Local Network Voronoi Diagrams

Sarana Nutanong[†], Egemen Tanin[†‡], Mohammed Eunus Ali[†], Lars Kulik[†‡]

[†]Department of Computer Science and Software Engineering
University of Melbourne, Victoria, Australia
[‡]NICTA Victoria Research Laboratory, Australia
{sarana,egemen,eunus,lars}@csse.unimelb.edu.au

ABSTRACT

Continuous queries in road networks have gained significant research interests due to advances in GIS and mobile computing. Consider the following scenario: *"A driver uses a networked GPS navigator to monitor five nearest gas stations in a road network."* The main challenge of processing such a moving query is how to efficiently monitor network distances of the k nearest and possible resultant objects. To enable result monitoring in real-time, researchers have devised techniques which utilize precomputed distances and results, e.g., the *network Voronoi diagram (NVD)*. However, the main drawback of preprocessing is that it requires access to all data objects and network nodes, which means that it is not suitable for large datasets in many real life situations. The best existing method to monitor kNN results without precomputation relies on executions of snapshot queries at network nodes encountered by the query point. This method results in repetitive distance evaluation over the same or similar sets of nodes. In this paper, we propose a method called the *local network Voronoi diagram (LNVD)* to compute query answers for a small area around the query point. As a result, our method requires neither precomputation nor distance evaluation at every intersection. According to our extensive analysis and experimental results, our method significantly outperforms the best existing method in terms of data access and computation costs.

Keywords

Nearest neighbor, Spatial databases, Query processing.

1. INTRODUCTION

Advances in GIS and mobile computing have led to research interests in continuous spatial queries in road networks. An application scenario is: "a driver who uses a GPS navigator to continually query the nearest gas station with respect to the moving query location." The main challenge of performing such a task in real time is that the user's movement may result in changes in the shortest paths from the user to the surrounding nodes. These changes cause reevaluation of network distances, which is a costly operation in terms of graph traversal [7, 14].

ACM GIS '10, November 2-5 2010. San Jose, CA, USA

To facilitate monitoring of the nearest neighbor (NN) with respect to a moving query point, research in this area largely focuses on techniques which utilize precomputed network distances and/or partial results. One common example of such techniques is the *network Voronoi diagram (NVD)* [10, 11]. The NVD of a set $\mathcal{D}$ of data objects consists of $|\mathcal{D}|$ cells (i.e., collections of road segments) such that every location in a cell shares the same nearest object in $\mathcal{D}$. Figure 1 shows the NVD of four objects $\boldsymbol{a}$, $\boldsymbol{b}$, $\boldsymbol{c}$ and $\boldsymbol{d}$ on a road network. As shown in the figure, the road network is decomposed into four cells $\text{VC}(\boldsymbol{a})$, $\text{VC}(\boldsymbol{b})$, $\text{VC}(\boldsymbol{c})$ and $\text{VC}(\boldsymbol{d})$ corresponding to $\boldsymbol{a}$, $\boldsymbol{b}$, $\boldsymbol{c}$ and $\boldsymbol{d}$, respectively. Assume that a user at the location $\boldsymbol{q}$ is searching for the nearest neighbor. Using the NVD technique, we can see that $\boldsymbol{q}$ is in the Voronoi cell $\text{VC}(\boldsymbol{c})$ (the grey region), hence $\boldsymbol{c}$ is the NN of $\boldsymbol{q}$. The NN remains unchanged as long as $\boldsymbol{q}$ remains inside the same region.

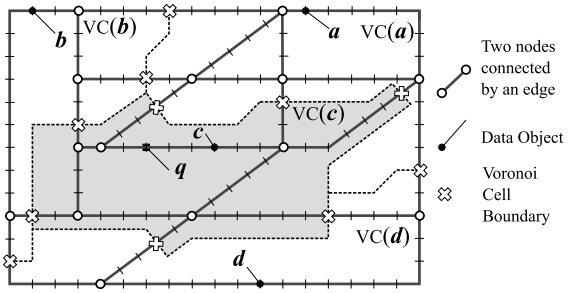


Figure 1: Network Voronoi diagram of a dataset $\{\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}, \boldsymbol{d}\}$

A major drawback of this technique is that it requires access to all objects and network nodes in the data space. In a large network, the practicality of this technique lies in an assumption that construction of an NVD can be done offline. The offline-construction assumption may not hold in navigation applications where the dataset $\mathcal{D}$ contains different kinds of objects and users submit object preferences during runtime. For example, a driver searching nearby gas stations that offer bio-diesel and accept petrol discount cards would require an NVD to be built from that specific subset of $\mathcal{D}$. Another case is when a navigation application uses the travel time derived from the current traffic condition as the proximity measure. In such an application, changes in the traffic condition will invalidate precomputed NVDs. When offline-construction is inapplicable, the best existing method [1] is to execute snapshot kNN queries at network nodes encountered by the query point. As mentioned earlier, a drawback of this method is the high graph traversal cost.

To alleviate both drawbacks, we propose a technique called the *local network Voronoi diagram* (LNVD). Specifically, an LNVD is constructed by computing the NVD of a subspace around the query point. The main novelty of our approach is a method to determine

upon which part of the LNVD can be relied to produce accurate results. No graph traversal operation is required as long as the query point remain in this part. Consequently, our method requires neither offline-construction nor repetitive graph traversal operations.

The contributions of this paper can be summarized as follows.

- We propose a technique, called LNVD, to locally compute an NVD. Our technique is the first to address the problem of locally computing a network Voronoi diagram.
- We generalize the LNVD technique to the order-k LNVD (kLNVD) to support monitoring of k nearest objects and provide proof of correctness for the technique.
- We conduct a complexity analysis and show the construction time with respect to the value of k and the number n of nodes in the kLNVD. We also show that kLNVD has a lower overall cost than the best competitor [1].
- We conduct extensive performance evaluations and our results show our proposed technique significantly outperforms the best competitor.

2. PROBLEM FORMULATION

We study the problem of road-network nearest neighbor monitoring for a moving query point and a static dataset. A road network can be represented as a graph G, i.e., a set N of nodes (junctions) and a set E of edges (road segments). The weight of an edge EDGE($\boldsymbol{n_i}$,$\boldsymbol{n_j}$) denotes the cost of travelling from node $\boldsymbol{n_i}$ to node $\boldsymbol{n_j}$, e.g., the distance or time. Edge weights can only assume non-negative values. For any given two locations $\boldsymbol{v_1}$ and $\boldsymbol{v_2}$ on the graph G, DIST($\boldsymbol{v_1}, \boldsymbol{v_2}$) denotes the cost of traversing PATH($\boldsymbol{v_1}$,$\boldsymbol{v_2}$), the path connecting $\boldsymbol{v_1}$ to $\boldsymbol{v_2}$ that minimizes the travelling cost. Since in practice a road network may consist of both one-way and two-way roads, G is modeled as a directed graph. That is, an edge in E can be unidirectional or bidirectional. In summary, G satisfies the following quasi-metric conditions: (i) *non-negativity*, (ii) *identity of indiscernibles*, and (iii) *triangle inequality*.

The k *nearest neighbor (kNN) query* [6] finds the nearest k data objects with respect to a given query point. A formal definition of the kNN query is given as follows.

DEFINITION 1 (k NEAREST NEIGHBOR (kNN) QUERY). *Given a set $\mathcal{D}$ of objects and a query point $\boldsymbol{q}$, the kNN query finds a list $\mathcal{A}$ of objects such that:*

(i) $\mathcal{A}$ contains k objects from $\mathcal{D}$;

(ii) for any object $\boldsymbol{x} \in \mathcal{A}$ and object $\boldsymbol{y} \in (\mathcal{D}-\mathcal{A})$, DIST($\boldsymbol{q}, \boldsymbol{x}$) $\leq$ DIST($\boldsymbol{q}, \boldsymbol{y}$)*;*

(iii) for any adjacent entries $\boldsymbol{x_i}$ and $\boldsymbol{x_{i+1}}$ in the list $\mathcal{A}$, DIST($\boldsymbol{q}, \boldsymbol{x_i}$) $\leq$ DIST($\boldsymbol{q}, \boldsymbol{x_{i+1}}$).

The *moving kNN (MkNN) query* is defined as follows.

DEFINITION 2 (MOVING kNN (MkNN) QUERY). *Given a set $\mathcal{D}$ of objects and a moving query point $\boldsymbol{q}$, the MkNN query finds a kNN result for every position of $\boldsymbol{q}$.*

3. RELATED WORK

Papadias et al. [12] proposed two approaches to processing spatial-network kNN queries, the *incremental network expansion (INE)* and *incremental Euclidean restriction (IER)*. INE-kNN is based on the Dijkstra's algorithm [4], i.e., incrementally expanding the search space along network edges until k nearest objects are found. IER-kNN finds a number k' of nearest neighbors in the Euclidean distance where k' is greater than or equal to k and calculates the network distances of these k' objects to obtain the actual spatial-network k NNs. Furthermore, an optimization in terms of data access and distance calculations to IER-kNN was proposed by Deng et al. [3]. In the same manner as the A* algorithm [13], their method uses the Euclidean distance measure to derive an upper bound value for pruning purposes.

Cho and Chung [1] proposed a continuous kNN technique that performs snapshot kNN queries at the intersections on the query path. Specifically, using one of the aforementioned snapshot kNN techniques [3, 12], continuous query answers can be produced by exploiting the fact that the movement of a user travelling in a spatial network is constrained by network connectivities. They showed that kNN results between any two nodes can be inferred from those of the nodes. The drawback of this approach is that it can incur repetitive distance calculations over similar sets of nodes.

