have the same set of possible NNs, then the possible NNs of $\overline{v'_s v'_e}$ is directly obtained as the possible NNs of the endpoints (lines 7–8). Otherwise, Algorithm 6 (SplitEdge) is executed to compute the u-edges derived from $\overline{v'_s v'_e}$ (lines 9–10). Note that if there exist extreme points on $\overline{v_s v_e}$, we should compute the set of possible NNs of each extreme point. Suppose $E_i(o_j)$ is an extreme point, then we have $nn\_dom(E_i(o_j)) = \{o_j, \ldots, o_m\}$, where $o_j, \ldots, o_m$ are the uncertain objects whose uncertainty regions contain $E_i(o_j)$.

In Algorithm 6 (SplitEdge), we only consider the uncertain objects contained in $nn\_dom(v'_s)$ or $nn\_dom(v'_e)$, since only such uncertain objects are possible to be the NNs of a point on $\overline{v'_s v'_e}$ (Lemma 1). Thus, each pair of uncertain objects $o_i \in nn\_dom(v'_s)$ and $o_j \in nn\_dom(v'_e)$ will be examined.

In each examination, we use a set $TmpE$ to keep the current sub-edges derived from $e = \overline{v'_s v'_e}$, and assign each sub-edge e' a set $TmpNN(e')$ to maintain the presently known possible NNs of points on e'. By Lemma 1, if $o_i = o_j$, then o_i (or o_j) is a possible NN of any point on e. Thus, we insert o_i into $TmpNN(e')$ for each $e' \in TmpE$ (lines 3–4). If $o_i \neq o_j$, we compute the corresponding break points and further split the sub-edges in $TmpE$ by using the break points (lines 5–13). Specifically,

- When o_i is contained in both sets but o_j is only in $nn\_dom(v'_e)$, we compute break point $b_{(o_i,o_j)}$ and use it to partition the edges in $TmpE$. After updating $TmpE$ by replacing the out-of-date sub-edges with newly generated sub-edges, we compare the sub-edges in $TmpE$ with line segment $\overline{v'_s b_{(o_i,o_j)}}$. For each sub-edge $e' \in TmpE$ that is contained in $\overline{v'_s b_{(o_i,o_j)}}$, we insert o_i into $TmpNN(e')$. For other sub-edges in $TmpE$, we insert both o_i and o_j into $TmpNN(e')$ (lines 5–7).
- When o_j is contained in both sets but o_i is only in $nn\_dom(v'_s)$, we conduct a similar operation to the second case, but the break point $b_{(o_j,o_i)}$ is calculated (lines 8–10).
- When o_i and o_j are separately contained only in $nn\_dom(v'_s)$ and $nn\_dom(v'_e)$, break points $b_{(o_i,o_j)}$ and $b_{(o_j,o_i)}$ are computed (lines 11–13).



(a) Split point in NN search (b) Split points in NPNN search

Fig. 4. Examples of split points.

The algorithm stops until any pair of uncertain objects $o_i \in nn\_dom(v'_s)$ and $o_j \in nn\_dom(v'_e)$ has been examined. Since different sub-edges in *TmpE* may have the same set of possible NNs, we merge such sub-edges that share the same endpoints into a single edge (line 16). Finally, the sub-edges in *TmpE* are the u-edges derived from e, and the corresponding set *TmpNN* is the set of possible NNs (line 17).

Illustrative Example. To help understand Algorithm 6 (SplitEdge), we now give an example in Fig. 5, where we assume $nn\_dom(v'_s) = \{o_1, o_2\}$ and $nn\_dom(v'_e) = \{o_2, o_3\}$. Note that the execution steps are shown in a top-to-bottom order. *TmpE* is initialized to be $\overline{v'_s v'_e}$ and $TmpNN(\overline{v'_s v'_e})$ is empty. In the first step, o_1 and o_2 are examined. Because o_1 is only the possible NN of v'_s and o_2 is the possible NN of both endpoints, the break point $b_{(o_2,o_1)}$ is computed and *TmpE* is updated to be $\{\overline{v'_s b_{(o_2,o_1)}}, \overline{b_{(o_2,o_1)} v'_e}\}$. Meanwhile, $\{o_1, o_2\}$ and o_2 are inserted into $TmpNN(\overline{v'_s b_{(o_2,o_1)}})$ and $TmpNN(\overline{b_{(o_2,o_1)} v'_e})$, respectively. In the second step, o_1 and o_3 are examined, and the break points $b_{(o_1,o_3)}$ and $b_{(o_3,o_1)}$ are computed. In the third step, o_2 is inserted into $TmpNN(e')$ for each edge $e' \in TmpE$. In the last step, o_2 and o_3 are examined, and the split point $b(o_2, o_3)$ is computed. Finally, the edges $\overline{b_{(o_1,o_3)} b_{(o_2,o_3)}}, \overline{b_{(o_2,o_3)} b_{(o_3,o_1)}}$ and $\overline{b_{(o_3,o_1)} b_{(o_2,o_1)}}$ are merged into one edge, and we obtain two split points S_1 ($b(o_1, o_3)$) and S_2 ($b(o_2, o_1)$). S_1 and S_2 partition $\overline{v'_s v'_e}$ into three u-edges, $\overline{v'_s S_1}, \overline{S_1 S_2}$ and $\overline{S_2 v'_e}$. The corresponding sets of possible NNs are $\{o_1, o_2\}, \{o_1, o_2, o_3\}$ and $\{o_2, o_3\}$, respectively.

4. Processing NPNN queries

Although UNVD has the potential to answer various spatial queries such as NN queries, RNN queries and range queries, due to space limitation, we only focus on static and continuous PNN queries in road Networks (NPNN) in this paper. Given a query point q in a road network, a NPNN query returns the uncertain objects with non-zero probabilities for being the NN of q, as well as their probabilities. Thus, processing NPNN queries involves efficiently finding the answer objects and evaluating the probability of each answer object. In this section, we first show the computation of the probabilities, and then propose two indexes as well as the corresponding processing methods for static and continuous NPNN.

4.1. Evaluation of probability

Let q be a query point located on a u-edge e, and $P(o)$ be the probability of an uncertain object $o \in O$ being q's NN. Obviously, if $o \notin nn\_dom(e)$, then $P(o) = 0$; If o is the only possible NN of e, then $P(o) = 100\%$. Otherwise, if o is one of the several possible NNs associated with e, then we need to compute $P(o)$, as presented in the following.

Since the *pdf*s of any two uncertain objects are independent, for each $o \in nn\_dom(e)$, we have

$$P(o) = \prod_{\forall o_j \in nn\_dom(e) \wedge o_j \neq o} P(o_j, o) \tag{6}$$

where $P(o_j, o)$ is the probability of o_j being farther to q than o to q. Further, let $E_1 = \{e_1, e_2, \ldots\}$ and $E_2 = \{e'_1, e'_2, \ldots\}$ be the sets of line segments which constitute $C(o)$ and $C(o_j)$, respectively, then

$$P(o_j, o) = \sum_{e_i \in E_1} P(e_i) \sum_{e'_j \in E_2} P(e'_j) P(o_j, o | e_i, e'_j) \tag{7}$$

where $P(e_i)$ and $P(e'_j)$ are the probabilities of o being located on e_i and e'_j, respectively, and $P(o_j, o|e_i, e'_j)$ is the probability of o_j being farther to q than o to q, when o and o_j are located on e_i and e'_j, respectively.

