Regular Paper

RkNN Query on Road Network Distances

Aye Thida Hlaing[1] Tin Nilar Win[1] Htoo Htoo[1] Yutaka Ohsawa[1,a)]

Received: June 23, 2014, Accepted: November 10, 2014

Abstract: Reverse k-nearest neighbor (RkNN) queries on road network distances require long processing times because most conventional algorithms require a k-nearest neighbor (kNN) search on every visited node. This causes a large number of node expansions; therefore, the processing time is drastically increased when data points are sparsely distributed. In this paper, we propose a fast RkNN search algorithm that runs using a simple materialized path view (SMPV). In addition, we adopt an incremental Euclidean restriction strategy for fast kNN queries, the main function in RkNN queries. The SMPV used in our proposed algorithm only constructs an individual partitioned subgraph; therefore, the amount of data is drastically reduced compared to conventional materialized path views (MPVs). According to our experimental results using real road network data, our proposed method achieved a processing time that was 100 times faster than conventional approaches when data points are sparsely distributed on a road network.

Keywords: location based services, RkNN, road network, distance materialization

1. Introduction

When a set of points P is given, a query to find the nearest neighbor of a point q ($\in P$) is called a nearest neighbor (NN) query. In **Fig. 1**, the values along the dotted line indicate the distances between two points. In this figure, the NN of A is B, and the NN of B is A. In this case, one NN is searched for; however, when a number k (an arbitrary number) of NNs are sought, the query is called a kNN query. Figure 1 (b) shows the 1NN and 2NN of each point.

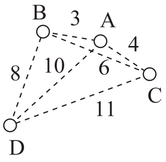
Conversely, when q ($\in P$) is the NN of p ($\in P$), p is called a reverse nearest neighbor (RNN) of q. The result of an RNN query is given as a set. For example, if A is the NN of B, then A is included in the RNN of B. To broaden the definition, when q is included in the kNN of p, q is called an RkNN of p. Figure 1 (c) shows the R1NN and R2NN of each point.

Generally, when a set of points P and a query point q ($q \in P$) are given, an RkNN query finds the points for which q is included in their kNN, i.e.,

$$\text{R}k\text{NN}(q) = \{p \in P | d(p, q) \leq d(p, p_k(p))\}$$

where $p_k(p)$ is the k-th NN of p, and $d(a, b)$ is the distance between two points, a and b.

This type of query is required in a wide variety of applications, including facility management, taxi allocation, location-based services, advertising distribution, and games; however, most existing algorithms work with Euclidean distance. In contrast, in location-based services (LBS) that use mobile phones or in-car navigation systems, queries based on road network distances are required. When a river or mountain lies between two points, the road network distance is the relevant parameter and substantially differs from the Euclidean distance. However, very limited research has focused on queries involving road network distances. Yiu et al. [1] proposed two algorithms applicable to the road network distance. However, these methods require long processing times, especially when points are sparsely distributed on the road network or when k is large.



(a)

	1NN	2NN
A	B	B,C
B	A	A,C
C	A	A,B
D	B	A,B

(b)

	R1NN	R2NN
A	{B,C}	{B,C,D}
B	{A,D}	{A,C,D}
C	∅	{A,B}
D	∅	∅

(c)

Fig. 1 Example of an NN and RNN query.

In this paper, we propose a fast RkNN query algorithm for road network distances using simple materialized path view (SMPV) data [2]. This algorithm runs on SMPV and refers to the SMPV tables to obtain the road network distances for pairs of terminal points. The algorithm presented in this paper adapts the SMPV in a manner suitable for RkNN queries. The proposed algorithm searches RkNN approximately 100 times faster than the conventional algorithms when points are sparsely distributed in the road network. The processing time is stable and independent of the point distribution density.

The rest of the paper is organized as follows. Related work is described in Section 2. In Section 3, the SMPV data structure and shortest path search algorithm on this structure are described. We also describe the principles for an RkNN query on road network distance and present the proposed method in Section 4. Experimental results are presented in Section 5, and we conclude our paper in Section 6.

2. Related Work

In this paper, we build upon the RkNN algorithm on a mate-

[1] Graduate School of Science and Engineering, Saitama University, Shimo-okubo 255, Sakura-ku, Saitama 338–8570, Japan

[a)] ohsawa@mail.saitama-u.ac.jp

rialized path view (MPV); therefore, in this section, we describe the relevant related work on both RkNN queries and MPVs.

Query algorithms for RkNN based on Euclidean distance have been actively studied. The RNN query and its corresponding algorithm were first proposed by Korn et al. [3]. Their RNN algorithm requires precomputed data, in which the distance from each point to its NN has been calculated. Given this data, a set of points and their distances to the NN are registered in an R-tree, and the circle centered at a data point with a radius equal to the distance to the NN is called its vicinity circle. The RNN of the query point q is found in the R-tree by searching the set of points for those whose vicinity circles overlap with q. However, this method is not suitable for an RkNN query because the R-tree is constructed using vicinity circles of predefined k-th NN distances, but the value of k in an RkNN query is usually determined when a query is issued. Therefore, the distance to the k-th NN cannot be determined when the structure is actually constructed.

