Monitoring Aggregate k-NN Objects in Road Networks

Lu Qin[1], Jeffrey Xu Yu[1], Bolin Ding[1] and Yoshiharu Ishikawa[2]

[1] The Chinese University of Hong Kong, China, {lqin,yu,blding}@se.cuhk.edu.hk
[2] Nagoya University, Japan, ishikawa@itc.nagoya-u.ac.jp

Abstract. In recent years, there is an increasing need to monitor k nearest neighbor (k-NN) in a road network. There are existing solutions on either monitoring k-NN objects from a single query point over a road network, or computing the snapshot k-NN objects over a road network to minimize an aggregate distance function with respect to multiple query points. In this paper, we study a new problem that is to monitor k-NN objects over a road network from multiple query points to minimize an aggregate distance function with respect to the multiple query points. We call it a continuous aggregate k-NN (CANN) query. We propose a new approach that can significantly reduce the cost of computing network distances when monitoring aggregate k-NN objects on road networks. We conducted extensive experimental studies and confirmed the efficiency of our algorithms.

1 Introduction

With the development of positioning technologies such as the Global Positioning System (GPS), many applications are developed in transportation domains by taking advantages of monitoring object movements in road networks where the position and distance of objects are constrained by spatial networks. An important type of these queries is a k nearest neighbor (k-NN) query, which is widely used in location-based services, traffic monitoring, emergency management. Existing solutions focused on either monitoring k-NN objects over a road network from a single query point (observation point) [1], or computing the snapshot k-NN objects over a road network to minimize an aggregate distance function with respect to the multiple query points [2]. In this paper, we study a new problem that is to monitor k-NN objects over a road network from multiple query points to minimize an aggregate distance function with respect to multiple query points. We call it a continuous aggregate k-NN (CANN) query. In brief, it deals with the network distance instead of Euclidean distance, and it monitors the top-k objects, where an object is ranked based on an aggregate function value of the distances between the object and multiple query points. As an example, consider people in n companies/organizations need to schedule meetings in downtown frequently. The room availabilities in hotels and restaurants is monitored, and the best place is selected to reduce the total travel time for people to meet.

The main difficulties for processing CANN query are as follows. First, when there are a large number of objects in the road network or there are a large number of CANN queries, the cost of computing network distances becomes the bottleneck. Second, an object is ranked in the road network based on an aggregate function value in terms

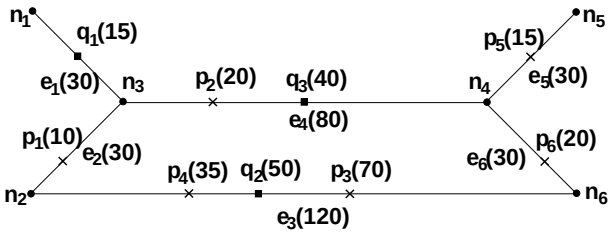


Fig. 1. Road Network

of the network distances to a set of query points. Unlike computing a CANN query for a single query point in the road network where the order of visiting edges can be determined using an expansion tree from the query point, computing the CANN query from multiple query points makes it difficult to find an order of visiting edges.

The main contributions of this paper are summarized below. (1) We study a new problem of processing the continuous aggregate nearest neighbor queries (CANN) over large road network. To the best of our knowledge, this is the first attempt to study this problem. (2) We propose new approaches that do not need to expand tree to compute CANN queries. Our approach can reduce the cost of computing network distances significantly. (3) We conducted extensive performance studies, and confirmed the efficiency of our new approaches.

The rest of the paper is organized as follows. Section 2 gives the problem statement. Section 3 introduces two existing solutions. Section 4.2 discusses our new approaches followed by discussions on implementations in Section 5. Section 6 shows our experimental results. The related work is given in Section 7. Finally, Section 8 concludes this paper.