Since $P(e_i)$ and $P(e'_j)$ are known a *priori*, we only need to calculate $P(o_j, o|e_i, e'_j)$. As shown in Fig. 6, suppose the two endpoints of e, e_i and e'_j are (v_s, v_e), (v_{s1}, v_{e1}) and (v_{s2}, v_{e2}), respectively, and the lengths of them are L, L_1 and L_2 respectively. Moreover, for presentation convenience, we let $d(v_{s1}, p_1) = y_1, d(v_s, q) = x$ and $d(v_{s2}, p_2) = y_2$. When $u \in \{v_{s1}, v_{e1}\}$ and $w \in \{v_s, v_e\}$, according to the definition of shortest-path distance, we have $d(u, w) = \min_{1 \leqslant k \leqslant Z_1}\{d(u, E_k(o)) + d(E_k(o), w)\}$, where Z_1 is the total number of extreme points of o. Similarly, when $u \in \{v_{s2}, v_{e2}\}$ and $w \in \{v_s, v_e\}$, we have $d(u, w) = \min_{1 \leqslant k \leqslant Z_2}\{d(u, E_k(o_j)) + d(E_k(o_j), w)\}$, where Z_2 is the total number of extreme points of o_j.

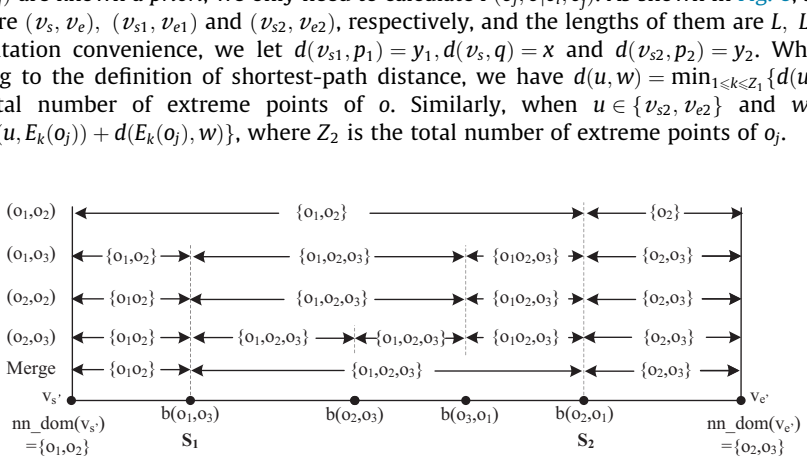


Fig. 5. An illustrative example of Algorithm 6 (SplitEdge).

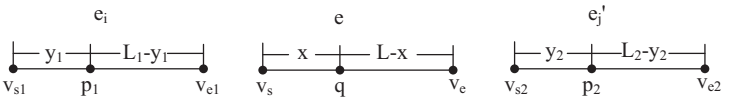


Fig. 6. An example of e_i and e_j'.

It is obvious that $P(o_j, o|e_i, e_j') = P\{d(p_1, q) < d(p_2, q)\}$. For simplicity, we use d_1, d_2, d_3 and d_4 to denote $d(v_{s1}, v_s)$, $d(v_{e1}, v_s)$, $d(v_{s1}, v_e)$ and $d(v_{e1}, v_e)$, respectively, and use d_1', d_2', d_3' and d_4' to denote $d(v_{s2}, v_s)$, $d(v_{e2}, v_s)$, $d(v_{s2}, v_e)$ and $d(v_{e2}, v_e)$, respectively. Then $d(p_1, q)$ and $d(p_2, q)$ can be represented as follows:

$$\begin{cases} d(p_1, q) = \min\{y_1 + d_1 + x, y_1 + d_2 + L - x, L_1 - y_1 + d_3 + x, L_1 - y_1 + d_4 + L - x\} \\ d(p_2, q) = \min\{y_2 + d_1' + x, y_2 + d_2' + L - x, L_2 - y_2 + d_3' + x, L_2 - y_2 + d_4' + L - x\} \end{cases}$$

Clearly, $d(p_1, q)$ can be expressed by a continuous piecewise function of y_1 and x over a finite region R_1 of the xy_1-plane, and $d(p_2, q)$ can be expressed by a continuous piecewise functions of y_2 and x over a finite region R_2 of the xy_2-plane. The corresponding piecewise functions are presented below:

$$d(p_1, q) = \begin{cases} y_1 + d_1 + x, & (x, y) \in D_1 \\ y_1 + d_2 + L - x, & (x, y) \in D_2 \\ L_1 - y_1 + d_3 + x, & (x, y) \in D_3 \\ L_1 - y_1 + d_4 + L - x, & (x, y) \in D_4 \end{cases} \tag{8}$$

$$d(p_2, q) = \begin{cases} y_2 + d_1' + x, & (x, y) \in D_1' \\ y_2 + d_2' + L - x, & (x, y) \in D_2' \\ L_1 - y_2 + d_3' + x, & (x, y) \in D_3' \\ L_1 - y_2 + d_4' + L - x, & (x, y) \in D_4' \end{cases} \tag{9}$$

where $D_i (1 \leqslant i \leqslant 4)$ is a subdomain of each subfunction of $d(p_1, q)$ and $D_i' (1 \leqslant i \leqslant 4)$ is a subdomain of each subfunction of $d(p_2, q)$. The domain of $d(p_1, q)$ is $R_1 = [0, L] \times [0, L_1]$ and the domain of $d(p_2, q)$ is $R_2 = [0, L] \times [0, L_2]$.

By using $g(x, y_1, y_2)$ to denote $d(p_1, q) - d(p_2, q)$, and using $f_1(y_1)$ and $f_2(y_2)$ to respectively represent the continuous *pdf*s of p_1 and p_2, $P(o_j, o|e_i, e_j')$ can be rewritten as:

$$P(o_j, o|e_i, e_j') = \sum_{D_i \in \mathcal{D}} \int_{D_i} f_1(y_1) dy_1 \sum_{D_i' \in \mathcal{D}'} \int_{D_i' \cap (g(x, y_1, y_2) < 0)} f_2(y_2) dy_2 = \sum_{D_i \in \mathcal{D}} \sum_{D_i' \in \mathcal{D}'} \iint_{\Omega} f_1(y_1) f_2(y_2) dy_2 dy_1 \tag{10}$$

where $\mathcal{D} = \{D_1, D_2, D_3, D_4\}$, $\mathcal{D}' = \{D_1', D_2', D_3', D_4'\}$ and $\Omega = \{(x, y_1, y_2) | (x, y_1) \in D_i, (x, y_2) \in D_i', g(x, y_1, y_2) < 0)\}$. The inner integral is integrated with respect to y_2, regarding x and y_1 as constants, while the outer integral is integrated with respect to y_1, regarding x as a constant. The final integral result is a function of x. Based on the above discussion, we can derive $P(o)$ as a function of x.