Stanoi et al. [4] proposed an approach without precomputation called SAA. Tao et al. [5] proposed another efficient algorithm called TPL that recursively prunes the search space using the bisector between a query point q and its NN. These methods do not require any precomputation. Therefore, they are applicable to general RkNN queries, however, these efficient methods cannot be applied to RkNN queries of road network distances.

Yiu et al. [1] proposed the first RkNN algorithms applicable to road networks. The basic idea is that the area in which the RkNN exist is searched by gradually enlarging the search area using Dijkstra's shortest path algorithm. They proposed two algorithms (called the Eager and Lazy algorithms) that differ in their respective pruning methods. In these methods, the Eager algorithm searches RkNN significantly faster, especially when the value of k is small and the set of points are densely distributed. In other words, it is efficient when the search area is small. In contrast, for large k or sparsely distributed points, it requires long processing times because the Eager algorithm gradually enlarges the search area, similar to Dijkstra's algorithm, and the kNNs are searched at every visited road network node. To cope with this performance problem, Yiu et al. also proposed a path materialization method. In addition to the work of Yiu et al., Safer et al. [6] proposed algorithms using network Voronoi diagrams. Cheema et al. [7] proposed an RkNN algorithm for moving objects on a road network.

To obtain the road network distance between two nodes, several MPV approaches have been proposed. They retrieve the shortest path using a lookup query in a precomputed distance table. This method requires $O(n^2)$ space, with the number of nodes in the given graph represented as n. Therefore, several attempts have been dedicated to reducing the size of the data. Jing et al. [8] proposed a semi-materialized method in which only a portion of the shortest path routes are stored to reduce the amount of data. In particular, this approach only records the next node along the shortest path, and the entire shortest path route is restored by tracking the next visiting node in sequence. Samet et al. [9] reduced the data amount to $O(n^{1.5})$ by using a semi-materialized approach.

The shortest path can be retrieved quickly using an MPV; however, this approach suffers from the problem of requiring a huge memory space. Therefore, several hierarchical representation methods have been proposed to reduce the data requirements. For example, Jing et al. [8] proposed the hierarchical encoded path view (HEPV) using a hierarchical representation and semi-materialization approach. The principle behind this method is to partition a given graph G into several subgraphs SG_i. Distances between each pair of nodes are calculated to compose a locally materialized distance table. Next, by merging the neighboring subgraphs, the data structure constructs higher-level subgraphs in a stepwise fashion. At each higher level, the distance table is built only for the border nodes between subgraphs.

Hierarchical representations such as HEPV are suitable for the fast calculation of the shortest path between two points; however, the table size at the higher levels increases rapidly and the total memory size of this memory structure still becomes very large. Furthermore, when an edge weight is changed because of traffic accidents or road maintenance, changed weights (e.g., distances) impact a large portion of the distance table.

Jung et al. proposed another hierarchical MPV approach called HiTi graph [10], [11]. This method also materializes distances between pairs of nodes in the graph and constructs a hierarchy. The key difference between HiTi and HEPV is that HiTi does not materialize the leaf-level subgraphs. Therefore, the total data requirement of the HiTi graph is smaller than that of HEPV. HiTi prunes the hierarchical tree leaves using an A* algorithm. Shekhar et al. [12] analyzed hierarchical MPV in terms of the storage/computation-time tradeoffs.

3. Simple Materialized Path View

3.1 Data Structure

A road network is modeled as a directed graph $G(V, E, W)$, where V is the set of nodes (intersections), E is the set of edges (road segments), and W is the set of edge weights. A fragment $SG_i(V_i, E_i, W_i)$ of graph $G(V, E, W)$ is a partitioned subgraph, with $V_i \in V$, $E_i \in E$, and $W_i \in W$. If the endpoints of an edge $e_{jk} \in E_i$ are v_j and v_k, then $v_j \in V_i$ and $v_k \in V_i$. This subgraph is denoted as SG_i in the rest of the paper where there is no ambiguity.

Figure 2 shows an example of a road network graph, in which the small circles (black and white) are nodes and the lines are edges. This graph is partitioned into four subgraphs by dotted lines. In this partition, the nodes shown by the black dots belonging to at least two neighboring subgraphs, such nodes are called *border nodes*. On the other hand, the nodes shown by white cir-

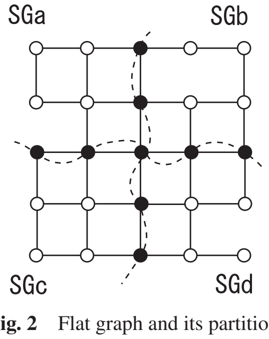


Fig. 2 Flat graph and its partition.

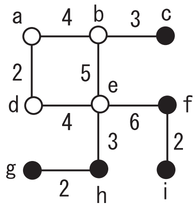


Fig. 3 SG_a in Fig. 2.

	c	f	g	h	i
c	0	14	13	11	16
f	14	0	11	9	2
g	13	11	0	2	13
h	11	9	2	0	11
i	16	2	13	11	0

Fig. 4 BBDT for SG_a.

	c	f	g	h	i
a	7	12	11	9	14
b	3	11	10	8	13
d	12	10	9	7	12
e	8	6	5	3	8

Fig. 5 IBDT for SG_a.

cles belonging to only one subgraph are called *inner nodes*. They are defined as following.

Definition 1 (Border Node) The nodes belong to plural subgraphs.

Definition 2 (Inner Node) The nodes belong to only one subgraph.