2 Problem Definition

Road Network is an undirected weighted connected graph, $G(V, E)$, where V is a set of nodes (road intersections), and E is a set of edges (roads). An edge, $e \in E$, connects two nodes n_i and n_j. A positive number, $len(e)$, denotes the length of the edge e. *(Data or Query) points* lie on edges of road network G. We use $pos_e(p)$ to denote the position of a point p on $e = (n_i, n_j)$ by the distance from point p to node n_i on edge e, provided $i < j$.

Network Distance: For two nodes $n_i, n_j \in V$, the network distance $d(n_i, n_j)$ is the length of the shortest path between n_i and n_j in the road network. The network distance between a point, p that lies on the edge $e = (n_i, n_j)$, and a node, n_k, is computed as $d(p, n_k) = min\{pos_e(p) + d(n_i, n_k), (len(e) - pos_e(p)) + d(n_j, n_k)\}$. For any two data points p and p', if p and p' are on different edges, their network distance is computed as $d(p, p') = min\{pos_e(p) + d(p', n_i), (len(e) - pos_e(p)) + d(p', n_j)\}$. Otherwise, $d(p, p')$ is $min\ \{|pos_e(p) - pos_e(p')|, pos_e(p) + d(p', n_i), (len(e) - pos_e(p)) + d(p', n_j)\}$.

Figure 1 shows a simple road network. There are 6 nodes and 6 edges. The number in the brackets under each edge e_i denotes its length ($len(e_i)$). For instance, e_4 is the

edge that connects nodes n_3 and n_4, and the length of e_4 is $len(e_4) = 80$. In Figure 1, a data point is indicated by a cross. The position of a data point is marked in the brackets above it. For instance, p_3 lies on edge e_3, and its position is $pos_{e_3}(p_3) = 70$. The network distance between two nodes, n_1 and n_6, is $d(n_1, n_6) = 30+80+30 = 140$, along the shortest path $e_1 \rightarrow e_4 \rightarrow e_6$, the network distance between data point p_3 and node n_4 is $d(p_3, n_4) = min\{pos_{e_3}(p_3) + d(n_2, n_4), (len(e_3) - pos_{e_3}(p_3)) + d(n_6, n_4)\} = min\{70 + 110, 50 + 30\} = 80$, and the network distance between two data points, p_3 and p_2, that are on two different edges e_3 and e_4, is $d(p_3, p_2) = min\{pos_{e_3}(p_3) + d(n_2, p_2), (len(e_3) - pos_{e_3}(p_3)) + d(n_6, p_2)\} = min\{70 + 50, 50 + 90\} = 120$.

Problem Statement (CANN Query): Given a road network $G(V, E)$ and the set of data points (moving objects) $P = \{p_1, p_2, \cdots\}$ over $G(V, E)$. A continuous aggregate nearest neighbor query is denoted as $\mathsf{CANN}(Q, k, h)$, where $Q = \{q_1, q_2, \cdots\}$ is a set of fixed query points over $G(V, E)$, k is a positive number (> 0), and h is an aggregate function (sum, min, max). Here, for a data point, $p_i \in P$, $h(p_i) = h\{d(p_i, q_1), d(p_i, q_2), \cdots, d(p_i, q_{|Q|})\}$, regarding the query points Q. The CANN (Q, k, h) query is to monitor the top-k data points in P that has the smallest h function values while all data points are moving.

Consider a $\mathsf{CANN}(Q, k, sum)$ where $Q = \{q_1, q_2, q_3\}$, $k = 3$ against $G(V, E)$ (Figure 1). Here, $sum(p_1) = sum\{d(p_1, q_1), d(p_1, q_2), d(p_1, q_3)\} = 35 + 60 + 60 = 155$, $sum(p_2) = 155$, $sum(p_3) = 255$, $sum(p_4) = 200$, $sum(p_5) = 280$, and $sum(p_6) = 255$. The top-3 result is $\{p_1, p_2, p_4\}$.