We now give a simple example in Fig. 7, which is based on the UNVD shown in Fig. 2(b), to illustrate how to compute the probability. For simplicity, we assume the precise position of each uncertain object o is evenly distributed within $C(o)$ and the length of any line segment included in $C(o)$ is 1. It is clear that $P(o_1) = 100\%$ and $P(o_2) = P(o_3) = 0$ when the query point is located on $\overline{v_1 v_2}$, due to that o_1 is the only possible NN of $\overline{v_1 v_2}$.

Assume the query point q is located on $\overline{s_1 s_2}$, then the probability $P(o_2)$ of o_2 being the NN of q is 0, since the set of possible NNs of q is $\{o_1, o_3\}$. Moreover, we have $P(o_1) = P(o_3, o_1)$, where $P(o_3, o_1)$ is the probability of o_3 being farther to q than o_1 to q. It is obvious that the probability of o_1 located on e_i is 100%, and the probabilities of o_3 located on $\overline{E_1(o_3) v_6}$, $\overline{E_2(o_3) v_6}$ and $\overline{E_3(o_3) v_6}$ are all $\frac{1}{3}$. Thus, $P(o_3, o_1) = \frac{1}{3}(P(o_3, o_1|e_i, \overline{E_1(o_3) v_6}) + P(o_3, o_1|e_i, \overline{E_2(o_3) v_6}) + P(o_3, o_1|e, \overline{E_3(o_3) v_6}))$, where $P(o_3, o_1|e_i, \overline{E_1(o_3) v_6})$,

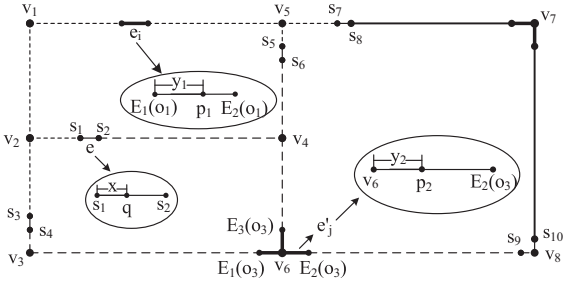


Fig. 7. Probability evaluation.

$P(o_3, o_1|e_i, \overline{E_2(o_3)v_6})$ and $P(o_3, o_1|e, \overline{E_3(o_3)v_6}))$ are the probabilities of o_3 being farther to q than o_1 when o_3 is located on $\overline{E_1(o_3)v_6}$, $\overline{E_3(o_3)v_6}$ and $\overline{E_2(o_3)v_6}$, respectively. Without loss of generality, suppose $d(v_2, v_4) = d(v_3, v_6) = d(v_1, v_5) = 11$, $d(v_2, v_3) = d(v_4, v_6) = d(v_1, v_2) = d(v_5, v_4) = 5$, then based on the construction method of UNVD, we can get that $d(v_2, s_1) = 2.5$, $d(s_1, s_2) = 1.5$. Let p_1 and p_2 denote the precise positions of o_1 and o_3, respectively. To compute $P(o_3, o_1|e_i, \overline{E_2(o_3)v_6})$, we assume p_2 is located on $e_j = \overline{E_2(o_3)v_6}$ and $d(E_2(o_3), p_2) = y_2$. Consequently, there are $d(p_1, q) = y_1 + x + 11.5$ and $d(p_2, q) = 12 + y_2 - x$, where $x \in [0, 1], y_1 \in [0, 1]$ and $y_2 \in [0, 1]$. Then, given the condition of Eq. (10) and $g(x, y_1, y_2) = d(p_1, q) - d(p_2, q) = 2x + y_1 - y_2 - 0.5$, we can get,

$$P(o_3, o_1|e, e_1) = \begin{cases} 1 - \frac{1}{2}(0.5 + 2x)^2, 0 \leqslant x \leqslant 0.25 \\ \frac{1}{2}(1.5 - 2x)^2, 0.25 \leqslant x \leqslant 0.75 \\ 0, 0.75 \leqslant x \leqslant 1 \end{cases}$$

Following a similar way, $P(o_3, o_1|e_i, \overline{E_1(o_3)v_6})$ and $P(o_3, o_1|e_i, \overline{E_3(o_3)v_6})$ can be computed, which means $P(o_1)$ can be readily obtained.

4.2. Index and NPNN processing

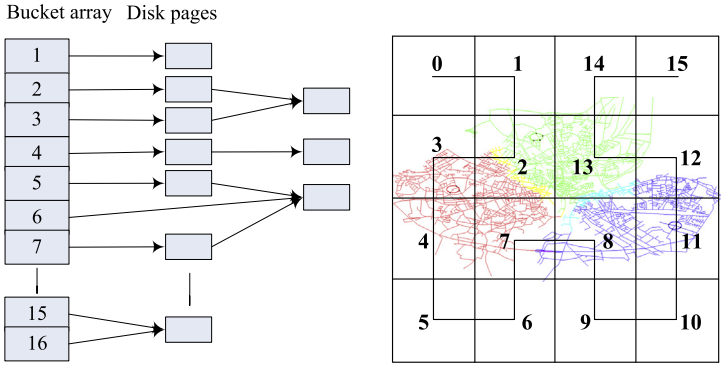
In this section, we study how to process NPNN queries by utilizing UNVD. Since the networks and uncertain objects studied in this work are considered to be static, it is clear that the UNVD can be re-used for a large number of queries. Hence, we pre-compute the entire UNVD. Moreover, since computing the probability online requires more time and needs to store the information of the original network (including edge weight and connection), we also pre-compute the probabilities for the possible NNs. By storing the possible NNs as well as the corresponding probabilities (functions) together with each u-edge in UNVD, the results of a NPNN query can be obtained immediately, once the corresponding u-edges are located.

A NPNN query can be static or continuous. A static NPNN query searches the possible NNs as well as the corresponding probabilities for a specified point. To answer a static NPNN query issued by a query point q, we need to find the u-edges that contain q, and the union of the possible NNs associated with these u-edges is the set of possible NNs of q. Unlike static NPNN, a continuous NPNN query requests the possible NNs of a specified path, which means the query object is not a point but a set of line segments. To answer a NPNN query issued by a query path p, which consists of n straight line segments, we divide the query into n sub-queries. A sub-query is issued by a straight line segment l, and returns a set of intervals of l, each associated with a series of possible NNs and corresponding probabilities (functions). For each sub-query, our task is to find the u-edges overlapped with the query line segment.