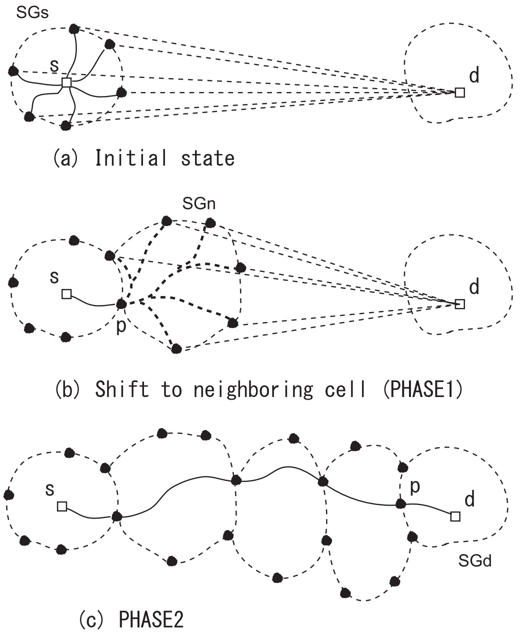
Two subgraphs are defined as being adjacent if they have at least one common border node. The set of border nodes of SG_i is denoted by BV_i. In this partition, each edge belongs to only one subgraph.

Figure 3 shows SG_a extracted from the graph of Fig. 2. The numerical value attached to each edge shows the weight of the edge, e.g., the length of the edge or the time required to travel along that edge. In the rest of this paper, we assume the weight is the length of the edge. The table shown in **Fig. 4** shows the shortest path length between every pair of border nodes of SG_a. The lengths are calculated by traveling inside the subgraph, therefore, these values are not always the globally shortest path lengths. If there is no connected path between a pair of nodes inside the subgraph, infinity is assigned. Although the matrix is symmetric in this example, it is not guaranteed to be symmetrical in a real road network because of the existence of one-way roads and obstructions or delays affecting only one direction of a two-way road. We refer to this table as a "border to border node distance table" (BBDT).

Figure 5 shows the "inner node to border node distance table" (IBDT) that lists the distance from an inner node to a border node. This table is used to retrieve the distance from an inner node as the starting point to a border node. Because the distance on the road network is asymmetric, the transposed matrix of Fig. 5 is also necessary to obtain the distance between a border node and an inner node.

3.2 Shortest Path Finding Algorithm

In the R*k*NN query algorithm described in Section 4, an operation to find the road network distance between two points (*s* and *d*) is necessary. This search can be achieved by referring to the IBDT and BBDT. **Figure 6** shows the process flow of the shortest path finding (SPF) algorithm [2].



(a) Initial state

(b) Shift to neighboring cell (PHASE1)

(c) PHASE2

Fig. 6 Processing flow of SPF.

The SPF is controlled by a best-first search using a priority queue PQ, which manages the records constructed as:

$$< p, Cost, dfs, fSG, phase >$$

where p is the current node (e.g., s, d, or a border node), $Cost$ is the lower-bound road network distance between s and d (and is the key used for ordering PQ), dfs (distance-from-source) is the shortest road network distance between s and p, fSG denotes the subgraph to which p belongs, and $phase$ is one of two values that shows the progression of the algorithm: PHASE1 (searching stage) or PHASE2 (final stage).

First, subgraph SG_s, which contains the road segment under s, is determined. Next, $Cost$ is calculated by the equation $Cost = d_N(s, b_i) + d_E(b_i, d)$ for all border nodes $b_i \in BV_s$ of SG_s. Here, $d_E(x, y)$ denotes the Euclidean distance between x and y and $d_N(x, y)$ denotes the road network distance between x and y. In the above equation, $d_N(s, b_i)$ can be obtained by referring to the IBDT of SG_s. In this initial stage, the following records are composed and added to PQ with $PHASE1$ as their phase value:

$$< b_i, d_N(s, b_i) + d_E(b_i, d), 0, SG_s, PHASE1 > \quad \forall b_i \in BV_s$$

If s and d are very close and included in the same subgraph, the distance cannot be obtained from the IBDT. In this case, a pairwise A* (PWA*) algorithm can search for the shortest path efficiently, because the two terminal points are located close to one another.

When the $phase$ value of the obtained record (e) from PQ is $PHASE1$ (see Fig. 6 (b)), the subgraph is shifted to the neighboring subgraph, SG_n. For each subgraph SG_n, $Cost$ is calculated as:

$$Cost = e.dfs + d_N(p, b_i) + d_E(b_i, d) \quad (b_i \in BV_n),$$

where BV_n is the border node set of SG_n. The following record is composed and added to PQ:

$$< b_i, Cost, e.dfs + d_N(p, b_i), SG_n, PHASE1 > \quad \forall b_i \in BV_s$$

Continuing the process, when a record obtained from PQ reaches a border node of the subgraph containing d (SG_d) (see Fig. 6 (c)), the record shown below is constructed and added to PQ.

$$< e.p, e.dfs + d_N(e.p, d), e.dfs, e.fSG, PHASE2 >,$$

where $d_N(e.p, d)$ is obtained from the IBDT of SG_d and $PHASE2$ indicates that a route between s and d has been found.