Mouratidis et al. [9] studied a kNN monitoring problem in a dynamic environment where locations of the query point and data objects as well as edge weights can change. They proposed a method to handle this problem by keeping part of the kNN answer though incremental maintenance of the shortest path tree (SPT). Incremental SPT maintenance is suitable for changing edge weights and moving data objects, since such updates trigger only small changes to the SPT. However, using this method with moving query points still incurs frequent reevaluation of the kNN answer. This is because as the query point moves from the original location $\boldsymbol{q}$ to $\boldsymbol{q'}$, a vast majority of the shortest path tree (with the root at $\boldsymbol{q}$) can become invalid with respect to the new location $\boldsymbol{q'}$.

Both Dijkstra's and the A* algorithms incur expensive graph traversal operations, this cost is accentuated with continuous queries. This is because, the shortest paths from a query point $\boldsymbol{q}$ to its surround nodes change as $\boldsymbol{q}$ moves from one node to another [14]. To address this issue, algorithms that utilize precomputed distances or partial results have been proposed. One classic example is the *network Voronoi diagram (NVD)* [10, 11]. In a spatial network, the NVD of a set $\mathcal{D}$ of data objects consists of regions (i.e., collections of road segments) and each object in $\mathcal{D}$ is associated with a region/cell in which it dominates.

Kolahdouzan and Shahabi proposed a technique called the *Voronoi-based network nearest neighbor (VN^3)* search [7] based in the first-order network Voronoi diagram. The technique can provide M1NN results via a simple lookup operation, while processing MkNN queries with a higher value of k can be done by examining neighboring regions. In addition, they also proposed continuous kNN algorithms [8] based on the VN^3 technique. An important drawback of these approaches [7, 10, 11] is that data objects are used in the precomputation which means that changes in the dataset or edge weights can render the NVD obsolete.

To remedy the dataset update problem, data objects are decoupled from the precomputation process. This decoupling is done by precomputing shortest paths between network nodes and storing the shortest path information in a hierarchical or grid-based data structure. Samet et al. [14] formulated a kNN algorithm that utilizes precomputed shortest path information stored in quadtrees. Specifically, for each node $\boldsymbol{x}$, the shortest path to every other node in the network is computed. Each edge EDGE($\boldsymbol{n_x}$,$\boldsymbol{n_y}$) (from $\boldsymbol{n_x}$ to an immediate neighbor $\boldsymbol{n_y}$) is associated with destination nodes $\boldsymbol{n_z}$ such that EDGE($\boldsymbol{n_x}$,$\boldsymbol{n_y}$) forms part of the shortest path PATH($\boldsymbol{x}$,$\boldsymbol{z}$). This creates a shortest path map of all nodes reachable by $\boldsymbol{n_x}$, where each destination node is categorized according to the edges to the immediate neighbors. For efficiency purposes, collections of destination nodes are stored as quadtree regions where each region contains only one node type.

Demiryurek et al. [2] proposed a grid-based solution to storing shortest path information. Each EDGE($\boldsymbol{n_x}$,$\boldsymbol{n_y}$) stores a boolean vector $\langle b_1, ..., b_c \rangle$ where b_i indicates whether EDGE($\boldsymbol{n_x}$,$\boldsymbol{n_y}$) forms

part of the shortest path from x to any node in Grid i. In addition to a kNN algorithm, they also proposed kNN monitoring techniques which utilize this shortest path information. Although these techniques [2, 14] are unaffected by dataset updates, other updates such as changes in edge weights may affect shortest paths and invalidate precomputed shortest path information.

4. PRELIMINARIES

In this section, we describe the *shortest path tree (SPT)* and *network Voronoi diagram (NVD)* concepts [11] which form a basis of our proposed method. The simplest form of an SPT is one with a single reference point, which can be obtained by executing the Dijkstra's algorithm [4]. Figure 2 illustrates how PATH(n_{12}, c) is calculated by incrementally discovering surrounding nodes according to their distances to c. In this example, n_9 is discovered first with a distance of 3 units. Next, n_8, n_5 and n_7 are discovered with distances of 5 (via c), 6 (via n_9) and 6 units (via n_8), respectively. The search halts when n_{12} is discovered with a distance of 8 units via n_9.

Figure 2 also shows that each of the nodes involved in the search is associated with a label (p, d, n_h) where p denotes the reference point, d denotes the shortest distance, and n_h denotes the node from which the shortest distance is derived. The shortest path from n_{12} to c can be obtained by recursively traversing the next hop n_h in the label (p, d, n_h) until 9 is reached. In this case, we obtain $\langle n_{12}, n_9, c\rangle$ as the shortest path. As displayed in Figure 2, a *shortest path tree* (where the reference point c is the root) is formed by linking the next hop n_h of all nodes involved in the calculation.

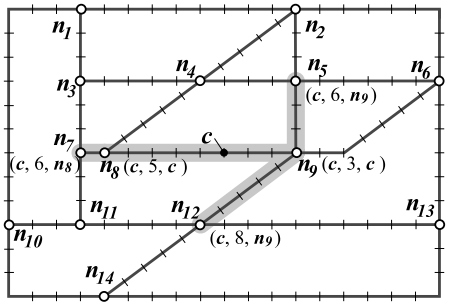


Figure 2: Calculation of DIST(n_{12}, c) via the Dijkstra's algorithm where each node (involved in the computation) is associated with a label (q, d, n_h).

The same incremental expansion principle can also be used to construct an NVD of a set $\mathcal{D}$ of data objects. The first step is to build an SPT with objects in $\mathcal{D}$ as reference points and incrementally expand the search space around each of these objects until all nodes are visited. By ensuring that nodes are visited according to the minimum distance to the objects in $\mathcal{D}$, each node is associated with its nearest object.

After the SPT of $\mathcal{D}$ is constructed, NVD cell boundaries are computed based on the information regarding the distance and NN obtained during the SPT construction. Specifically, a boundary point is computed for each edge where the two end nodes have different nearest objects. After the Voronoi cells boundaries are identified, the NN with respect to any query location q in the data space can be obtained by identifying the cell in which q resides. The main drawbacks of this method is that it requires: (i) global access to the data, where costs could be prohibitive; (ii) a preconstructed Voronoi diagram for each dataset. In the next section, we will show how such drawbacks can be mitigated by our proposed technique.

5. LOCALIZING VORONOI DIAGRAMS

This section presents our proposed technique, the *local network Voronoi diagram (LNVD)*, and how it can be used to process moving nearest neighbor queries. As exemplified in Figure 3, an LNVD is a Voronoi diagram of a subspace. In other words, we delimit the area of interest to road segments (edges), intersections (nodes) and data objects within a specific search region. As a result, construction of an LNVD requires only local information and by limiting the search region to a small, manageable size, an LNVD can be constructed on the fly. In this example, we set our search region to the region within a range of 14 units from the query point q. This search region (highlighted in grey) is denoted as SR($q, 14$). Based on information within SR($q, 14$), an LNVD is constructed. Since SR($q, 14$) contains three objects a, b and c, our LVND, in this case, has three local Voronoi cells where each is associated with its nearest neighbor (NN). The correctness of these NNs are guaranteed even under partial information.

An LNVD comprises local Voronoi cells of data objects within the search region. Each local Voronoi cell LVC(p) can be bounded by two bisector types:

(i) *internal*: a boundary between LVC(p) and one of its immediate neighbors known.

(ii) *external*: a boundary between a cell of a known neighbor and a region in which the search region does not provide further NN information.

As illustrated in Figure 3, the cell LVC(c) is bounded by three internal bisectors and three external bisectors. The LNVD in this example is the road segments covered by the three cells LVC(a), LVC(b) and LVC(c). As long as the query point q remains inside the LVND, the NN with respect to q can be obtained by identifying the cell in which q resides.

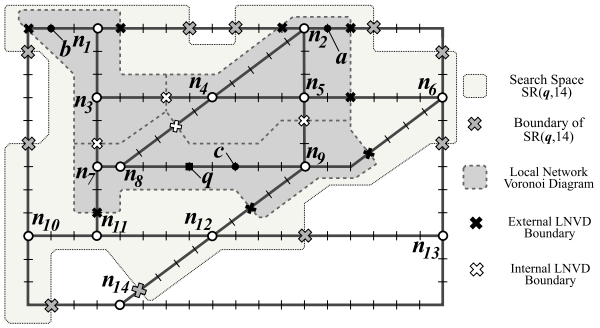


Figure 3: Local network Voronoi diagram with respect to a search region SR($q, 14$) and objects a, b and c

5.1 Data Retrieval

The first step to constructing an LNVD is to retrieve the data objects in the search region SR(q, r). We utilize the *incremental-network-expansion (INE) range query* [12] to retrieve objects and to calculate the network distances of nodes within the search region. For example, SR($q, 14$) in Figure 3 comprises nodes and edges in the grey region whose distances from q are less than or equal to 14 units. The boundaries that delimit this search space are shown as grey crosses in the figure.

5.2 Reliability of Local Information

Since we consider only a subset of the whole data space, the Voronoi diagram constructed based on this local information cannot be complete. In this subsection, we present a method to determine the reliability of a local NN result. The condition of an object being reliable with respect to a node n_i is formally defined as follows.