Observe that by using UNVD, the problem of NPNN search is reduced to the "where am I" problem in road networks, which only requires the knowledge about the u-edges. Hence, the main task is to enable efficient retrieval of all the u-edges that go through a specified point (in static NPNN) or overlap with a set of specified line segments (in continuous NPNN). Below we present two index structures, namely *gIndex* and *qIndex*, which win by ease of implementation, conceptual clarity and reasonable time cost on processing static and continuous NPNN, respectively.

4.2.1. gIndex

gIndex is implemented as a one-dimensional hash table, which means the array of buckets has to be stored in main memory. With the development of memory techniques, it is not difficult to achieve this. *gIndex* partitions the entire UNVD space into a number of equal-sized regions, each called a grid cell. The Hilbert curve [16] is used to sort the grid cells in a linear order, based on which the grid cells are assigned to the buckets sequentially. In this way, each bucket corresponds to a grid



(a) *gIndex* structure (b) partitioning

Fig. 8. An example of *gIndex*.

cell and contains a pointer pointing to a list of disk pages. Each page contains a set of 3-tuples in the form of $< Position, nn\_dom, Probability >$, where *Position* records the endpoints of a u-edge, *nn_dom* is the set of possible NNs, and *Probability* indicates the probability (function) of each possible NN. To avoid waste of space and reduce the number of I/Os during processing NPNN queries, different buckets are allowed to point to the same disk pages. The construction of *gIndex* includes four steps, as presented in the following.

Step 1. Algorithm 7 (GetSplitLevel) is executed to determine the split level n. It starts by considering the entire UNVD space D as the first grid cell, and then partitions the grid cell into four equal sub-cells repeatedly until the current cell overlaps with no more than α u-edges. Here α denotes the maximum number of u-edges overlapped with a grid cell. In this way, the UNVD space can be partitioned into $2^n \times 2^n$ equal-sized grid cells by 2^n vertical lines $x = W/2^i$ $(i = n, \ldots, 1)$ and 2^n horizon lines $y = H/2^i$ $(i = n, \ldots, 1)$, where W and V are the weight and height of the UNVD space, respectively.

Algorithm 7. GetSplitLevel

Input: Grid cell C
Output: The split level n
1: $X \Leftarrow$ the number of the u-edges overlapped with C;
2: **if** $X > \alpha$ **then**
3: Split C to four equal sub-cells c_1, c_2, c_3 and c_4;
4: $temp = \max\{\text{GetSplitLevel}(c_1), \text{GetSplitLevel}(c_2), \text{GetSplitLevel}(c_3), \text{GetSplitLevel}(c_4)\}$;
5: $n = temp + 1$;
6: **else**
7: $n = 0$;
8: **end if**
9: Return n;

Step 2. A Hilbert curve [31] with an order of n (the split level) is generated to sort the grid cells in a linear order. We use Hilbert curve since it has better locality-preserving behavior than other curves, such as Z-order, Gray-code and Peano curve. This property is important because NPNN queries, especially the continuous NPNN queries, usually search for u-edges that are closely located. For simplicity, we assume the UNVD space is square, but it is worth noting that it is also possible to implement Hilbert curves efficiently when the UNVD space does not form a square [14].

Step 3. Create an array of 4^n elements (buckets), and allocate each bucket a list of disk pages. Let n_b be the number of the u-edges which overlap with the grid cell corresponding to bucket b. Depending on whether n_b fills in some pages, we classify the buckets into two types. A bucket b belongs to the first type if $\alpha - \alpha \% n_b$ is smaller than a given threshold θ. Otherwise, b belongs to the second type. For the buckets belonging to the first type, we directly allocate a linked list of $\lceil \frac{n_b}{\alpha} \rceil$ disk pages to each bucket b of them. For the buckets belonging to the second type, we search which buckets can be allowed to share the same pages. Such a set of buckets, say $g_k (1 \leqslant k \leqslant m)$, has to satisfy the following two conditions: (1) $\alpha - \sum_{k=1}^{m} \alpha \% n_{g_k} < \theta$; and (2) the difference between the maximal and minimal bucket indexes is smaller than a predefined threshold γ. γ indicates how near these buckets should be, and we set $\gamma = 10$ in this work. For each bucket g_k in the set, we allocate a list of $\lfloor \frac{n_{g_k}}{\alpha} \rfloor$ disk pages to g_k. Meanwhile, we allocate a special page which is linked to these k buckets ($g_k (1 \leqslant k \leqslant n)$) at the same time. For other buckets belonging to the second type, we directly allocate a linked list of $\lceil \frac{n_b}{\alpha} \rceil$ disk pages to each bucket b of them.

Step 4. Insert u-edges to the disk pages. For each u-edge, we first find the grid cells overlapped with it, and then compute the hash key values (Hilbert curve positions) of these grid cells. Given the mapping function of the Hilbert curve, it is easy to convert the coordinates into Hilbert curve positions. Based on the hash key values, we insert the u-edge into the pages pointed by the buckets, of which the array indexes are equal to the hash key values.

An example of *gIndex* with 16 buckets is shown in Fig. 8(a), and the corresponding partitioning on the UNVD space is shown in Fig. 8(b).

The static NPNN processing using *gIndex* works as follows. Suppose q is a query point, we first compute its Hilbert curve position, and then access the corresponding bucket to retrieve the associated disk pages. By comparing q with the u-edges stored in the pages, we can find the exact u-edge where q resides. Finally, based on q's position in the u-edge, we can compute the probability for each possible NN. For continuous NPNN queries, since they can be divided into several sub-queries, we focus on processing a single sub-query which is issued by a line segment l. To process a continuous NPNN sub-query, we first compute the Hilbert curve positions of the grid cells overlapped with l, and then access disk pages to find the u-edges that overlap with l. Such u-edges' endpoints divide l into several parts, and the possible NNs of each part are the same as that of the u-edge overlapping with it. Finally, a set of intervals of l, together with their corresponding possible NNs and probabilities (functions), are returned.

4.2.2. qIndex

qIndex adopts a framework similar to PMR quad tree [17]. A non-leaf node records a pointer pointing to each of its four child nodes as well as the coordinates of the upper-left and lower-right points of the corresponding region. A leaf node stores

a linked list of disk pages, and each page contains a set of 3-tuples in the form of $< Position, nn\_dom, Probability >$, where *Position* records the coordinates of the u-edge's endpoints, *nn_dom* is the set of possible NNs and *Probability* indicates the probability (function) for each possible NN. The region covered by each child node is one-fourth of the region covered by its parent node, while the entire UNVD space is covered by the root node.

Similar to the construction of *gIndex*, a predefined threshold α is used to control the size of the grid cells. Algorithm 8 (Split) shows the constructing procedure of *qIndex*. This algorithm is a recursive procedure, and is first invoked for $g = root$, where $g.region = root.region$ is the entire UNVD space. For each node g, we check whether the number of u-edges which overlap with $g.region$ exceeds α (line 1). If the answer is affirmative, then g is split into four child nodes and we execute Split(gg) for each child node gg (line 3). Otherwise, new pages are allocated for g and the u-edges overlapping with $g.region$ are inserted into the new pages (line 5).