The shortest path is searched for using a best-first search. First, a record that has the minimum $Cost$ value is dequeued from PQ. When the $phase$ value of the dequeued record is $PHASE2$, the shortest path distance between s and d has been determined. The fact that the record has been dequeued from PQ indicates that it has the minimum $Cost$ value among all records contained in PQ. Therefore, the (shortest) distance has been returned and the search process is terminated.

Algorithm 1 shows the pseudo-code of the procedure described above.

Algorithm 1 SPF

```
function SPF($s$,$d$)
    $SG_s \leftarrow determineSG(s)$
    $SG_d \leftarrow determineSG(d)$
    $CS \leftarrow \emptyset$
    for all $p \in BV_s$ do
        $PQ.enqueue(p, d_N(s, p) + d_E(p, d), 0, SG_s, PHASE1)$
    end for
    while $PQ$ is not empty do
        $e \leftarrow PQ.dequeue()$
        if $e.phase = PHASE2$ then
            break
        else if $e.fSG = SG_d$ then
            $c \leftarrow e.dfs + d_N(e.p, d)$
            $PQ.enqueue(e.p, c, c, SG_d, PHASE2)$
        else
            $Nsg \leftarrow findNeighborSubGraph(e.p)$
            for all $sg \in Nsg$ do
                for all $b \in BV_{sg}$ do
                    $c \leftarrow e.dfs + d_N(e.p, b) + d_E(b, d)$
                    $PQ.enqueue(e.p, c, e.dfs + d_N(e.p, b), sg, PHASE1)$
                end for
            end for
        end if
    end while
    return $e.dfs$
end function
```

3.3 Partitioning a Large Graph

The partitioning of the road network into subgraphs can be performed by the following method: (1) we select the source nodes on the given road network that represent the desired number of subgraphs and (2) by applying Dijkstra's multi-source shortest path algorithm, we classify each road network node that has the same source node as its NN to group into a subgraph. Then, the BBDT and IBDT tables are prepared for each subgraph.

4. R*k*NN query on SMPV

4.1 Basic Method for R*k*NN Search

In this section, we describe a basic method for RkNN search in road networks, followed by an improved method based on the incremental Euclidean restriction (IER) method.

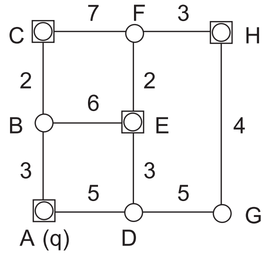


Fig. 7 Example of a road network.

Yiu et al. [1] presented the following lemma for an RkNN search in a road network.

Lemma 1 Let q be a query point, n be a road network node, and p be a data point that satisfies $d_N(q, n) > d_N(p, n)$. For any data point $p'(\neq p)$ whose shortest path to q passes through n, $d_N(q, p') > d_N(p, p')$. This means that p' is not an RNN of q.

This Lemma is proved in Ref. [1], where $d_N(a, b)$ denotes the road network distance between a and b.

Figure 7 shows a simple road network. Here, the circles indicate the nodes in the road network and the squares indicate data points. In this example, data points are assumed to be located on nodes; however, this restriction can be easily relaxed [7]. The numbers attached to edges show the cost (e.g., distance) of the edge. When we observe D, the NN data point of D is E and the NN data point of E is H; hence, A is not the NN data point of E. If we substitute n with D, p with E, and p' with H in Lemma 1, we obtain the relations $d_N(A, D) > d_N(E, D)$ and $d_N(A, H) > d_N(E, H)$. Therefore, even if we continue the search beyond D, we cannot find the RNN of q.

Yiu et al. [1] proposed the Eager algorithm based on Lemma 1 and a branch-and-bound approach. The Eager algorithm visits road network nodes from q to surrounding nodes using a method similar to that of Dijkstra's algorithm. When query q is on A in Fig. 7, node B is visited first. Next, at most k NNs of B are searched for within the distance $Dst = d_N(B, A)$. This function is called rangeNN(n, q, Dst). In the above example, n is B and q is A. For simplicity, we assume k is one.

In the previous query, C is found as B's NN. Next, we check whether C is included as an RNN of A. This check can be done to investigate whether A is the NN of C. This function is called verify(p, k, q) and returns true when q is the NN of p, otherwise, it returns false. In this example, the result of verify$(C, 1, q)$ is true; therefore, C is determined as an RNN of q. The next visited node is D; thus, rangeNN$(D, q, 5)$ is called and E is obtained as the NN of D. To check whether E is a RNN of q, verify$(E, 1, q)$ is called; however, false is obtained in this case. Hence, the edges beyond D are safely pruned. At this time, there is no search path left, therefore, the search process is terminated.

In Yiu's Eager algorithm, Dijkstra's algorithm is used for verify(p, k, q) and rangeNN(n, q, Dst). For simplicity, these functions are hereafter denoted as VERIFY and RANGENN. When the density of data points is high and the search area is small, this algorithm completes quickly. In contrast, when the density is low or k is large, the processing time becomes very long because the search area is large.

Conditions for the inefficiency of the Eager algorithm are summarized as follows:

- a large search area for the verify and rangeNN functions
- a drastic increase in processing time caused by performing rangeNN on every visited node

To cope with these problems, we propose a method to adapt an IER framework for the verify and rangeNN functions. Furthermore, in the next section, we propose an efficient method of R*k*NN search to perform these queries on the SMPV: (1) to adapt an IER framework for both rangeNN and verify and (2) to use the Eager algorithm only on the border nodes in the SMPV.