DEFINITION 3 (RELIABILITY OF A DATA OBJECT). *Given an object $\boldsymbol{p}$ inside $\text{SR}(\boldsymbol{q}, r)$ and $\boldsymbol{s}$ outside $\text{SR}(\boldsymbol{q}, r)$, $\boldsymbol{p}$ is reliable with respect to $\boldsymbol{n_i}$ if it is guaranteed that $\boldsymbol{s}$ is not closer to $\boldsymbol{n_i}$ than $\boldsymbol{p}$. That is, $\text{DIST}(\boldsymbol{n_i}, \boldsymbol{p})$ is less than or equal to a lower bound of $\text{DIST}(\boldsymbol{n_i}, \boldsymbol{s})$.*

We now describe how to derive $\text{INFIMUM}(\text{DIST}(\boldsymbol{n_i}, \boldsymbol{s}))$, the greatest lower bound of $\text{DIST}(\boldsymbol{n_i}, \boldsymbol{s})$. Since we only know that $\boldsymbol{s}$ is outside $\text{SR}(\boldsymbol{q}, r)$, $\text{INFIMUM}(\text{DIST}(\boldsymbol{n_i}, \boldsymbol{s}))$ is the MINIMUM DISTANCE (MINDIST) from $\boldsymbol{n_i}$ to the boundary set $\mathcal{B}$ of $\text{SR}(\boldsymbol{q}, r)$. That is,

$$\text{INFIMUM}(\text{DIST}(\boldsymbol{n_i}, \boldsymbol{s})) = \text{MINDIST}(\boldsymbol{n_i}, \mathcal{B}).$$

As a result, we need to rule out the region containing locations $\boldsymbol{v}$ such that $\boldsymbol{v}$ is closer to $\mathcal{B}$ than any object. This is because, an object outside $\text{SR}(\boldsymbol{q}, r)$ can be the NN with respect to $\boldsymbol{v}$. In other words, $\text{SR}(\boldsymbol{q}, r)$ does not pertain enough information to produce the NN with respect to $\boldsymbol{v}$. As shown in Figure 3, $\text{MINDIST}(\boldsymbol{n_{12}}, \mathcal{B})$ is 4 units while $\boldsymbol{c}$ as the nearest object in $\text{SR}(\boldsymbol{q}, r)$ produces a distance of 8 units. Hence, $\boldsymbol{n_{12}}$ does not have a guaranteed NN. A method to identify regions that have no guaranteed NN is presented in the next subsection.

5.3 Construction

The intuition behind our LNVD construction method is to identify the following two region types:

- *local Voronoi cell (LVC)*, a region with a guaranteed NN;
- *boundary-dominated cell (BDC)*, a region in which the NN cannot be inferred from information in $\text{SR}(\boldsymbol{q}, r)$.

We propose a method to incrementally label nodes based on their distances to its nearest object and the boundaries of $\text{SR}(\boldsymbol{q}, r)$. Specifically, we consider the boundary set $\mathcal{B}$ as one data object whose distance from a node $\boldsymbol{n_i}$ is given by $\text{MINDIST}(\boldsymbol{n_i}, \mathcal{B})$. By associating the closest object (or the boundary set $\mathcal{B}$) to each node in $\text{SR}(\boldsymbol{q}, r)$, we can identify edges $\text{EDGE}(\boldsymbol{n_i},\boldsymbol{n_j})$ such that $\boldsymbol{n_i}$ and $\boldsymbol{n_j}$ are associated with two different nearest objects (denoted as $\boldsymbol{p_i}$ and $\boldsymbol{p_j}$, respectively). The bisector between the cells of $\boldsymbol{p_i}$ and $\boldsymbol{p_j}$ on $\text{EDGE}(\boldsymbol{n_i},\boldsymbol{n_j})$ is the location equidistant to $\boldsymbol{p_i}$ and $\boldsymbol{p_j}$.

Algorithm 1 provides the steps of our proposed LVND construction method. We first define the network data structure used in the algorithm. A network/graph is represented using the adjacency-list format, i.e., a list of nodes where each node entry contains information regarding its adjacent nodes. The data structure NODE is defined as follows.

DEFINITION 4 (NODE). *The structure of a node $\boldsymbol{n_i}$ contains the following attributes:*

- ID: *the node identification.*
- Adjacency list (AdjList): *a list of edges to/from immediate neighbors and associated weights.*
- Label: *a label of $\boldsymbol{n_i}$ contains three attributes $(\boldsymbol{p}, d, \boldsymbol{n_h})$ where*
  - *$\boldsymbol{p}$ is the associated NN,*
  - *d represents $\text{DIST}(\boldsymbol{n_i}, \boldsymbol{p})$, and*
  - *$\boldsymbol{n_h}$ represents the next hop in order to reach $\boldsymbol{p}$.*
- Type: *a node type is "Labelable" by default and becomes "Permanent" after it is labelled.*

Another important data structure used in the construction process is the *priority queue*, which is defined as follows.

DEFINITION 5 (PRIORITY QUEUE). *A priority queue is a container of priority queue entries. Each priority queue entry comprises four attributes $(\boldsymbol{n}, \boldsymbol{p}, d, \boldsymbol{n_h})$. The first attribute $\boldsymbol{n}$ is a reference of the node to which the entry corresponds. The other three attributes $(\boldsymbol{p}, d, \boldsymbol{n_h})$ form a labeling candidate for $\boldsymbol{n}$. Entries in a priority queue are organized in such a way that an entry $(\boldsymbol{p}, d, \boldsymbol{n_h})$ with the smallest labelling distance d is always the head entry.*

Algorithm 1: Construct-LNVD($\mathcal{D}$, G, $\boldsymbol{q}$, r)

input : Dataset $\mathcal{D}$, Graph G, Query Point $\boldsymbol{q}$, Distance r
output: Graph G with labels and LVC boundaries

```
1  Initialize Priority Queue PQ;
2  A ← INE-RangeQuery(D, G, q, r);
3  B ← {v : DIST(q, v) = r};
4  for each (object p in A) do
5      Node n_p ← Create a network node from p;
6      G.Insert(n_p);
7      PQ.Insert(PQEntry (n_p, p, 0, −));
8  for each (node n_b adjacent to B) do
9      PQ.Insert(PQEntry (n_b, B, r − DIST(q, n_b), B));
10 while PQ is not empty do
11     PQEntry (n, p, d, n_h) ← PQ.DequeueHead();
12     if n.Type is Labelable then
13         n.Label ← (p, d, n_h);
14         n.Type ← Permanent;
15         for each Adjacent node n_a of n in SR(q, r) and n_a is
           Labelable do
16             Distance d_a ← d+ weight of EDGE(n_a,n);
17             PQ.Insert(PQEntry (n_a, p, d_a, n));
18 for each bidirectional edge EDGE(n_i,n_j) in SR(q, r) do
19     Check for external and internal bisectors on EDGE(n_i,n_j);
20 return G;
```

The LNVD construction steps in Algorithm 1 can be described as follows. The algorithm accepts four input parameters: the dataset $\mathcal{D}$, the graph G, the query point $\boldsymbol{q}$ and the range r. The construction process is divided into the following three stages.

Initialization (Lines 1 to 9): The first step is initialization of a priority queue PQ. Next we execute a range query to retrieve objects whose distances from $\boldsymbol{q}$ is less than or equal to r, and to calculate network distances of nodes within the range. Upon completion of the range query execution, we also identify the boundary set $\mathcal{B}$ of $\text{SR}(\boldsymbol{q}, r)$, i.e., a set of locations $\boldsymbol{v}$ such that $\text{DIST}(\boldsymbol{q}, \boldsymbol{v})$ is equal to r. (A method to determine an appropriate size for $\text{SR}(\boldsymbol{q}, r)$ is given in Section 8.1.) For each of the retrieved objects $\boldsymbol{p}$, we create a node entry $\boldsymbol{n_p}$, insert it into G and modify the affected edges accordingly. Next, a priority queue entry $(\boldsymbol{n_p}, \boldsymbol{p}, 0, -)$ is created and inserted into PQ. Note that the fourth attribute, next hop, is *null* ("−") since $\boldsymbol{n_p}$ is already at $\boldsymbol{p}$. For each node $\boldsymbol{n_b}$ adjacent to $\mathcal{B}$, we create an entry $(\boldsymbol{n_b}, \mathcal{B}, \text{MINDIST}(\boldsymbol{n_b}, \mathcal{B}), \mathcal{B})$, where $\text{MINDIST}(\boldsymbol{n_b}, \mathcal{B})$ is equal to $(r - \text{DIST}(\boldsymbol{q}, \boldsymbol{n_b}))$. The entry is then inserted into PQ.