Algorithm 8. Split

Input: Node g
1: $X \Leftarrow$ the number of u-edges overlapped with $g.region$;
2: **if** $X > \alpha$ **then**
3: Split g into four child nodes gWN, gEN, gWS and gES, and partition $g.region$ into four equal regions and assign each region to one child node;
4: Split (gWN); Split (gEN); Split (gWS); Split (gES);
5: **else**
6: Allocate new pages for g and insert the u-edges overlapping with $g.region$ into the new pages;
7: **end if**

The static NPNN processing using *qIndex* works as follows. Suppose the query point is q, we first traverse *qIndex* to find the leaf node whose region contains q, and then retrieve the associated disk pages to find the u-edges that go through q. Finally, based on q's location, we compute the probabilities for each possible NN. To answer a continuous NPNN sub-query issued by a line segment l, we first traverse *qIndex* to find the leaf nodes that overlap with l, then retrieve the associated disk pages to find the u-edges that overlap l. Finally, a set of intervals of l as well as their corresponding possible NNs and probabilities are returned. The possible NNs of each interval are the same as that of the u-edge overlapped with it.

5. Experiment

We conduct a set of experiments to evaluate the performance of the proposed methods. Section 5.1 describes the experiment setup, and Section 5.2 discusses the experimental results.

5.1. Experiment setup

We use the generator described in [5] to generate uncertain objects and query points. To be exact, we first use the generator [5] to create a set P of discrete points, each of which represents an uncertain object $o \in O$. Then for each point $p \in P$, we obtain a circle with a center p and a diameter of 30 units. Finally, the line segments covered by the circle is regarded as the uncertainty region $C(o)$. We assume the *pdf*s of uncertain objects follow a uniform distribution. To generate the query line segment for a continuous sub-query, we randomly select two different points on an edge, which is randomly selected from the original network, and the interval between the two points is taken as the query line segment.

All the algorithms are evaluated on both synthetic and real datasets. Three real datasets downloaded from [29] are used in our experiments: California road network with 21,047 vertices and 21,692 edges, San Francisco road network with 174,955 vertices and 223,000 edges, and city of Oldenburg road network with 6104 vertices and 7034 edges. For short, the three road networks are denoted by "SF", "CA" and "OL", respectively. In addition, a series of synthetic graphs are generated by using GTgraph [36]. The extra edges between pairs of vertices in the synthetic graphs are deleted, since road network vertices are generally connected by only one edge in reality. By default, a synthetic graph is labeled as "Syn" and has 100k vertices and 150k edges that are evenly distributed.

Since no previous approaches have been proposed to answer NPNN queries, we compare *gIndex* and *qIndex* with two techniques termed as *Rtree* and *Transformed*, both are evolved from existing methods. Specifically, *Rtree* uses the Hilbert R-tree [25] to index the u-edges, which are represented by the minimal bounding rectangles (MBRs). In a Hilbert R-tree, the u-edges in the same page are more likely to be close in the UNVD space and the resulting leaf nodes in R-tree tend to have smaller areas. *Transformed* adopts the data structure proposed in [20] to index the u-edges, which stores a u-edge only once and is proved to perform as well as or better than the representative methods using minimum bounding rectangles or end-points. The basic idea of *Transformed* is to transform the u-edges into points in four-dimensional space. To organize the transformed points, we use grid file [34], since it is suitable for indexing point objects. For simplicity, we use the most efficient

implementations of grid partitions which are obtained by drawing the boundary lines at fixed values on each dimension. By *Transformed*, static and continuous NPNN queries are transformed into range queries on points in four-dimensional space.

The bucket array in *gIndex*, non-leaf nodes in *qIndex* and *Rtree*, and the grid directory in *Transformed* are all stored in the main memory. In our experiments, the amounts of memory occupied by *Rtree* and *Transformed* are higher than that of *gIndex* and *qIndex*. *qIndex* takes up more memory space than *gIndex* when the u-edges are distributed more evenly, but requires less with skewed data. In all the indexes, the u-edges are stored in the disk. In the default setting, the page size is 4096 bytes, and the maximal number of u-edges contained in a grid cell α is 100.

To improve efficiency, the UNVD construction algorithms are implemented by `C/C++` language with OpenMP, which allows us to add simple parallelism through OpenMP directives. Other algorithms are implemented in Java by using XXL [4]. All the programs are tested on an Intel dual-core, 3.07 GHz, 4G-main memory machine.

5.2. Experimental results

5.2.1. UNVD construction

In this set of experiments, we examine the UNVD-constructing approach against data sets SF, CA, OL and Syn. Due to space limitations, the results on different data sets are shown in the same graph. Recall that during the UNVD construction, we mark the network vertices and edges successively. Fig. 9(a) shows the ratio (R) of the time spent on marking vertices by utilizing the straightforward method mentioned in Section 3.2.1, to the total time cost of constructing UNVD. We can see that marking vertices takes up most of the UNVD-constructing time on all the data sets (R exceeds 75% when $|O| = 100$, and

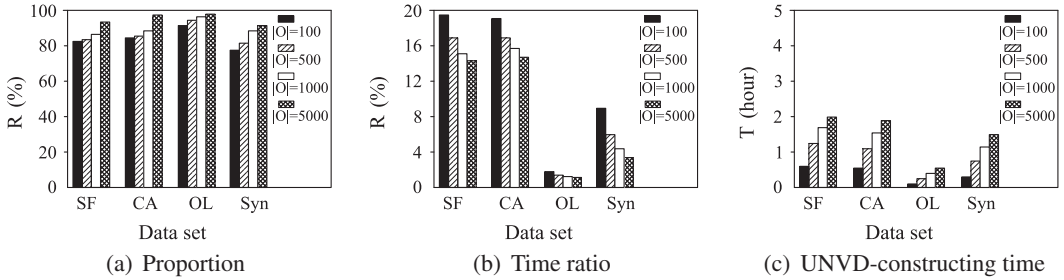


Fig. 9. Analysis of UNVD-constructing algorithms.