4.2 *k*NN query Using an IER Framework

Papadias et al. [13] proposed the IER framework to be adaptable for several types of queries on the road network. The basis of this framework is that the Euclidean distance between two terminal points is always a lower bound of the road network distance. For example, when query point q and distance r are specified and the task is to find all data points whose distances from q are less than or equal to r, this query (called a range query) can be performed by the following two steps:

(a) finding all data points residing in a circle whose center is q and radius is r

(b) verifying the road network distance of each data point obtained by the above step to eliminate the data points that have road network distances larger than r

Queries on Euclidean distance can be performed quickly via a spatial index (e.g., an R-tree). In addition, the distance verification can be performed quickly using the algorithm described in Section 3.2.

For the Eager algorithm, IER can be adapted to both rangeNN and verify. First, rangeNN$(n, k, d_N(n, q))$ is a range query centered at n with range distance $d_N(n, q)$. This query can be performed by the method mentioned above. Since the verify(p, k, q) function is essentially a kNN query, this query can also be efficiently performed via IER [13]. The advantage of using the IER framework increases when k is large and the distribution of data points is sparse.

4.3 R*k*NN Query on an SMPV Structure

The most time-consuming step in the Eager algorithm is the call to rangeNN at every expanded node. In Algorithm 3 presented in this subsection, rangeNN is invoked only on the border nodes of the subgraphs to alleviate this problem in the Eager algorithm.

When a query point q is given, the subgraph in the SMPV structure that q belongs to is determined and the data points belonging to the subgraph are searched. Let this data point set be P. Next, each element in P is checked to determine whether q is an RkNN or not. This procedure is the same as verify(p, k, q) in the Eager algorithm, as verify(p, k, q) searches for the kNNs of each $p \in P$. If q is included in the kNN set, p is determined to be an RkNN of q. This check requires a wide area search and is not exclusive to only a subgraph; IER can efficiently perform it using SMPV.

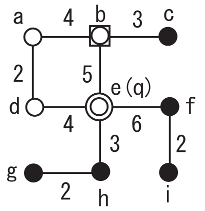


Fig. 8 Processing of a subgraph where q is included.

Figure 8 shows the same subgraph as in Fig. 3. In this example, a query point q is on node e. A square overlapped on node b indicates a data point. For simplicity, the following explanation considers the case for which k is one. By searching for the NN of b, q is obtained as the result. Therefore, b is an RNN of q. Consequently, b is added to the result set.

Next, we enlarge the search area to include the neighboring subgraph. For each border node b_i of this subgraph, the distance from q to b_i is obtained by referring to the IBDT of the subgraph. Thereafter, a record is composed and inserted into priority queue PQ. The record is composed as

$$< d, n, p, cid >$$

where d is the road network distance between q and the border node concerned (n), p is the previous node on the shortest path from q to n, and cid denotes the subgraph ID to which n belongs. The first record inserted into PQ is as follows.

$$< d_N(q, b_i), b_i, q, SG_a >$$

Here, SG_a denotes the subgraph in which q is included. For example, for border node c of Fig. 8, the record $< 8, c, q, SG_a >$ is inserted into PQ. The steps described above comprise the StartSG procedure, detailed in Algorithm 2.

Algorithm 2 StartSG

```
procedure StartSG($q, PQ, R$)
    $sg \leftarrow determineSG(q)$
    $P \leftarrow findPOIinSG(q)$
    for all $p \in P$ do
        if verify$(p, k, q)$ then $R \leftarrow$ R $\cup p$        ▷ add $p$ to result set
        end if
    end for
    for all $b \in BV$ do
        PQ.enQueue($< d_N(q, b), b, q, sg >$)
    end for
end procedure
```

Next, the RkNN search starts. When a record is dequeued from PQ, the search propagates to the neighboring subgraphs. In **Fig. 9**, SGa is the subgraph in which query point q is included and SGb is a neighboring subgraph. When record v is dequeued from PQ and $v.n$ is the border node b, data points in SGb are searched. In this subgraph, data point d is included. Next, the kNNs of d are searched for, and if q is included in the kNN set, d is added to the result set. Otherwise, d is ignored. This subgraph can be visited several times from different border nodes. Thereafter, SGb is marked as visited to avoid duplicate searches.

Next, rangeNN is invoked from the border node b_i to find candidate data points. If the result set is not empty, verify is invoked to check whether each data point is truly an RkNN of q. If the result of VERIFY is true, the data point is added to the result set. If the size of RANGENN is smaller than k, other RkNNs could exist on the path through this node. Therefore, new records from b_i to the other border nodes in the subgraph are created and inserted into PQ.