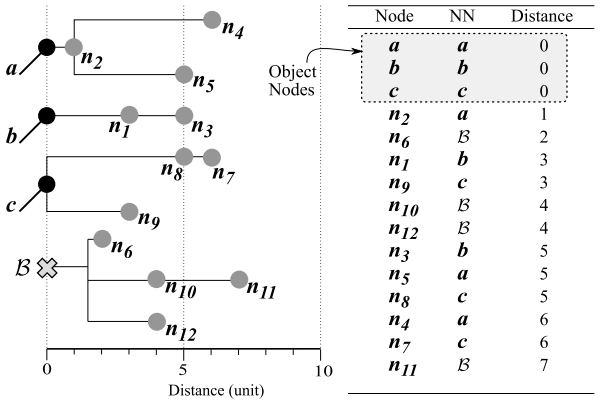
In the context of our running example (Figure 3), after the initialization steps, PQ has

- initial object entries of $(\boldsymbol{a}, \boldsymbol{a}, 0, -)$, $(\boldsymbol{b}, \boldsymbol{b}, 0, -)$ and $(\boldsymbol{c}, \boldsymbol{c}, 0, -)$;
- initial boundary entries of $(\boldsymbol{n_2}, \mathcal{B}, 3, \mathcal{B})$, $(\boldsymbol{n_1}, \mathcal{B}, 4, \mathcal{B})$, $(\boldsymbol{n_{10}}, \mathcal{B}, 4, \mathcal{B})$, and $(\boldsymbol{n_{12}}, \mathcal{B}, 4, \mathcal{B})$.

SPT construction (Lines 10 to 17): Our SPT construction process is conducted in a best-first manner. In particular, at each iteration of the *while loop*, the head entry $(\boldsymbol{n}, \boldsymbol{p}, d, \boldsymbol{n_h})$ is retrieved from PQ. Then, we check whether the corresponding node $\boldsymbol{n}$ can still be labelled. If so, the node label is set to $(\boldsymbol{p}, d, \boldsymbol{n_h})$ and the node type is changed to permanent. Next, for each node $\boldsymbol{n_a}$ in $\text{SR}(\boldsymbol{q}, r)$ that can reach $\boldsymbol{n}$ by a single hop, we calculate a labelling distance d_a (for $\boldsymbol{n_a}$) as the sum of d and the weight of $\text{EDGE}(\boldsymbol{n_a},\boldsymbol{n})$. The

while loop iterates until PQ is exhausted. An end result of this step is an SPT of objects in $\mathrm{SR}(\boldsymbol{q}, r)$ as generators. Each node in the SPT is associated with a reliable NN or the boundary set $\mathcal{B}$ as displayed in Figure 4.

Based on the example given in Figure 3, Figure 4 shows the order in which nodes are labelled during the SPT construction. The SPT is generated from $\{\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}\}$ and $\mathcal{B}$ as reference locations. The construction process starts from nodes with smallest distances to these reference locations and incrementally explore neighboring nodes until every node $\mathrm{SR}(\boldsymbol{q}, r)$ is associated with its nearest object.



Node	NN	Distance
$\boldsymbol{a}$	$\boldsymbol{a}$	0
$\boldsymbol{b}$	$\boldsymbol{b}$	0
$\boldsymbol{c}$	$\boldsymbol{c}$	0
$\boldsymbol{n_2}$	$\boldsymbol{a}$	1
$\boldsymbol{n_6}$	$\mathcal{B}$	2
$\boldsymbol{n_1}$	$\boldsymbol{b}$	3
$\boldsymbol{n_9}$	$\boldsymbol{c}$	3
$\boldsymbol{n_{10}}$	$\mathcal{B}$	4
$\boldsymbol{n_{12}}$	$\mathcal{B}$	4
$\boldsymbol{n_3}$	$\boldsymbol{b}$	5
$\boldsymbol{n_5}$	$\boldsymbol{a}$	5
$\boldsymbol{n_8}$	$\boldsymbol{c}$	5
$\boldsymbol{n_4}$	$\boldsymbol{a}$	6
$\boldsymbol{n_7}$	$\boldsymbol{c}$	6
$\boldsymbol{n_{11}}$	$\mathcal{B}$	7

Figure 4: Shortest path trees generated from $\{a, b, c\}$ and $\mathcal{B}$

Bisector calculation (Lines 18 to 19): After having created an SPT, each node is associated with its reliable NN or the boundary set $\mathcal{B}$. The next step is to compute external and internal bisectors by checking each bidirectional edge $\textsc{edge}(\boldsymbol{n_i},\boldsymbol{n_j})$ in $\mathrm{SR}(\boldsymbol{q}, r)$.[1]

- *Internal bisector.* An internal bisector is on an edge $\textsc{edge}(\boldsymbol{n_i},\boldsymbol{n_j})$ such that $\boldsymbol{n_i}$ and $\boldsymbol{n_j}$ are associated with two different NNs (assuming no boundaries on $\textsc{edge}(\boldsymbol{n_i},\boldsymbol{n_j})$). Let $\boldsymbol{p_i}$ and $\boldsymbol{p_j}$ be the NNs of $\boldsymbol{n_i}$ and $\boldsymbol{n_j}$ respectively. The internal bisector on $\textsc{edge}(\boldsymbol{n_i},\boldsymbol{n_j})$ is the location $\boldsymbol{v}$ such that

$$\textsc{Dist}(\boldsymbol{v}, \boldsymbol{p_i}) = \textsc{Dist}(\boldsymbol{v}, \boldsymbol{p_j}).$$

  Consider $\textsc{edge}(\boldsymbol{n_3},\boldsymbol{n_7})$ in Figure 3 as an example. The NN of $\boldsymbol{n_3}$ is $\boldsymbol{b}$ with the distance of 5 units and the NN of $\boldsymbol{n_7}$ is $\boldsymbol{c}$ with the distance of 6 units. The bisector between $\boldsymbol{b}$ and $\boldsymbol{a}$ on $\textsc{edge}(\boldsymbol{n_3},\boldsymbol{n_7})$ is the location 1 units from $\boldsymbol{n_3}$ and 2 units from $\boldsymbol{n_7}$. The location is 7 units to both $\boldsymbol{b}$ and $\boldsymbol{a}$.
- *External bisector.* An external bisector is on an edge $\textsc{edge}(\boldsymbol{n_i},\boldsymbol{n_j})$ such that $\boldsymbol{n_i}$ is associated with an NN $\boldsymbol{p_i}$ and $\boldsymbol{n_j}$ is associated with $\mathcal{B}$.[2] The external bisector on $\textsc{edge}(\boldsymbol{n_i},\boldsymbol{n_j})$ is the location $\boldsymbol{v}$ such that

$$\textsc{Dist}(\boldsymbol{v}, \boldsymbol{p_i}) = \textsc{MinDist}(\boldsymbol{v}, \mathcal{B}).$$

  Consider $\textsc{edge}(\boldsymbol{n_5},\boldsymbol{n_6})$ in Figure 3 as an example. The NN of $\boldsymbol{n_5}$ is $\boldsymbol{a}$ with the distance of 4 units while $\boldsymbol{n_6}$ is associated with $\mathcal{B}$ with $\textsc{MinDist}(\boldsymbol{n_6}, \mathcal{B})$ of 2 units. The bisector between $\boldsymbol{a}$ and $\mathcal{B}$ on $\textsc{edge}(\boldsymbol{n_5},\boldsymbol{n_6})$ is the location 2 units from $\boldsymbol{n_5}$ and 4 units from $\boldsymbol{n_6}$. The location is 6 units to both $\boldsymbol{n_b}$ and $\mathcal{B}$.

[1]If $\textsc{edge}(\boldsymbol{n_i},\boldsymbol{n_j})$ is unidirectional ($\boldsymbol{n_i} \rightarrow \boldsymbol{n_j}$), the NN of any location $\boldsymbol{v}$ between $\boldsymbol{n_i}$ and $\boldsymbol{n_j}$ is the same as that of $\boldsymbol{n_j}$, since we cannot travel to $\boldsymbol{n_i}$ without passing $\boldsymbol{n_j}$. This principle also applies to the order-k LNVD described in the next section.

[2]For the purpose of bisector calculations, we consider each boundary in $\mathcal{B}$ as a node with the $\textsc{MinDist}$ to $\mathcal{B}$ of 0 units.

Note that boundary calculation on one edge is independent from others. Hence one may choose to defer the computation until an edge is visited by the query point to avoid unnecessary boundary computation.

5.4 Query Processing

Processing a moving nearest neighbor query using the LNVD technique can be done as follows. Assume that the initial location of the query point is $\boldsymbol{q}$. We can use Algorithm 1 to construct an LNVD with respect to a search region $\mathrm{SR}(\boldsymbol{q}, r)$. After an LNVD is constructed, it can be used to provide the following information:

- the nearest neighbor with respect to any location $\boldsymbol{q'}$ within the LNVD by identifying the local Voronoi cell in which $\boldsymbol{q'}$ resides;
- the shortest path to the nearest neighbor using the shortest path tree.

If $\boldsymbol{q'}$ is outside the LNVD (i.e., having crossed an external bisector), we can again use Algorithm 1 to construct an LNVD with respect to the current query location $\boldsymbol{q'}$. It is also possible to derive a technique to make use of existing information to reduce the construction cost of the new LVND. Formulation of such a technique and its computational benefits (in comparison to constructing a new one from scratch) can be investigated as future work.

5.5 Proof of Correctness

This section shows that Algorithm 1 produces accurate results. First, we show that network distances calculated by the algorithm is correct in Lemma 1.