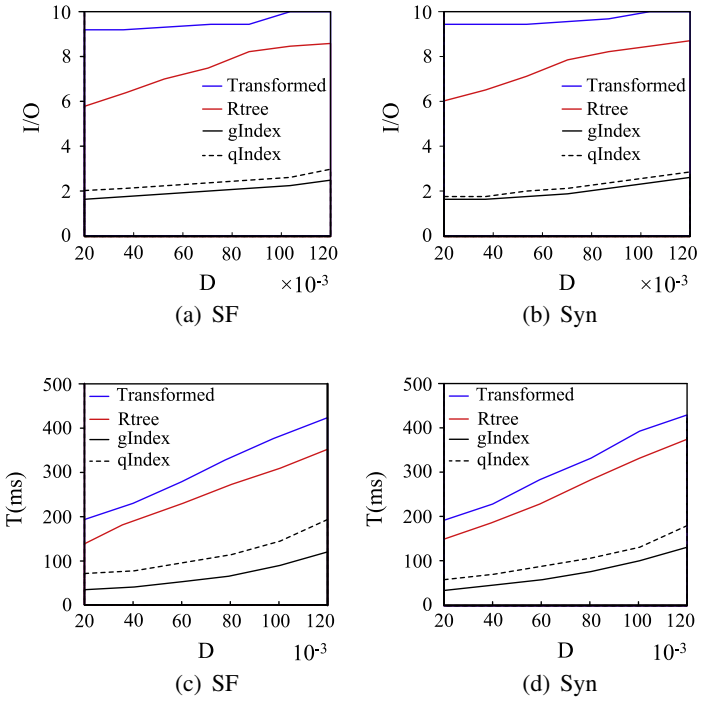


Fig. 10. I/O in static queries.

exceeds 90% when $|O| = 5000$), and the proportion R increases as $|O|$ grows. The reason lies in that the network has to be scanned $O(|O|)$ times to compute the possible maximal and minimal distances during the process of finding the possible NNs for all the vertices. The results described in Fig. 9(a) illustrate that efficiency enhancement in marking vertices is crucial.

Fig. 9(b) depicts the time ratio R of MarkV to *straight* (the straightforward method mentioned in Section 3.2.1). It can be seen that R is relatively small (at most 19.6% when $|O| = 100$) and decreases as $|O|$ grows on all the data sets. This is because the time cost of *straight* is proportional to $|O|$, while MarkV only needs to traverse the network once.

Fig. 9(c) shows the UNVD construction time for different data sets. As can be seen, for all data sets, the construction time grows as $|O|$ increases. This is because according to the construction approach introduced in Section 3.2, the more uncertain objects there are, the more calculation operation it takes. But it is worth mentioning that the time cost T is still acceptable (when $|O| = 5000$, even for the largest data set SF, T does not exceed 2 h).

5.2.2. Query performance

In this set of experiments, we evaluate the I/O performance and query time of all the indexes on data sets SF, CA, OL and Syn. Since the data sets vary in size, we use the density of uncertain objects, defined as $D = \frac{|O|}{|V|}$, rather than $|O|$, to evaluate the query performance. We examine the running time of 50 queries. Due to space limitations, only the results on SF and Syn are presented. For a data set, when D increases, the probability of an network edge being split into more u-edges increases, and thus $|U|$ also grows. Fig. 10(a) and (b) shows the I/Os of static NPNN queries versus D on SF and Syn, respectively. As can be seen, *gIndex* and *qIndex* require significantly less number of I/Os than *Transformed* and *Rtree*. The reason lies in that, *Transformed* has to retrieve a great number of grids since the static NPNN problem has been transformed into range queries with large query areas. For *Rtree*, when determining which object is associated with a specific point, many extra objects may be searched, because in *Rtree*, an object is only associated with one bounding rectangle, but the area spanned by it may be included in several bounding rectangles. However, in *gIndex* and *qIndex*, we only need to look for the leaf node (or bucket) that contains the query point. In most cases, *gIndex* has small grid cells than *qIndex*, and thus requires less I/Os. This explains the difference between *gIndex* and *qIndex*.

The query time for all the indexes can be divided into four components: (1) finding target page(s); (2) disk page access; (3) finding desired u-edges; and (4) obtaining possible NNs and probabilities (in static NPNN queries) or probability functions (in continuous NPNN queries). In all the experiments, accessing disk pages is the most time-consuming operation. Fig. 10(a) and (d) evaluate the query time on SF and Syn, respectively. As can be seen, as D increases, the query time of all the indexes increases correspondingly. There are two main reasons: First, the increasing number of I/O increases the index retrieval time in *Rtree* and *qIndex*; Second, a larger D incurs more u-edges and more possible NNs, which in turn will increase the probability computation time in all the indexes.

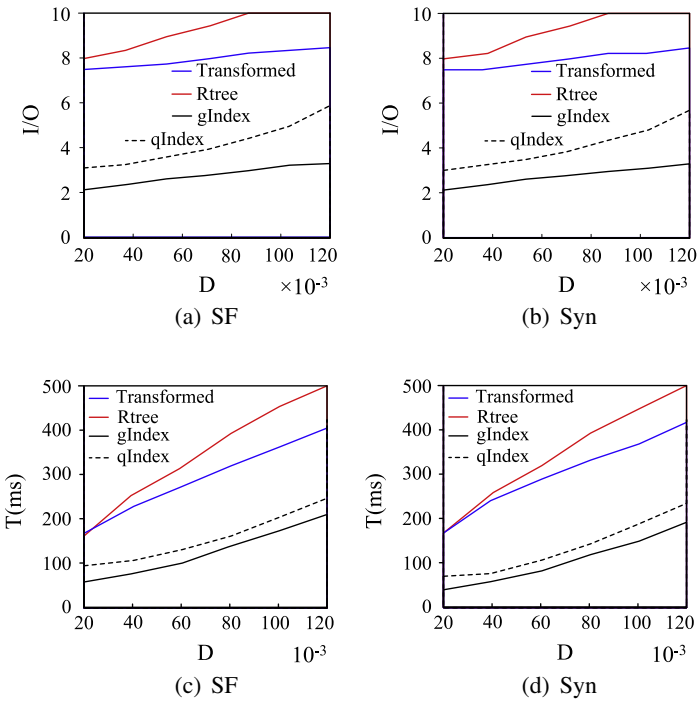


Fig. 11. I/O in continuous sub-queries.

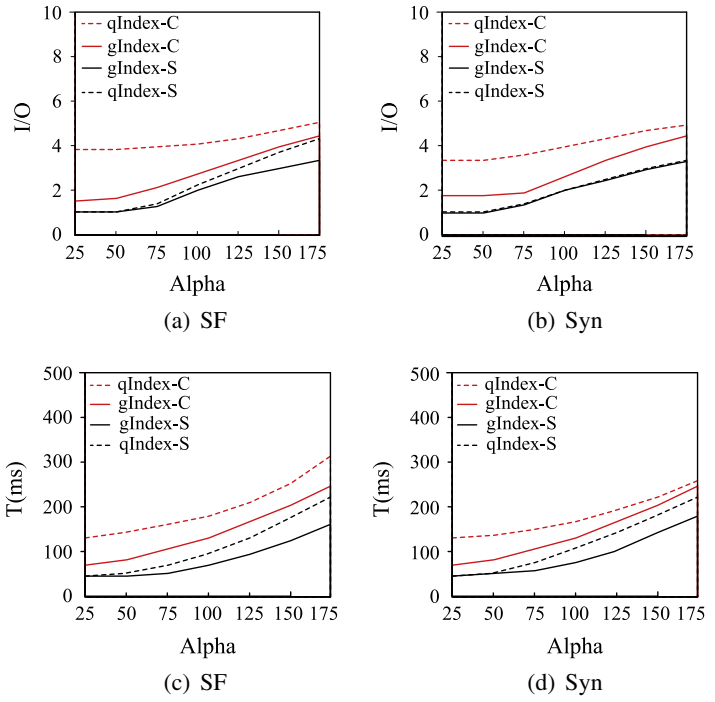


Fig. 12. Effect of α.