3 Existing Solutions

While many recent researches have focused on continuous monitoring of nearest neighbors over dynamic objects, we first propose the solution for CANN query in road networks. Mouratidis et al.'s work in [1] is the one closest to ours. They gave two algorithms, IMA and GMA, to process continuous nearest neighbor queries over a road network, when there is a single query point, i.e., $\mathsf{CANN}(Q, k, h)$ where $|Q| = 1$ (a special case of CANN).

The incremental monitoring algorithm (IMA) retrieves the initial top-k data points using the shortest path expansion tree of the query point for a single CANN query. The group monitoring algorithm (GMA) groups multiple CANN queries that lie on the same edge, as a group, to process them together, based on IMA. IMA keeps expanding the tree and updating the top-k result until the next edge to be expanded has minimal distance that is no less than the kth distance in the current result. When data points move, the result for the query is maintained by incrementally expanding or shrinking the expansion tree.

Figure 2 shows an example to explain the expansion tree for $\mathsf{CANN}(\{q_3\}, k, sum)$, where $k = 3$. Assume the current top-3 result is $\{p_1, p_2, p_5\}$, and the edges (called partial edges) that may partially affect the new top-3 results when data points move are $P = \{e_1, e_2, e_5, e_6\}$. Suppose the data point p_1 moves out of the partial edges. IMA needs to expand the expansion tree from nodes n_3 and n_4 and retrieve all the data points on the edges in P to obtain the new top-k result $\{p_2, p_4, p_5\}$. In summary, when data

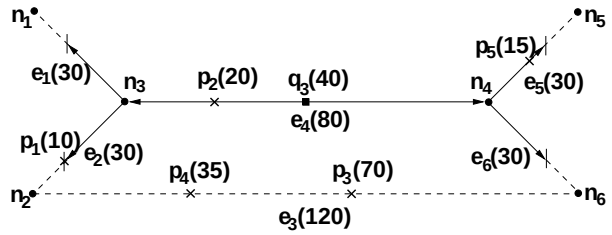
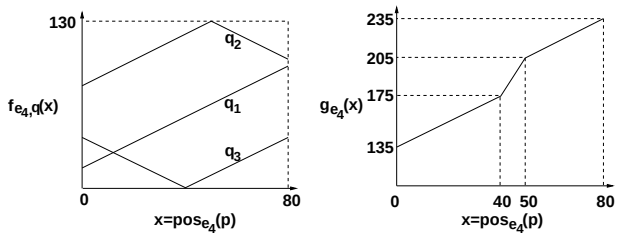
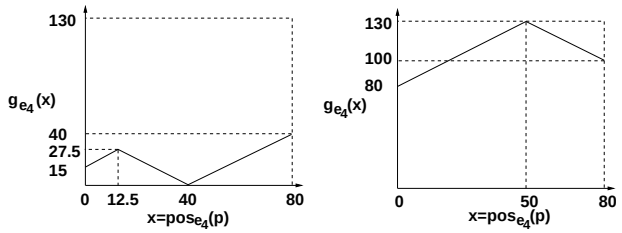


Fig. 2. Expansion tree and partial Edges



(a) Single source edge functions (b) Sum



(c) Min (d) Max

Fig. 3. Edge functions on e_4

points move, IMA does not need to recompute the result from scratch, for a CANN query, but it needs to retrieve the data points, that lie on the partial edges, which is time consuming. In this paper, we propose a new approach that does not need to use an expansion tree for CANN, and allows multiple query points in one CANN query.

4 A New Non-Tree-Expanding Approach

The high online processing cost for CANN queries dues to the frequent update of the expansion tree. In this paper, we propose a new approach that does not need an expansion tree. In brief, for a new CANN(Q, k, h) query registered, we construct a query graph, $G_Q(V_Q, E_Q)$, based on CANN and the road network G. The query graph, $G_Q(V_Q, E_Q)$, is static when processing CANN(Q, k, h) (no update is needed). It facilitates computing the value of aggregate function $h(p)$, for a data point $p \in P$. With the assistance of query graph G_Q, we can efficiently monitor the top-k results, when the data points move.