LEMMA 1 (CORRECTNESS OF DISTANCE CALCULATIONS). *For any location $\boldsymbol{v}$ in $\mathrm{SR}(\boldsymbol{q}, r)$, only nodes in $\mathrm{SR}(\boldsymbol{q}, r)$ are required to calculate (i) the distance from $\boldsymbol{v}$ to the set $\mathcal{B}$ of $\mathrm{SR}(\boldsymbol{q}, r)$ boundaries, and (ii) the distance from $\boldsymbol{v}$ to its its guaranteed NN $\boldsymbol{p}$ (if exists).*

PROOF. First, we show that a path from $\boldsymbol{v}$ to a boundary in $\mathcal{B}$ that includes a location $\boldsymbol{v'}$ outside $\mathrm{SR}(\boldsymbol{q}, r)$ can never be the path that minimizes the distance from $\boldsymbol{v}$ to $\mathcal{B}$. Since $\mathrm{SR}(\boldsymbol{q}, r)$ is fully enclosed by $\mathcal{B}$, in order to travel from $\boldsymbol{v}$ to $\boldsymbol{v'}$, one has to encounter a boundary $\boldsymbol{x}$ in $\mathcal{B}$. Since, $\textsc{Dist}(\boldsymbol{x}, \boldsymbol{v'})$ cannot assume a negative value, $\textsc{Dist}(\boldsymbol{v}, \boldsymbol{v'})$ can never be smaller than or equal to $\textsc{MinDist}(\boldsymbol{v}, \mathcal{B})$ and the calculation of $\textsc{MinDist}(\boldsymbol{v}, \mathcal{B})$ need not include any node outside $\mathrm{SR}(\boldsymbol{q}, r)$.

Second, we show that for each location $\boldsymbol{v}$ with a reliable NN of $\boldsymbol{p}$, $\textsc{path}(\boldsymbol{v},\boldsymbol{p})$ does not include any location $\boldsymbol{v'}$ outside $\mathrm{SR}(\boldsymbol{q}, r)$. According to Definition 3, for $\boldsymbol{p}$ to be reliable with respect to $\boldsymbol{v}$, the following condition has to be satisfied:

$$\textsc{Dist}(\boldsymbol{v}, \boldsymbol{p}) \leq \textsc{MinDist}(\boldsymbol{v}, \mathcal{B}).$$

Since $\boldsymbol{v'}$ is outside the boundaries of $\mathrm{SR}(\boldsymbol{q}, r)$,

$$\textsc{Dist}(\boldsymbol{v}, \boldsymbol{p}) \leq \textsc{MinDist}(\boldsymbol{v}, \mathcal{B}) < \textsc{Dist}(\boldsymbol{v}, \boldsymbol{v'}).$$

Hence, $\textsc{path}(\boldsymbol{v},\boldsymbol{p})$ can never include $\boldsymbol{v'}$.

As a result, we need to consider only nodes and edges in $\mathrm{SR}(\boldsymbol{q}, r)$ in order to calculate distances from a location $\boldsymbol{v}$ to its guaranteed NN or the boundary set $\mathcal{B}$. □

After having established that distances calculated by Algorithm 1 is correct, we are now ready to show that the NN associated to each local Voronoi cell is also correct in Lemma 2.

LEMMA 2 (CORRECTNESS OF ALGORITHM 1). *Given an object $\boldsymbol{p}$ inside a search space $\mathrm{SR}(\boldsymbol{q}, r)$, for any location $\boldsymbol{v}$ in $\textsc{lvc}(\boldsymbol{p})$, $\boldsymbol{p}$ is the nearest neighbor of $\boldsymbol{v}$.*

PROOF. Since LVC($\boldsymbol{p}$) is bounded by internal and external bisectors, any location $\boldsymbol{v}$ in LVC($\boldsymbol{p}$) satisfies the following two conditions. First, for all data objects $\boldsymbol{p'}$ inside SR($\boldsymbol{q}, r$),

$$\text{DIST}(\boldsymbol{v}, \boldsymbol{p}) \leq \text{DIST}(\boldsymbol{v}, \boldsymbol{p'}).$$

Second, $\boldsymbol{p}$ is reliable with respect to $\boldsymbol{v}$. That is, for all data objects $\boldsymbol{s}$ outside SR($\boldsymbol{q}, r$),

$$\text{DIST}(\boldsymbol{v}, \boldsymbol{p}) \leq \text{DIST}(\boldsymbol{v}, \boldsymbol{s}).$$

Since each data object is either inside or outside SR($\boldsymbol{q}, r$), $\boldsymbol{p}$ is guaranteed to be the nearest neighbor with respect to any given location $\boldsymbol{v}$ in LVC($\boldsymbol{p}$). □

6. EXTENDING TO ORDER-K

The earlier described LNVD concept can be generalized to the order-k LNVD (kLNVD). An kLNVD with respect to SR($\boldsymbol{q}, r$) consists of local Voronoi cells (LVCs), where each cell corresponds to a specific kNN result. As displayed in Figure 5, an kLVND (where k is 2) consists of four LVCs corresponding to the following four *rank-sensitive* kNN answers: $\langle \boldsymbol{a}, \boldsymbol{c} \rangle$, $\langle \boldsymbol{b}, \boldsymbol{c} \rangle$, $\langle \boldsymbol{c}, \boldsymbol{a} \rangle$ and $\langle \boldsymbol{c}, \boldsymbol{b} \rangle$. Similar to an order-1 LNVD, the boundaries of each LVC in an kLVND may comprise internal and external bisectors which are defined as follows.

- *Internal bisector (white cross).* A location where the query answer changes from one guaranteed kNN to another.
- *External bisector (black cross).* A location where the boundary set $\mathcal{B}$ has the same distance as the kth NN. That is, the kth NN is about to become unreliable.

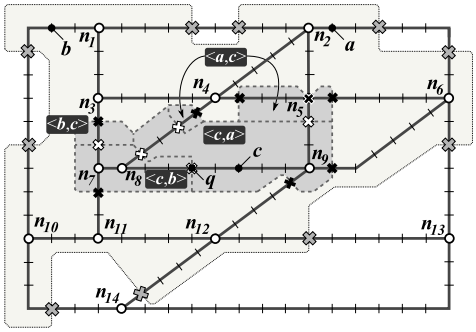


Figure 5: Order-2 local network Voronoi diagram with respect to a search region SR($\boldsymbol{q}, 14$) and objects $\boldsymbol{a}$, $\boldsymbol{b}$ and $\boldsymbol{c}$

6.1 Algorithm

Algorithm 2 provides the steps to construct a kLNVD. The algorithm is based on the same best-first graph traversal principle in Algorithm 1. In the same fashion as the order-1 LNVD where each network node is associated with its NN, each network node in a kLNVD is associated with its k NNs. Specifically, for each node $\boldsymbol{n}$, $\boldsymbol{n}$.Label is replaced by $\boldsymbol{n}$.LabelList (a list of labels). Hence, the labelling of $\boldsymbol{n}$ is done by inserting a label into $\boldsymbol{n}$.LabelList and $\boldsymbol{n}$ becomes permanent upon completion of k labels or inclusion of a boundary label.

The algorithm shares the same initialization steps (Lines 1 to 9) as Algorithm 1. For the SPT construction (Lines 10 to 18), we also utilize the best-first graph traversal principle using the priority queue PQ. At each iteration, the head entry $(\boldsymbol{n}, \boldsymbol{p}, d, \boldsymbol{n_h})$ is dequeued from PQ. The entry is added into $\boldsymbol{n}$.LabelList if (i) $\boldsymbol{n}$.Type is labelable, and (ii) there is no label with $\boldsymbol{p}$ as the associated NN

Algorithm 2: Construct-kLNVD($\mathcal{D}$, G, k, $\boldsymbol{q}$, r)

input : Dataset $\mathcal{D}$, Graph G, k, Query Point $\boldsymbol{q}$, Distance r
output: Graph G with labels and LVC boundaries

```
1  Initialize Priority Queue PQ;
2  A ← INE-RangeQuery(D, G, q, r);
3  B ← {v : DIST(q, v) = r};
4  for each (object p in A) do
5      Node n_p ← Create a network node from p;
6      G.Insert(n_p);
7      PQ.Insert(PQEntry (n_p, p, 0, −));
8  for each (node n_b adjacent to B) do
9      PQ.Insert(PQEntry (n_b, B, r − DIST(q, n_b), −));
10 while PQ is not empty do
11     PQEntry (n, p, d, n_h) ← PQ.DequeueHead();
12     if n.Type is Labelable and no existing label with p then
13         n.LabelList.Add(Label (p, d, n_h));
14         if n.LabelList.Length = k or Object p is the set B of
           SR(q, r) boundaries then
15             n.Type ← Permanent;
16         for each Adjacent node n_a of n in SR(q, r) and n_a is
           Labelable do
17             Distance d_a ← d+ weight of EDGE(n_a,n);
18             PQ.Insert(PQEntry (n_a, p, d_a, n));
19 for each bidirectional edge EDGE(n_i,n_j) in SR(q, r) do
20     Check for external and internal bisectors on EDGE(n_i,n_j);
21 return G;
```

in $\boldsymbol{n}$.LabelList. The node type is changed from labelable to permanent if there are k labels or an entry with $\mathcal{B}$ is inserted. For each adjacent node $\boldsymbol{n_a}$ in SR($\boldsymbol{q}, r$), a priority queue entry is created and is inserted into PQ. The *while loop* repeats until PQ is exhausted. As a result, each node $\boldsymbol{n_i}$ is associated with at most k unique reliable NNs, which are ranked according to the distances from $\boldsymbol{n_i}$. For each node $\boldsymbol{n_i}$ in SR($\boldsymbol{q}, r$), the number of associated NNs depends on the number of reliable NNs it has, which is determined by MINDIST($\boldsymbol{n_i}, \mathcal{B}$). For examples,

- $\boldsymbol{n_{10}}$ has no NN associated because $\mathcal{B}$ is closer to $\boldsymbol{n_{10}}$ than any object in SR($\boldsymbol{q}, r$);
- $\boldsymbol{n_1}$ is associated with only $\boldsymbol{b}$, because $\boldsymbol{b}$ is the only object with a distance smaller than MINDIST($\boldsymbol{n_1}, \mathcal{B}$);
- $\boldsymbol{n_5}$ has two NNs, $\boldsymbol{a}$ and $\boldsymbol{c}$, because both DIST($\boldsymbol{n_5}, \boldsymbol{a}$) and DIST($\boldsymbol{n_5}, \boldsymbol{c}$) are smaller than MINDIST($\boldsymbol{n_5}, \mathcal{B}$).