Algorithm 3 RkNN

```
1:  function RkNN($q$)
2:      $PQ \leftarrow \emptyset, R \leftarrow \emptyset$
3:      StartSG($q, PQ, R$)
4:      while PQ not empty do
5:          $v \leftarrow PQ.deQueue()$
6:          $CS.add(v)$
7:          $KNN \leftarrow$ rangeNN($v.n, k, d_N(v.n, q), PQ$)
8:          for all $p$ in KNN do
9:              if verify($p, k, q$) then
10:                 $R \leftarrow R \cup p$
11:             end if
12:         end for
13:         if $|KNN| < k$ then
14:             for all $b \in BV$ do
15:                 if v.cid is visited first time then
16:                     $CP \leftarrow findPOIinSG(v.cid)$
17:                     for all $p \in CP$ do
18:                         if verify($p, k, q$) then
19:                             $R \leftarrow R \cup p$
20:                         end if
21:                     end for
22:                 end if
23:                 PQ.enQueue($< d_N(q, b), b, p, v.cid >$)
24:             end for
25:         end if
26:     end while
27:     return $R$                                  ▷ RkNN of $q$
28: end function
```

Algorithm 3 shows the pseudo-code of the proposed method described above. Lines 4–12 are similar to the process described by the Eager algorithm. When record v is obtained from PQ, at most k NNs of the network node $v.n$ are searched and added to KNN. For each element p of KNN, p is checked to determine whether q is included in its kNN. If it is included, p is inserted into the result set R.

Line 13 of Algorithm 3 checks whether the number of elements in KNN is less than k, i.e., the number of data points that exist in the area whose distance from $v.n$ is less than k. If the result is true, node $v.n$ is expanded and the search is continued. Otherwise, no more RkNNs exist on the path through $v.n$, and therefore, node expansion at $v.n$ is not executed.

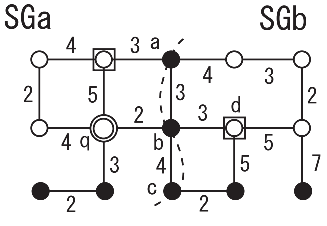


Fig. 9 Border node expansion.

5. Experimental Results

We evaluated our proposed method by comparing it with the Eager algorithm presented in Ref. [1]. Both algorithms were implemented in Java and evaluated on a PC with an Intel Core i7-4770 CPU (3.4 GHz) and 32 GB of memory. **Table 1** shows the road network maps used in this experiment. In this table, “adj. list” refers to the size of the adjacency list, and BBDT and IBDT are the size of the tables described in Section 3.

The adjacency list was prepared as follows: (1) the Peano-Hilbert order [14] was assigned to all nodes and (2) neighboring nodes in this order were clustered into 8 KB blocks. For adjacency list management, 0.5 MB (64-block) LRU buffer was assigned. For SMPV, road network graphs were partitioned by the method described in Section 3.3. The average number of nodes in a subgraph was approximately 240 for these two types of maps.

Table 2 compares the data size of the adjacency list (used primarily by the Eager algorithm), SMPV (total of the adjacency list, BBDT, and IBDT), and HEPV [11]. The data size of SMPV is approximately 4–5 times larger than that of the Eager algorithm which uses only an adjacency list. However, SMPV drastically reduces the data size in comparison to HEPV.

We generated several data point sets on the road network links using pseudorandom sequences, while changing the density D. For example, $D = 0.01$ indicates that a data point exists once every 100 links.

To evaluate the processing time of the SPF of SMPV, we compared it with the PWA* algorithm. In this experiment, 10 nearest neighbors (10NN) were searched based on the IER strategy, i.e. candidates were searched using Euclidean distance, and then the distance between the query point and each candidate was searched using the A* algorithm and SMPV, respectively. **Figure 10** shows the result. As shown in this figure, SMPV searched kNN about 100 times faster than PWA*.

Figure 11 compares the processing time of the kNN query on SMPV by varying the average number of nodes in the subgraphs. In SMPV, the processing time is minimal and stable when the average number of nodes is 240.

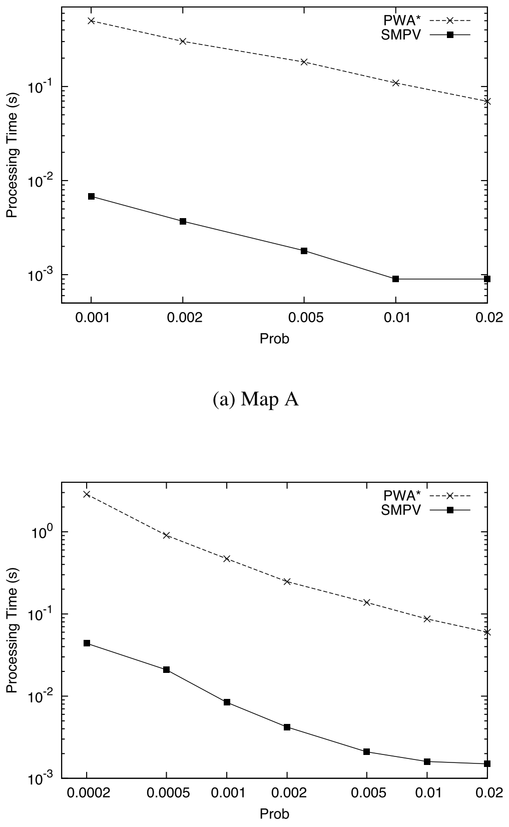
Figure 12 shows the processing times of RkNN queries when the density of data points is 0.01. Figure 12 (a) and (b) show the results for Map A and Map B, respectively. In the figures, the horizontal axes show k and the vertical axes show the processing times in seconds. As shown in Fig. 12 (a), the processing time of the Eager algorithm sharply increases with k because the search area also expands. In contrast, the proposed algorithm linearly increases with k. For Fig. 12 (b), the tendency is almost the same, however, the difference between the Eager algorithm and our proposed method is larger for Map B (i.e., for Fig. 12 (b)).