4.1 Query Graph Construction

The query graph $G_Q(V_Q, E_Q)$ facilitates computing the value of aggregate function $h(p)$ in $\mathsf{CANN}(Q, k, h)$, for a data point, $p \in P$. We require that, given the position of p on edge e, $pos_e(p)$, the value of aggregate function $h(p)$ can be computed efficiently. We first discuss the relationship between the distance function $d(q, p)$ / the aggregate function $h(p)$ and the position of p.

Distance function w.r.t. $pos_e(p)$: Consider a data point p on an edge $e = (n_i, n_j)$ in G_Q, and a query point $q \in Q$. The distance $d(p, q)$ between q and p can be specified as a function of $pos_e(p)$, denoted as $f_{e,q}$: $f_{e,q}(pos_e(p)) = d(p, q)$. We note that function $f_{e,q}(\cdot)$ is a continuous piecewise-linear function in the domain $[0, len(e)]$. We discuss the main idea behind $f_{e,q}(pos_e(p))$ followed by the discussion on how to compute it.

An example is illustrated in Figure 3(a) over the road network G (Figure 1). Take the edge $e_4 = (n_3, n_4)$ in G as an example. Its three functions, $f_{e_4,q_1}(pos_{e_4}(p))$, $f_{e_4,q_2}(pos_{e_4}(p))$, and $f_{e_4,q_3}(pos_{e_4}(p))$, for three different query points, q_1, q_2, and q_3, are shown in Figure 3(a). Note: on x-axis, $[0, len(e_4)]$, is $pos_{e_4}(p)$, the distance from n_3. The curve of $f_{e_4,q_1}(pos_{e_4}(p))$ suggests that the shortest distance between q_1 and any data points p on e_4 should first go to the end node n_3 of e_4, and then go to p. The curve of $f_{e_4,q_2}(pos_{e_4}(p))$ suggests that the shortest distance between q_2 and any data points p on e_4 may come from two different ends of e_4 (from either n_3 or n_4). When the data point p is on the left side of $[0, len(e_4)]$ before the peak value of $f_{e_4,q_2}(pos_{e_4}(p))$, the shortest distance between q_2 and p should come from the end of n_3; when the data point p is on the right side of $[0, len(e_4)]$ after the peak value of $f_{e_4,q_2}(pos_{e_4}(p))$, the shortest distance between q_2 and p should come from the end of n_4.

Function $f_{e,q}(pos_e(p))$ can be computed as follows. Assume $e = (n_i, n_j)$, where $i < j$. With Dijkstra's single-source shortest-path algorithm, we obtain the shortest distance from q to every node in G. There are two cases.

i) q is not on edge e: If $|d(q, n_i) - d(q, n_j)| = len(e)$, $f_{e,q}(pos_e(p))$ is a 1-piece linear function of $pos_e(p) \in [0, len(e)]$. In this case, its 1-piece segment is $(0, f_{e,q}(0))$-$(len(e), f_{e,q}(len(e)))$, where $f_{e,q}(0) = d(q, n_i)$ and $f_{e,q}(len(e)) = d(q, n_j)$. Otherwise, $f_{e,q}(pos_e(p))$ is a 2-piece linear function, and its two linear segments are $(0, d(q, n_i))$-(x, y), and (x, y)-$(len(e), d(q, n_j))$, where x and y are computed as follows.

$$\begin{cases} x = \frac{d(q,n_j) - d(q,n_i) + len(e)}{2} \\ y = \frac{d(q,n_j) + d(q,n_i) + len(e)}{2} \end{cases} \quad (1)$$

ii) q is on edge e: Query point q split e into two parts, from n_i to q and from q to n_j respectively. Consider q as a node, function $f_{e,q}(pos_e(p))$ on each part shares high similarity to case i), thus, we omit further explanation. The curve of $f_{e_4,q_3}(pos_{e_4}(p))$ shows such an example. But notice that function $f_{e,q}(pos_e(p))$ of $pos_e(p) \in [0, len(e)]$ may be a 3-piece linear function here. The 3-piece case happens only if q is on e.