Since a continuous NPNN query can be divided into several sub-queries, each taking a single line segment as the query object, we also examine the performance of the sub-queries. The average performance of an entire continuous NPNN query can be roughly estimated by multiplying the performance of a sub-query by the number of the sub-queries. Fig. 11(a) and (b) evaluate the I/O performance in continuous NPNN sub-queries. As D increases, the number of I/Os in all the indexes also increase, and *gIndex* and *qIndex* outperform the other two schemes, due to the similar reasons as in the static context. Note that *gIndex*, *qIndex* and *Rtree* require more I/Os than in the static NPNN queries, while *Transformed* requires less. Compared to a single point, a query line segment tends to cover more grid cells in *gIndex* and *qIndex*, and belongs to more branches in *Rtree*. Thus, more disk pages need to be retrieved in these indexes. For *Transformed*, since the query line segment has been transformed to a point, and the query area in the transformed space has been greatly reduced, it requires less I/Os than in the static NPNN queries. Besides, *gIndex* outperforms *qIndex* in continuous queries, due to the organization of u-edges in disk pages by using Hilbert order. Fig. 11(c) and (d) evaluate the query time in continuous sub-queries. As can be seen, as D increases, the query time for all the indexes increases, due to the similar reason as in static queries.

In the last set of experiments, we vary the value of α (Recall that α denotes the maximal number of u-edges overlapped with a grid cell in *gIndex* and *qIndex*) from 25 to 175 to examine the performance of *gIndex* and *qIndex* in terms of I/O and query time. D is set to be 0.08. Due to space limitations, the results in static queries (with the suffix "S") and continuous sub-queries (with the suffix "C") are put in the same graph. Fig. 12(a) and (b) shows the number of I/Os on SF and Syn, respectively, while Fig. 12(c) and (d) present the query times against SF and Syn, respectively. In Fig. 12(a) and (b), all the I/Os increase as α grows. This is because when α grows, more disk pages are needed to store the u-edges overlapped with one grid cell, which means we have to access more disk pages to find the target u-edges in *gIndex* and *qIndex*, for static and continuous sub-queries. Besides, *gIndex-C* performs better than *qIndex-C*, which benefits from the utilization of Hilbert curve during the construction of *gIndex*. All the curves increase as α grows in Fig. 12(c) and (d), mainly owing to the increase in the number of I/Os.

6. Conclusion

In this paper, we addressed the problem of processing queries over uncertain data in road network space. As far as we know, this work serves as the first attempt to solve the given problem. First, we presented an efficient method to construct the Voronoi diagram on uncertain objects in road networks. Then, we designed two data structures, namely *gIndex* and *qIndex*, to index the built Voronoi diagram. We also illustrated how to process NPNN queries by using our indexes. Finally, we conducted extensive experiments to evaluate our UNVD constructing methods and the NPNN processing algorithms. Experimental results show that our approach performs quite well in terms of both I/O performance and query time.

For future work, we plan to extend our method to process NPkNN ($k > 1$) queries. Also, it would be interesting to study how to use the UNVD to support other location-based queries, such as reverse nearest neighbor queries.

Acknowledgments

The authors would like to thank the anonymous reviewers for their constructive and helpful comments. This work was substantially supported by the State Key Program of National Natural Science of China under Grant No. 61332001, National Natural Science Foundation of China under Grants Nos. 61173049, 61300045 and 61309002, China Postdoctoral Science Foundation under Grant No. 2013M531696 and the Fundamental Research Funds for the Central Universities, HUST: No. 2013QN119.