Table 1 Road network maps used in the experiments.

map name	# of nodes	# of edges	area size	adj. list	BBDT	IBDT
Map A	16,284	24,914	168 km^2	1.5 MB	1.1 MB	4.1 MB
Map B	109,373	81,233	284 km^2	6.8 MB	4.5 MB	17.4 MB

Table 2 Data size (MB).

map name	Eager & Lazy	SMPV	HEPV
Map A	1.5	6.7	30.1
Map B	6.8	28.7	376.1



(a) Map A

(b) Map B

Fig. 10 Processing time for *k*NN queries.

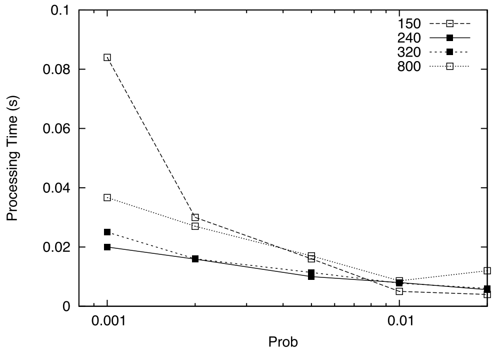
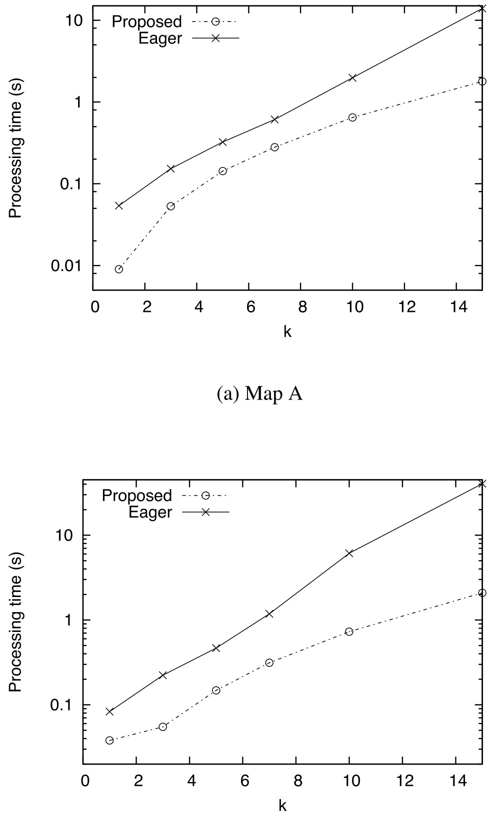


Fig. 11 Varying subgraph size.

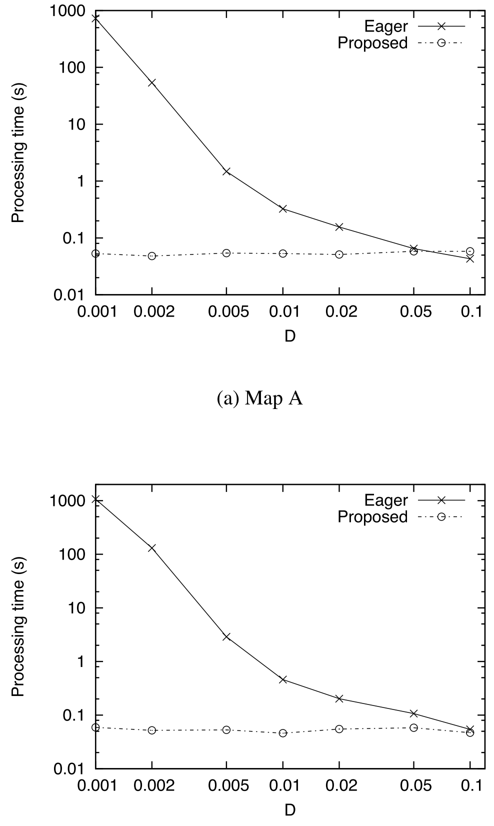
Figure 13 shows the processing times when the density of data points (*D*) varies. Here *k* was fixed to five. The processing time of the Eager algorithm increases sharply when the density is low. On the other hand, the proposed algorithm remains low even in that case. When the density of data point is high, the Eager algorithm performed well because the size of the search area decreases with increasing density. The proposed algorithm showed stable characteristics and was independent of the probability.



(a) Map A

(b) Map B

Fig. 12 Processing time for varying *k* value.



(a) Map A

(b) Map B

Fig. 13 Processing time for varying Prob value.

6. Conclusion

In this paper, we proposed an RkNN query algorithm using a simple distance materialization approach suitable for LBS. The amount of data used in these methods decreases when compared with conventional hierarchical network distance materialized methods.

The basis of the proposed method is to expand the search area in concentric circles, similar to the Eager algorithm. At every border node, the algorithm searches for data points in the range centered at the query point with a radius of the distance between the query point and the border node. If data points are found by the query point, whether the results are truly included in the RkNN of the query point is determined. In the proposed method, the RANGENN procedure is performed only on border nodes. This limitation drastically reduces the overall total time needed for invoking RANGENN. In addition, the IER adaptation of the RANGENN and VERIFY procedures helps reduces the overall processing time. Consequently, the proposed method performs RkNN on a road network quickly and efficiently, especially when the distribution of the data points is sparse or k is large. The complexity analysis of the proposed algorithm is for future work.