From above discussions, we have the following lemma.

Lemma 1. *$f_{e,q}(\cdot)$ is a continuous piecewise-linear function with at most 3 linear pieces on domain $[0, len(e)]$.*

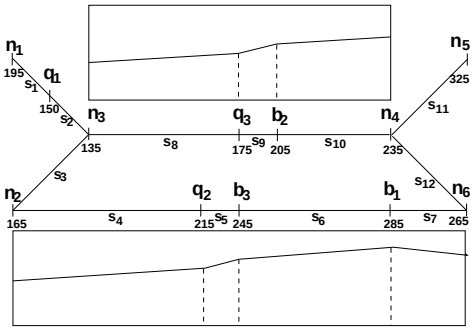


Fig. 4. Query graph

Aggregate function w.r.t. $pos_e(p)$: Since distance function $f_{e,q}(pos_e(p))$ is a *continuous piecewise-linear function* of $pos_e(p)$, given $\mathsf{CANN}(Q, k, h)$ and $Q = \{q_1, q_2, \cdots\}$, the aggregate function value for any data point p on edge e, regarding all query points, can also be specified as a *continuous piecewise-linear function* of $pos_e(p)$, denoted by

$$g_e(pos_e(p)) = h\{f_{e,q_1}(pos_e(p)), ..., f_{e,q_{|Q|}}(pos_e(p))\} \tag{2}$$

for $pos_e(p) \in [0, len(e)]$. Since $f_{e,q}(\cdot)$ has at most 3 linear pieces, $g_e(\cdot)$ has at most $O(|Q|)$ linear pieces.

Lemma 2. *$g_e(\cdot)$ is a continuous piecewise-linear function with at most $O(|Q|)$ linear pieces on domain $[0, len(e)]$.*

Reconsider the example in Figure 3(a) for the three query points, q_1, q_2, and q_3. The aggregate function on edge e_4 for $h = sum$, min, and max, are shown in Figure 3(b), Figure 3(c), and Figure 3(d), respectively.

Constructing the query graph $G_Q(V_Q, E_Q)$: Given a $\mathsf{CANN}(Q, k, h)$ query over a road network $G(V, E)$, we define a *query graph*, $G_Q(V_Q, E_Q)$, to efficiently compute the value of $h(p)$ given $pos_e(p)$, the position of a data point p on edge e. The idea to construct G_Q is to segment edges in G, such that aggregate function $g_e(\cdot)$ w.r.t. $pos_e(p)$ is a 1-piece linear function within each segment.

Formally, suppose on an edge, $e = (n_i, n_j)$ in E, $g_e(\cdot)$ is a z-piece linear function, then e needs to be segmented into a sequence of edges, (n_{k_0}, n_{k_1}), (n_{k_1}, n_{k_2}), $\cdots$, $(n_{k_{z-1}}, n_{k_z})$, where $n_i = n_{k_0}$ and $n_{k_z} = n_j$, such that $g_e(\cdot)$ is a 1-piece linear function on each segment $[pos_e(n_{k_{l-1}}), pos_e(n_{k_l})]$ $(1 \le l \le z)$. All such nodes n_{k_l}, for $0 \le l \le z$, will be included in V_Q, and all the segmented edges $(n_{k_{l-1}}, n_{k_l})$, for $1 \le l \le z$, will be included in E_Q. If $g_e(\cdot)$ is a 1-piece linear function, then there is no segmentation needed over an edge $e = (n_i, n_j)$ (n_i, n_j are included in V_Q, and e is included in E_Q).