References

[1] C.C. Aggarwal, On unifying privacy and uncertain data models, in: Proc. of the 24th International Conference on Data Engineering (ICDE), IEEE, 2008, pp. 386–395.
[2] R. Agrawal, R. Srikant, Privacy-preserving data mining, ACM Sigmod Rec. 29 (2) (2000) 439–450.
[3] M.E. Ali, E. Tanin, R. Zhang, R. Kotagiri, Probabilistic voronoi diagrams for probabilistic moving nearest neighbor queries, Data Knowl. Eng. 75 (2012) 1–33.
[4] J.v. Bercken, B. Blohsfeld, J.-P. Dittrich, J. Krämer, T. Schäfer, M. Schneider, B. Seeger, Xxl-a library approach to supporting efficient implementations of advanced database queries, in: Proc. of the Conference on Very Large Databases (VLDB), 2001, pp. 39–48.
[5] T. Brinkhoff, A framework for generating network-based moving objects, GeoInformatica 6 (2) (2002) 153–180.
[6] J. Chen, Y. Chin, A concurrency control algorithm for nearest neighbor query, Inf. Sci. 114 (1) (1999) 187–204.
[7] R. Cheng, J. Chen, M. Mokbel, C.-Y. Chow, Probabilistic verifiers: evaluating constrained nearest-neighbor queries over uncertain data, in: Proc. of the 24th International Conference on Data Engineering (ICDE), IEEE, 2008, pp. 973–982.
[8] R. Cheng, D.V. Kalashnikov, S. Prabhakar, Querying imprecise data in moving object environments, IEEE Trans. Knowl. Data Eng. 16 (9) (2004) 1112–1127.
[9] R. Cheng, X. Xie, M.L. Yiu, J. Chen, L. Sun, Uv-diagram: a voronoi diagram for uncertain data, in: Proc. of the 26th International Conference on Data Engineering (ICDE), IEEE, 2010, pp. 796–807.
[10] H.-J. Cho, C.-W. Chung, An efficient and scalable approach to cnn queries in a road network, in: Proc. of the 31st International Conference on Very Large Data Bases (VLDB), VLDB Endowment, 2005, pp. 865–876.
[11] J. Derrac, S. García, F. Herrera, Fuzzy nearest neighbor algorithms: taxonomy, experimental analysis and prospects, Inf. Sci. 260 (2014) 98–119.
[12] W. Evans, J. Sember, Guaranteed voronoi diagrams of uncertain sites, in: Proc. of the 20th Canadian Conference on Computational Geometry, 2008.
[13] Y. Gao, B. Zheng, G. Chen, Q. Li, C. Chen, G. Chen, Efficient mutual nearest neighbor query processing for moving object trajectories, Inf. Sci. 180 (11) (2010) 2176–2195.
[14] C.H. Hamilton, A. Rau-Chaplin, Compact hilbert indices: space-filling curves for domains with unequal side lengths, Inf. Proc. Lett. 105 (5) (2008) 155–163.
[15] M. Hasan, M.A. Cheema, X. Lin, Y. Zhang, Efficient construction of safe regions for moving knn queries over dynamic datasets, in: Advances in Spatial and Temporal Databases, Springer, 2009, pp. 373–379.
[16] D. Hilbert, Ueber die stetige abbildung einer line auf ein flächenstück, Math. Ann. 38 (3) (1891) 459–460.
[17] E.G. Hoel, H. Samet, Efficient processing of spatial queries in line segment databases, in: Advances in Spatial Databases, Springer, 1991, pp. 235–256.
[18] X. Huang, C.S. Jensen, S. Šaltenis, The islands approach to nearest neighbor querying in spatial networks, in: Advances in Spatial and Temporal Databases, Springer, 2005, pp. 73–90.
[19] Y.-K. Huang, Z.-W. Chen, C. Lee, Continuous k-nearest neighbor query over moving objects in road networks, in: Advances in Data and Web Management, Springer, 2009, pp. 27–38.
[20] H. Jagadish, On indexing line segments, in: Proceedings of the 16th International Conference on Very Large Data Bases, 1990, pp. 614–625.
[21] C.S. Jensen, J. Kolářvr, T.B. Pedersen, I. Timko, Nearest neighbor queries in road networks, in: Proc. of the 11th ACM International Symposium on Advances in Geographic Information Systems, ACM, 2003, pp. 1–8.
[22] Y. Jing, L. Hu, W.-S. Ku, C. Shahabi, Authentication of k nearest neighbor query on road networks, IEEE Trans. Knowl. Data Eng. (TKDE) 26 (6) (2014) 1494–1506.
[23] M. Jooyandeh, A. Mohades, M. Mirzakhah, Uncertain voronoi diagram, Inf. Proc. Lett. 109 (13) (2009) 709–712.
[24] H. Jung, Y.D. Chung, L. Liu, Processing generalized k-nearest neighbor queries on a wireless broadcast stream, Inf. Sci. 188 (0) (2012) 64–79.
[25] I. Kamel, C. Faloutsos, Hilbert r-tree: an improved r-tree using fractals, in: Proceedings of the 20th International Conference on Very Large Data Bases (VLDB), 1994, pp. 500–509.
[26] M. Kolahdouzan, C. Shahabi, Voronoi-based k nearest neighbor search for spatial network databases, in: Proc. of the Thirtieth international conference on Very Large Data Bases (VLDB), VLDB Endowment, 2004, pp. 840–851.
[27] M.R. Kolahdouzan, C. Shahabi, Continuous k-nearest neighbor queries in spatial network databases, in: STDBM, Citeseer, 2004, pp. 33–40.
[28] H.-P. Kriegel, P. Kunath, M. Renz, Probabilistic nearest-neighbor query on uncertain objects, in: Advances in Databases: Concepts, Systems and Applications, Springer, 2007, pp. 337–348.
[29] F. Li, Real Datasets for Spatial Databases: Road Networks and Points of Interest, 2005. <http://www.cs.fsu.edu/~lifeifei/SpatialDataset.htm>.
[30] Y. Li, J. Li, L. Shu, Q. Li, G. Li, F. Yang, Searching continuous nearest neighbors in road networks on the air, Inf. Syst. 42 (0) (2014) 177–194.
[31] S.-Y. Lin, C.-S. Chen, L. Liu, C.-H. Huang, Tensor product formulation for hilbert space-filling curves, in: Proc. of the International Conference on Parallel Processing, IEEE, 2003, pp. 99–106.
[32] L. Liu, S. Yang, D. Wang, Force-imitated particle swarm optimization using the near-neighbor effect for locating multiple optima, Inf. Sci. 182 (1) (2012) 139–155.
[33] K. Mouratidis, M.L. Yiu, D. Papadias, N. Mamoulis, Continuous nearest neighbor monitoring in road networks, in: Proc. of the 32nd International Conference on Very Large Data Bases (VLDB), VLDB Endowment, 2006, pp. 43–54.
[34] J. Nievergelt, H. Hinterberger, K.C. Sevcik, The grid file: an adaptable, symmetric multikey file structure, ACM Trans. Database Syst. (TODS) 9 (1) (1984) 38–71.
[35] S. Nutanong, R. Zhang, E. Tanin, L. Kulik, The v∗-diagram: a query-dependent approach to moving knn queries, in: Proc. of the International Conference on Very Large Data Bases (VLDB), VLDB Endowment, 2008, pp. 1095–1106.
[36] A. Okabe, B. Boots, K. Sugihara, S.N. Chiu, Spatial Tessellations: Concepts and Applications of Voronoi Diagrams, John Wiley & Sons, 2009.
[37] A. Okabe, T. Satoh, T. Furuta, A. Suzuki, K. Okano, Generalized network voronoi diagrams: concepts, computational methods, and applications, Int. J. Geograph. Inf. Sci. 22 (9) (2008) 965–994.
[38] D. Papadias, J. Zhang, N. Mamoulis, Y. Tao, Query processing in spatial network databases, in: Proc. of the 29th International Conference on Very Large Data Bases (VLDB), VLDB Endowment, 2003, pp. 802–813.
[39] N. Roussopoulos, S. Kelley, F. Vincent, Nearest neighbor queries, ACM Sigmod Rec. 24 (2) (1995) 71–79.

[40] M. Safar, Enhanced continuous knn queries using pine on road networks, in: Proc. of the 1st International Conference on Digital Information Management, IEEE, 2006, pp. 248–256.
[41] M. Sharifzadeh, C. Shahabi, Vor-tree: R-trees with voronoi diagrams for efficient processing of spatial nearest neighbor queries, Proc. VLDB Endowment 3 (1–2) (2010) 1231–1242.
[42] A.P. Sistla, O. Wolfson, S. Chamberlain, S. Dao, Querying the uncertain position of moving objects, in: Temporal Databases: Research and Practice, Springer, 1998, pp. 310–337.
[43] Z. Song, N. Roussopoulos, K-nearest neighbor search for moving query point, in: Advances in Spatial and Temporal Databases, Springer, 2001, pp. 79–96.
[44] H. Wang, R. Zimmermann, Location-based query processing on moving objects in road networks, in: Proc. of the International Conference on Very Large Data Bases (VLDB), 2007, pp. 321–332.
[45] X. Xiong, M.F. Mokbel, W.G. Aref, Sea-cnn: scalable processing of continuous k-nearest neighbor queries in spatio-temporal databases, in: Proc. of the 21st International Conference on Data Engineering (ICDE), IEEE, 2005, pp. 643–654.
[46] S. Yi, H. Ryu, J. Son, Y.D. Chung, View field nearest neighbor: a novel type of spatial queries, Inf. Sci. 275 (0) (2014) 68–82.
[47] J. Zhang, M. Zhu, D. Papadias, Y. Tao, D.L. Lee, Location-based spatial queries, in: Proceedings of ACM SIGMOD International Conference on Management of Data, ACM, 2003, pp. 443–454.
[48] B. Zheng, J. Xu, W.-C. Lee, L. Lee, Grid-partition index: a hybrid method for nearest-neighbor queries in wireless location-based services, Int. J. Very Large Data Bases 15 (1) (2006) 21–39.