Acknowledgments The present study was partially supported by the Japanese Ministry of Education, Science, Sports, and Culture (Grant-in-Aid Scientific Research (C) 24500107).

References

[1] Yiu, M.L., Papadias, D., Mamoulis, N. and Tao, Y.: Reverse Nearest Neighbor in Large Graphs, *IEEE Trans. Knowledge and Data Engineering*, Vol.18, No.4, pp.1–14 (2006).

[2] Hlaing, A.T., Htoo, H., Ohsawa, Y., Sonehara, N. and Sakauchi, M.: Shortest Path Finder with Light Materialized Path View for Location Based Services, *Proc. WAIM 2013*, Vol.LNCS7923, pp.229–234 (2013).

[3] Korn, F. and Muthukrishnan, S.: Influence Sets Based on Reverse Nearest Neighbor Queries, *ACM SIGMOD Record*, Vol.29, pp.201–212 (2000).

[4] Stanoi, I., Agawal, D. and Abbadi, A.E.: Reverse Nearest Neighbor Queries for Dynamic Databases, *Proc. 2000 ACM SIGMOD Workshop on Reserch Issues in Data Mining and Knowledge Discovery*, pp.44–53 (2000).

[5] Tao, Y., Papadias, D. and Lian, X.: Reverse kNN Search in Arbitrary Dimensionality, *Proc. 30th VLDB Conference*, pp.744–755 (2004).

[6] Safar, M., Ibrahimi, D. and Taniar, D.: Vorinoi-based reverse nearest neighbor query processing on spatial netowrks, *Multimedia Systems*, Vol.15, pp.295–308 (2009).

[7] Cheema, M.A., Zhang, W., Lin, X., Zhang, Y. and Li, X.: Continuous reverse k nearest neighbors queries in Euclidean space and in spatial networks, *VLDB Journal*, Vol.21, pp.69–95 (2012).

[8] Jing, N., Huang, Y.W. and Rundensteiner, E.A.: Hierarchical Optimization of Optimal Path Finding for Transportation Applications, *Proc. Fifth Int'l Conf. Information and Knowledge Management*, pp.268–276 (1996).

[9] Samet, H., Sankaranarayanan, J. and Alborzi, H.: Scalable Network Distance Browsing in Spatial Databases, *Proc. ACM SIGMOD Conference*, pp.43–54 (2008).

[10] Jung, S. and Pramanik, S.: HiTi Graph Model of Topographical Road Maps in Navigation Systems, *Proc. 12th International Conference on Data Engineering*, pp.76–84 (1996).

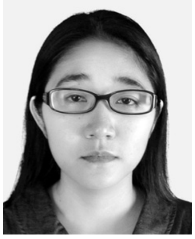
[11] Jung, S. and Pramanik, S.: An Efficient Path Computation Model for Hierarchically Structured Topographical Road Maps, *IEEE Trans. Knowledge and Data Engineering*, Vol.14, No.5, pp.1029–1046 (2002).

[12] Shekhar, S., Fetterer, A. and Goyal, B.: Meterialization Trade-Offs in Hierarchical Shortest Path Algorithms, *5th International Symposium on Large Spatial Databases*, pp.94–111 (1997).

[13] Papadias, D., Zhang, J., Mamoulis, N. and Tao, Y.: Query Processing in Spatial Network Databases, *Proc. 29th VLDB*, pp.790–801 (2003).

[14] Liu, X. and Schrack, G.: Encoding and Decoding the Hilbert Order, *Software – Practice and Experience*, Vol.26, No.12, pp.1335–1346 (1996).

Aye Thida Hlaing received her B.C.Tech and M.C.Sc. degrees from the University of Computer Studies, Yangon, Myanmar (UCSY) and Nanyang Technological University, Singapore in 2001 and 2008, respectively. She worked as teaching staff at UCSY from 2001 to 2002. She is currently a doctoral student at the Graduate School of Science and Engineering, Saitama University, Japan. Geographic information systems and spatio-temporal databases are her current research topics.



Tin Nilar Win received her Bachelor of Engineering (B.E. IT) from the Mandalay Technological University (MTU), Myanmar, in 2006. She is currently a student at the Graduate School of Science and Engineering, Saitama University, Japan. Her research interests include geographic information systems and location based services.

Htoo Htoo received her B.C.Sc. and M.C.Sc. degrees from the University of Computer Studies, Yangon, Myanmar in 2002 and 2004, respectively. She worked as a teaching staff at the University of Computer Studies, Yangon, Myanmar from 2004 to 2008 and received her Ph.D. degree from Saitama University in 2013. Currently, she is an assistant professor at Saitama University with a specialization in location based services (LBS) and spatio-temporal databases.

Yutaka Ohsawa received his B.E. and M.E. degrees from Shinshu University in 1976 and 1978, respectively, and his Ph.D. degree from the University of Tokyo in 1985. From 1979 to 1989, he worked for the Institute of Industrial Science at the University of Tokyo as a research associate and an assistant professor. Since 1989, he has worked for Saitama University. He is currently a professor at the Graduate School of Science and Engineering at Saitama University. His research interests include geographic information systems, spatio-temporal databases, and location based services.