Internal and external bisectors can be calculated by categorizing the bidirectional edges EDGE($\boldsymbol{n_i}$,$\boldsymbol{n_j}$) into three types:

- *Both $\boldsymbol{n_i}$ and $\boldsymbol{n_j}$ have data objects as k NNs.* In this case, we can use the *split-point calculation* method [8] to identify locations that the kNN answer changes. All bisectors calculated in this case are internal, since there is no boundary set $\mathcal{B}$ involved in the calculations. As exemplified in Figure 6, the bisector BT($\boldsymbol{a}, \boldsymbol{c}$) separates the two LVCs, LVC($\langle \boldsymbol{a}, \boldsymbol{c} \rangle$) and LVC($\langle \boldsymbol{c}, \boldsymbol{a} \rangle$).

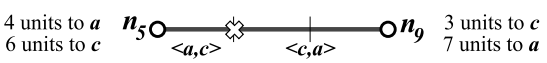


Figure 6: Both end nodes have 2 reliable data objects

- *One of the end nodes (namely $\boldsymbol{n_j}$) has the boundary set $\mathcal{B}$ in its label list.* We still use the split-point calculation method [8] to identify internal bisectors based on the NNs from the two end nodes. The only difference is that the calculation terminates when $\mathcal{B}$ replaces the kth nearest ob-

ject. That is, an external bisector $\boldsymbol{x}$ is found. By this means we guarantee that for any location $\boldsymbol{v}$ between $\boldsymbol{n_i}$ and $\boldsymbol{x}$, MINDIST$(\boldsymbol{v}, \mathcal{B})$ is always greater than or equal to the kth NN with respect to $\boldsymbol{v}$. Hence, all of the k NNs are reliable with respect to $\boldsymbol{v}$ (Definition 3). As exemplified in Figure 7, $\boldsymbol{n_7}$ is associated with 2 reliable objects $\boldsymbol{c}$ and $\boldsymbol{b}$, while $\boldsymbol{n_3}$ only has $\boldsymbol{b}$ as its only one reliable object. The edge EDGE($\boldsymbol{n_7}$,$\boldsymbol{n_3}$) has one internal bisector and one external bisector. The internal bisector BT$(\boldsymbol{c}, \boldsymbol{b})$ separates LVC$(\langle \boldsymbol{c}, \boldsymbol{b} \rangle)$ and LVC$(\langle \boldsymbol{b}, \boldsymbol{c} \rangle)$. The external bisector separates LVC$(\langle \boldsymbol{b}, \boldsymbol{c} \rangle)$ from the exterior of the LNVD.

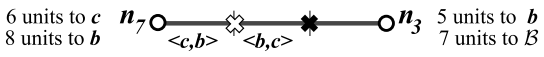


Figure 7: Only one end node has 2 reliable data objects

- *Both end nodes have the boundary set $\mathcal{B}$ in their label lists.* Such edges are ignored since neither end nodes have reliable k NNs.

As a result, each EDGE($\boldsymbol{n_i}$,$\boldsymbol{n_j}$) is decomposed into local Voronoi cells (LVCs) where each corresponds to a specific rank-sensitive kNN answer.

6.2 Proof of Correctness

Since network distances in Algorithm 2 are calculated in the same way as Algorithm 1, their correctness is also supported by Lemma 1. For the kNN results, we can extend Lemma 2 and its associated proof as follows.

LEMMA 3 (CORRECTNESS OF ALGORITHM 2). *Given an LVC with an associated kNN answer of $\langle \boldsymbol{p_1}, ..., \boldsymbol{p_k} \rangle$, for any location $\boldsymbol{v}$ the LVC, $\langle \boldsymbol{p_1}, ..., \boldsymbol{p_k} \rangle$ are the rank-sensitive kNN answer with respect to $\boldsymbol{v}$.*

PROOF. The function of internal and external bisectors of kLVND is similar to those of order-1 LNVD, so we extend the proof for Lemma 2 as follows. For any location $\boldsymbol{v}$ in an LVC of a kNN list $\langle \boldsymbol{p_1}, ..., \boldsymbol{p_k} \rangle$, internal and external bisectors ensures the following.

- First, internal bisectors ensure that the kNN list is sorted according to the distances to $\boldsymbol{v}$, and for any object $\boldsymbol{p'}$ in SR$(\boldsymbol{q}, r)$ but not in the kNN list, $\boldsymbol{p'}$ is not closer to $\boldsymbol{v}$ than any object in the kNN list.
- Second, external bisectors ensure that all of objects in the kNN list are reliable with respect to $\boldsymbol{v}$. That is, there is no object outside SR$(\boldsymbol{q}, r)$ closer to $\boldsymbol{v}$ than any object in $\langle \boldsymbol{p_1}, ..., \boldsymbol{p_k} \rangle$.

As a result, $\langle \boldsymbol{p_1}, ..., \boldsymbol{p_k} \rangle$ is guaranteed as the rank-sensitive kNN answer for all locations $\boldsymbol{v}$ in LVC$(\boldsymbol{p})$. □

7. COMPLEXITY ANALYSIS

In this section, we analyze the time complexity of the kLVND construction algorithm and the best competitor by Cho and Chung [1]. For conciseness, we call their method "CC-kNN". We assume that the number of intersections is much greater than the number of data objects as it is the case in existing literature [11, 12].

7.1 Proposed Method: Order-k LNVD

We now derive the complexity of Algorithm 2 as follows. The total cost can be considered as the sum of the costs of the following steps: (i) execution of a range query and insertion of initial entries into the priority queue; (ii) the main kLNVD construction loop.

In Sections 5 and 6, for exposition purposes, a search region is expressed as SR$(\boldsymbol{q}, r)$, where $\boldsymbol{q}$ denotes the search region's centroid and r denotes the search range. In other words, $\boldsymbol{q}$ determines the location and r determines the size of the search region. In our analysis, we use the number n of nodes inside the search region instead of r to represent the size of the search region. This is because, n has a more easy to analyze, observable effect on the processing cost than r. Note that the same INE range query concept can still be applied to discover n nearest nodes around $\boldsymbol{q}$. That is, the search terminates when n nearest nodes around the query point is discovered instead of when the distance r is reached. In this case, the value of r is determined by n. That is, r is the distance from $\boldsymbol{q}$ to the farthest node of the n nodes in the search space.

We first derive the range query cost. Assume that the Dijkstra's algorithm is used in the retrieval process to compute the distances and shortest paths of nodes in the search region. We obtain $\mathcal{O}(e + n \log n)$ as the cost of this operation, where e denotes the number of edges in the search region [5]. In a planar graph, e is proportional to the product of n and the degree of the graph, which is a small constant. Therefore, we have $\mathcal{O}(n \log n)$ as the range query cost in our setting. The next operations are insertions of the generators in $\mathcal{A}$ and nodes adjacent to the boundary set $\mathcal{B}$ into the priority queue. The cost of each insertion operation (using the Fibonacci heap) is constant [5]. Therefore the cost of these insertion operations are $\mathcal{O}(|\mathcal{A}| + |\mathcal{B}|)$. Both $|\mathcal{A}|$ and $|\mathcal{B}|$ are much smaller than n, so it is dominated by the range query cost. Therefore, the cost of the first step is $\mathcal{O}(n \log n)$.

Next, the cost of constructing a kSPT can be derived from the number of iterations of the *while loop* (Lines 10 to 18) and the cost of priority queue operations (i.e., dequeue and insertion) for each iteration. The number of *while loop* iterations is the maximum number of priority queue entries, which can be derived as a product of three parameters: the number n of nodes, the number k of labels per node and the maximum number of priority queue entries that can be generated from each label.[3] The last parameter is equal to the degree d of the graph, i.e., the number of adjacent nodes. We have $(n \cdot k \cdot d)$ as the number of priority queue entries and the number of iterations.

For each iteration, the cost of a dequeue operation is logarithmic with respect to the priority queue size. We also use $(n \cdot k \cdot d)$ as the priority queue size. Hence, the total cost of the dequeue operations is $\mathcal{O}\left(n \cdot k \cdot d \cdot \log(n \cdot k \cdot d)\right)$. For the priority queue insertions, only at most $(n \cdot k)$ of priority queue entries would satisfy the condition in Line 12 and the *for loop* (Lines 16 to 18) iterates at most d times. Therefore we have $\mathcal{O}\left(n \cdot k \cdot d\right)$ as the total cost of the insertion operations, which is dominated by the dequeue operations. In a planar graph, the degree d is a small constant, so the total cost of the second step is $\mathcal{O}\left(kn \log kn\right)$.

The cost of bisector calculations is not included in the construction cost. This is because calculations of bisectors on an edge EDGE($\boldsymbol{n_i}$,$\boldsymbol{n_j}$) can be deferred until EDGE($\boldsymbol{n_i}$,$\boldsymbol{n_j}$) is visited by the query point. That is, only edges along the query trajectory have their bisectors calculated. As a result, the construction cost involves only two cost components: (i) $\mathcal{O}(n \log n)$ for the initialization, and (ii) $\mathcal{O}(kn \log kn)$. Since the latter dominates the former, we have

$$\mathcal{O}\left(kn \log kn\right)$$

as the total construction cost.

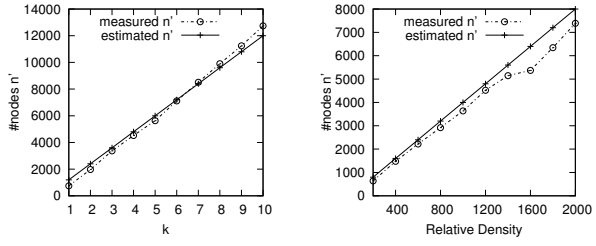
The value of k is the number of nearest neighbors reported to the user. The number n of nodes determines the search region size,

[3]Under the assumption that $(|\mathcal{A}| + |\mathcal{B}|)$ is much smaller than n, the $(|\mathcal{A}| + |\mathcal{B}|)$ initial priority queue entries can be neglected.

which has the two following effects. First, n has a positive correlation with the construction cost. Second, a greater value of n also means that we have a larger kLNVD, which in turn provides more kNN answers per each construction. Studies on these effects of the value of n are given in Section 8.1.

7.2 Best Existing Competitor

We now consider the competitor, CC-kNN [1], which executes the kNN query at each network node. Assume that the Dijkstra's algorithm [4] is used to discover the k NNs at each instance of the kNN query. The cost for each query instance can be given as $\mathcal{O}(n' \log n')$, where n' denotes the number of nodes visited by the Dijkstra's algorithm in order to obtain the k NNs. The number n' of visited nodes can be estimated as a product of the number k of objects and the relative density ρ of nodes with respect to objects (i.e., *the number of nodes* divided by *the number of objects*). Figure 8 provides an empirical support for this estimate $(k \cdot \rho)$ by comparing it with measured values of n'. The test was conducted on a road network dataset of North America with 175,813 nodes.



(a) k vs n' (b) ρ vs n'

Figure 8: The number n' of nodes involved in a kNN evaluation in comparison to the estimate $(k \cdot \rho)$ (Each result is the average from 20 query instances.)

We can therefore express the cost of each kNN retrieval as

$$\mathcal{O}\left(k\rho \log k\rho\right).$$

That is, the kNN cost is loglinear with respect to the product of k and ρ. This cost is incurred every time the query point encounters a new node.

7.3 Summary

For the kLVND, since the search space must include at least k objects, $(k \cdot n)$ is greater than $(k \cdot \rho)$. As a result, the kLVND cost of $\mathcal{O}\left(kn \log kn\right)$ is greater than the kNN cost. However, the kNN cost of $\mathcal{O}\left(k\rho \log k\rho\right)$ is incurred every time the query point encounters a new node, which can be disadvantageous in a setting with a high node density. In the next section, through experimental studies, we show that the order-k LNVD has a lower cost than the competitor, CC-kNN, when taking into account the fact that an LNVD contains kNN answers of multiple nodes.

8. EXPERIMENTAL STUDIES

In this section, we evaluate the performance of our proposed method in comparison to the best competitor, CC-kNN [1]. The experiments were conducted on a machine with an 2.66GHz Intel Core 2 Duo CPU, 4GB of main memory, and Linux 2.6.32 kernel as the operating system. All techniques were implemented on Java. We used a road network dataset of North America with 175,813 nodes and a diameter of 18,579 units. The experiments consist of two studies. In the first study, we present an empirical method to determine the size of a search region for the LNVD construction algorithm (Algorithm 2). In the second study, we compare the performance of our proposed method to that of the competitive one.

8.1 Determination of the Search Region Size

In the first experimental study, we present an empirical method to derive an appropriate size of the search region (SR) with different values of k and ρ. As mentioned earlier, the SR size can be varied by adjusting the number n of nodes it contains. The value of n is important to the performance of our kLNVD method. A greater n value provides a larger kLNVD but incurs a greater construction cost. If the n value is too small on the other hand, kLNVD may not contain enough complete nodes (nodes with a complete kNN answer) and results in frequent kLVND constructions. We use m to denote the number of complete nodes in a kLNVD.

In this experimental study we observe the effect of n on the processing cost t and the payoff, i.e., the number m of complete nodes. The processing cost (denoted as t) is the product of the following two measurements:

- the number of *while-loop* iterations in Algorithm 2, and
- the logarithm of the maximum priority queue size.

Our main objective is to find an SR size n that yields a reasonable cost-payoff ratio (t/m) for given values of k and ρ.

An appropriate search region size n depends on the k values, and the relative density ρ of nodes with respect to objects. Specifically, a greater value of k requires a greater value of n to counteract a stricter kNN completion requirement for each node, and a greater value of ρ requires a greater value of n to compensate for the reduced number of objects per node. As a result, we use $(n/\rho/k)$ as a normalized SR size to neutralize the effects of k and ρ. We varied the value of $(n/\rho/k)$ from 1 to 6 in this study. Experiments were conducted on k of 4 and 8 objects and ρ of 400 and 800 nodes per object.

Figure 9(a) shows the effect of $(n/\rho/k)$ on the processing cost t. We can see that t has a positive correlation with $(n/\rho/k)$, since an increase in n results in a greater number of iterations and a greater priority queue size. It can be seen that an increase in $(n/\rho/k)$ results in both an increase in the number m of complete nodes and the processing cost t. In order to find an appropriated value of n (with respect to fixed values of k and ρ), we display the processing cost per complete node, i.e., (t/m), in Figure 9(b). For all experimented values of k and ρ, (t/m) drastically reduced as $(n/\rho/k)$ increased from 1 to 2. An increase in the value of $(n/\rho/k)$ beyond 3 hardly improved the processing cost per complete node. In the interests of keeping the LNVD in a small and manageable size, we use 3 as our value of $(n/\rho/k)$. That is, we set the SR size such that the number n of nodes is

$$n = 3 \cdot \rho \cdot k. \tag{1}$$

Another measure we use to display the behavior of Algorithm 2 is the completion ratio $(m : n)$, i.e., the number of complete nodes divided by the number n of nodes inside the SR. Figure 9(c) displays the effect of $(n/\rho/k)$ on the completion ratio. The figure shows that the completion ratio increases in a diminishing rate as $(n/\rho/k)$ increases. The results from different values of k and ρ produce approximately the same curve. As a result, Equation 1 effectively sets the completion ratio to approximately 35%

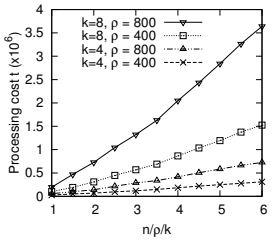
Since we set the number n of nodes inside SR to $(3 \cdot \rho \cdot k)$, we can express the kLNVD construction cost as

$$\mathcal{O}\left(\rho k^2 \log \rho k^2\right).$$

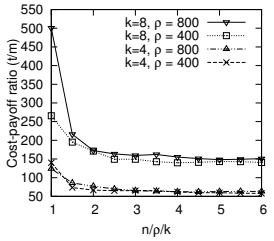
As earlier discussed, for this value of n, the number m of complete nodes is 35% of n, i.e., $m = (0.35 \cdot 3 \cdot \rho \cdot k)$. Hence, we can express the cost per complete node as

$$\mathcal{O}\left(k \log \rho k^2\right).$$

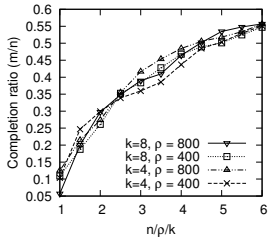
Since only some of the m complete nodes in SR are visited by the



(a) Processing cost t



(b) Processing cost per complete node (t/m)



(c) Completion ratio (m/n)

Figure 9: kLNVD construction costs and payoff

query point, a more accurate comparison can be done by measuring the performance of the two methods when used to process an actual moving kNN query. Specifically, the comparative cost of the two methods depends on the number n_q of nodes encountered by the query point per each time a kLNVD is constructed. For example, in a planar graph, we can assume that n_q is proportional to $\sqrt{m}$, which results in a cost per visited node of

$$\mathcal{O}\left(k\sqrt{\rho k}\log \rho k^2\right).$$

In comparison to the CC-kNN's processing cost of $\mathcal{O}\left(\rho k \log \rho k\right)$, we can see that kLNVD is more suitable to a setting of a small k value and high node density ρ. We argue that a setting of large ρ and small k is a valid assumption in actual application environments through the following reasons. First, a mobile user who is driving or walking can pay attention to only a small number k of resultant objects. For example, most navigation devices display only 5 to 10 NNs at a time. Second, a high number of nodes is required to produce an accurate model of a road network. For examples, eight nodes are required to model an intersection where right turns (assuming a left-hand traffic) are not permitted, and four nodes are required to model a road u-turn. To provide an insight into the behaviors of the two methods in a more realistic query processing environment, we report results of our experimental studies in the next section.

8.2 Performance Comparison

In this section, we study the performance of our proposed method in comparison to CC-kNN [1]. We used two trajectory types: *single destination (SD)* and *multiple destination (MD)*.

- An SD trajectory represents a user travelling along a shortest path from a start location and a destination, e.g., a user travelling interstate.
- An MD trajectory represents a case of a user changing their destination every 300 units, e.g., a delivery truck driver dropping off goods at multiple locations.

We generated 20 trajectories of each type and experimental results are reported as average of these trajectories. The trajectory length is fixed to 3,000 units, i.e., 1.61% of the network diameter. Experiments were conducted to study the effect of two parameters:

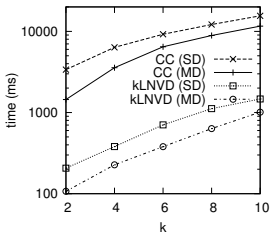
- *The k value*: ranged from 2 to 10 with a default of 6.
- *The density ρ of nodes with respect to objects*: ranged from 400 to 2,000 with a default of 1,200.

The aim of this set of experiments is to test the validity of the cost per visited node of the kLNVD, $\mathcal{O}\left(k\sqrt{\rho k}\log \rho k^2\right)$, and the cost of CC-$k$NN, $\mathcal{O}\left(\rho k \log \rho k\right)$, derived in Section 8.1 (also recall the analysis in Section 7.) The processing cost can be given as the product of the number of iterations and the cost per iteration. The cost per iteration is the logarithm of priority queue size. For kLNVD, $\mathcal{O}\left(k\sqrt{\rho k}\right)$ represents the number of iterations and $\mathcal{O}\left(\rho k^2\right)$ represents the priority queue size. For CC-kNN, both number of iterations and priority queue size are $\mathcal{O}\left(\rho k\right)$. As we vary the parameters k and ρ, we expect the behavior of both algorithms to conform with this analysis.

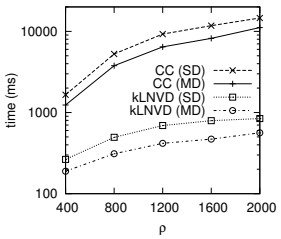
For each parameter, the two methods are compared using the following cost measures.

- *Total time:* the time taken to process a trajectory.
- *Processing cost:* the product of: (i) the number of iterations of the *while-loop* in Algorithm 2, and (ii) the logarithm of the maximum priority queue size.
- *Number of queries:* The number of times the NN query or the kLNVD construction method is executed.[4]

Total time. Figure 10 shows that the total times of both methods have positive correlations with ρ and k, which conform with the cost analysis given in Section 8.1. The figure also shows that SD trajectories are more expensive to process than MD, since MD trajectories are more likely to remain in the same area than SD. Our proposed method, kLNVD, consistently outperforms CC-kNN by approximately one order of magnitude. The *breakdown* of the total time is given in the next paragraph.



(a) k vs total time



(b) ρ vs total time

Figure 10: Effects of k and ρ on the total time

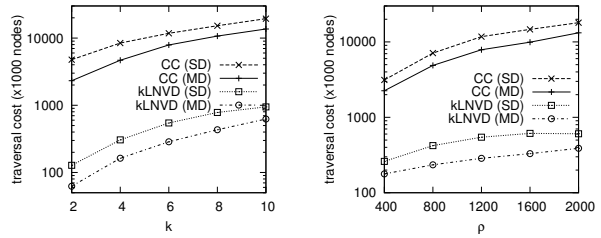
Processing cost. We define the processing cost as the product of the number of *while-loop* iterations and the cost per iteration (logarithm of the priority queue size). Figure 11 shows the processing cost and these two components with respect to different values of k and ρ. As suggested by the analysis, Figure 11(a) shows that an increase in k provides a more drastic effect on the number of iterations than CC-kNN and Figure 11(b) shows that an increase in ρ produces a drastic increase in the number of iterations of CC-kNN and a moderate increase for kLNVD. The number of iterations of kLNVD in both figures are significant lower than that of CC-kNN.

In terms of the priority queue size (Figures 11(c) and 11(d)), CC-kNN has a significantly smaller priority queue than kLNVD. This is because kLNVD construction involves more nodes and labels per node than CC-kNN. As suggested by the cost analysis, the priority queue size of kLNVD drastically increases as k increases, while an increase in ρ produces an increase in the same rate as CC-kNN.

The overall processing cost (Figures 11(e) and 11(f)) is approx-

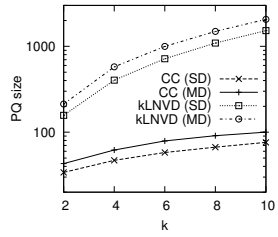
[4]This measure is indicative of the number of communications when data objects are retrieved from a remote location.

imately the product of the number of iterations (Figures 11(a) and 11(b)) and the logarithm of the priority queue size (Figures 11(c) and 11(d)). Hence, the number of iterations has a greater effect on the processing cost than the priority queue size. Figures 11(e) and 11(f) also shows that kLNVD significantly outperforms CC-kNN.

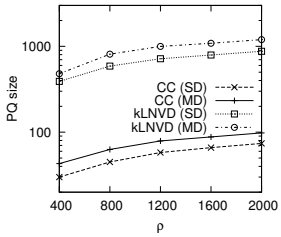


(a) k vs number of iterations

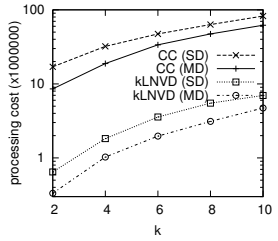
(b) ρ vs number of iterations



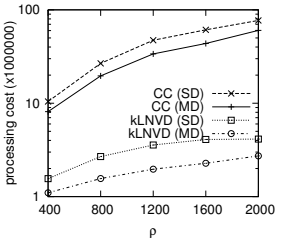
(c) k vs PQ size



(d) ρ vs PQ size



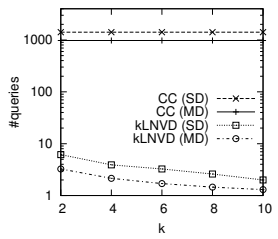
(e) k vs processing cost



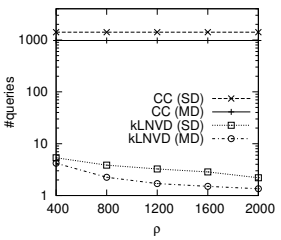
(f) ρ vs processing cost

Figure 11: Effects of k and ρ on the processing cost

Number of retrievals. The number of retrievals represents the number of queries executed to retrieve objects from the database, i.e., the number of kLNVD constructions, and the number of visited nodes (discounting the repeats) for CC-kNN. Figure 12 shows that the number of retrievals for kLNVD reduces as k and ρ increase, since the number m of complete nodes is given as $(0.35 \cdot 3 \cdot \rho \cdot k)$. The number of retrievals for CC-kNN is determined by the number of visited nodes. Hence, the parameters ρ and k have no effect on this measure. We can also see that kLNVD continues to outperforms CC-kNN significantly.



(a) k vs number of retrievals



(b) ρ vs number of retrievals

Figure 12: Effects of k and ρ the number of retrievals

8.3 Summary

We show that the number n of nodes in SR should be set to $(3 \cdot \rho \cdot k)$ to obtain a reasonable completion ratio (Figure 9(c)) and to keep the kLNVD size small and manageable. We continue to show that by setting n to this value, kLNVD is more scalable than CC-kNN as ρ increases and outperforms CC-kNN for all experimented values of k. We take advantage of the fact that the best-first priority queue processing cost is linear with respect to the number of iterations (traversal operations) and logarithmic with respect to the number of entries. Hence, by having a larger priority queue but less traversal operations, the overall processing cost is improved.

9. CONCLUSIONS

We have proposed a method to process *moving k nearest neighbor (MkNN)* queries which requires neither preprocessing nor repetitive distance evaluation. The complexity analysis and experimental results have shown that our proposed method, *order-k local network Voronoi diagram (kLNVD)* scales better than CC-kNN [1] as ρ increases. Specifically, kLNVD significantly outperforms CC-kNN in terms of the total time, traversal cost, processing cost and the number of data retrievals. As future work, we plan to apply the local computation of Voronoi diagrams to different domains, e.g., the Euclidean and other normed vector spaces.

Acknowledgement. This work is partially supported under the Australian Research Council's Discovery funding scheme (project number DP0880215).

10. REFERENCES

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

[2] U. Demiryurek, F. B. Kashani, and C. Shahabi. Efficient continuous nearest neighbor query in spatial networks using Euclidean restriction. In *SSTD*, pages 25–43, 2009.

[3] K. Deng, X. Zhou, H. T. Shen, S. W. Sadiq, and X. Li. Instance optimal query processing in spatial networks. *VLDB J.*, 18(3):675–693, 2009.

[4] E. W. Dijkstra. A note on two problems in connection with graphs. *Numeriche Mathematik*, 1:269–271, 1959.

[5] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987.

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

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

[8] M. R. Kolahdouzan and C. Shahabi. Alternative solutions for continuous k nearest neighbor queries in spatial network databases. *GeoInformatica*, 9(4):321–341, 2005.

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

[10] A. Okabe, B. Boots, K. Sugihara, and S. N. Chiu. *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams.* Wiley, Chichester, second edition, 2000.

[11] A. Okabe, T. Satoh, T. Furuta, A. Suzuki, and K. Okano. Generalized network Voronoi diagrams: Concepts, computational methods, and applications. *International Journal of Geographical Information Science*, 22(9):965–994, 2008.

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

[13] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, second edition, 2003.

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