We explain how to segment an edge using an example (Figure 3(b)), for a $\mathsf{CANN}(Q, k, h)$ where $Q = \{q_1, q_2, q_3\}$, and $h = sum$. Consider edge $e_4 = (n_3, n_4)$, as shown in Figure 3(b), its aggregate edge function is a continuous 3-piece-segment linear function. Therefore, we add two new nodes into query graph G_Q, denoted, n_{k_1} and n_{k_2} at position 40 and 50 on the x-axis as shown in Figure 3(b). Note: 40 and 50 are the distance from n_3. $e_4 = (n_3, n_4)$ will be segmented into three edges, (n_3, n_{k_1}), (n_{k_1}, n_{k_2}), and

Algorithm 1 $IRC(C_i)$

1: $C_i.top \leftarrow \emptyset$; $C_i.k \leftarrow +\infty$;
2: $e \leftarrow head(C_i.E)$;
3: **while** $e \neq \emptyset$ **and** $low(e) \leq C_i.k$ **do**
4: update $C_i.top$ and $C_i.k$ with data points on e;
5: $e \leftarrow next(C_i.E)$;

(n_{k_2}, n_4), in G_Q. Each of the three edges is associated with a 1-piece linear aggregate function.

It is important to note that in $G_Q(V_Q, E_Q)$, every edge is associated with a 1-piece linear function (a piece of $g_e(\cdot)$). We can compute the value of the aggregate function for any data point in any edge with the help of G_Q efficiently. Consider an edge $(n_{k_{l-1}}, n_{k_l})$ in G_Q, which is an edge segment of an edge e in G. Let $x_{l-1} = pos_e(n_{k_{l-1}})$ and $x_l = pos_e(n_{k_l})$. Let $y_{l-1} = g_e(x_{l-1})$ and $y_l = g_e(x_l)$ be the aggregate function values at nodes $n_{k_{l-1}}$ and n_{k_l}. When the position of a data point p, $pos_e(p)$, is within $[x_{l-1}, x_l]$, since $g_e(\cdot)$ is a 1-piece linear function on $[x_{l-1}, x_l]$, the aggregate function value at point p can be computed as:

$$g_e(pos_e(p)) = y_{l-1} + \frac{(y_l - y_{l-1}) \cdot (pos_e(p) - x_{l-1})}{(x_l - x_{l-1})}. \quad (3)$$

Figure 4 shows a query graph, $G_Q(V_Q, E_Q)$ over the road network G (Figure 1), for a $\mathsf{CANN}(\{q_1, q_2, q_3\}, k, sum)$ query. There are totally 12 edges in G_Q, and each of them is marked as s_i for $1 \leq i \leq 12$. In addition to the original 6 nodes in $G(V, E)$, n_j, for $1 \leq j \leq 6$, there are 6 nodes q_1 – q_3 (for the three query points), and b_1 – b_3, which segment edges in E into linear pieces. The number below each node denotes the g_e value. The relationship between the the aggregate edge functions and the two horizontal edges are illustrated in Figure 4.

Lemma 3. *The time complexity for the construction of query graph $G_Q(V_Q, E_Q)$ is $O((n \cdot \log n + m \cdot \log |Q|) \cdot |Q|)$, where $n = |V|$ and $m = |E|$, given graph $G(V, E)$.*

Proof. For each query point q in Q, the complexity to find the distances from source q to every other node in G is $O(n \cdot \log n + m)$. In sum, we need $O((n \cdot \log(n) + m) \cdot |Q|)$ time. Moreover, since $g_e(\cdot)$ has at most $O(|Q|)$ linear pieces (Lemma 2), $|V_Q|$ and $|E_Q|$ are both bounded by $O(|Q| \cdot m)$. To segment an edge $e \in E$ into a sequence of edges in E_Q, we need $O(|Q| \log |Q|)$ time (sort all the linear pieces and scan them). Therefore, the total time complexity is $O((n \cdot \log n + m \cdot \log |Q|) \cdot |Q|)$. □

4.2 Basic Top-k Monitoring algorithm

In this subsection, we introduce our basic algorithm to monitor the top-k result for a set of CANN queries, $\{C_1, C_2, \cdots\}$, where $C_i = \mathsf{CANN}(Q_i, k_i, h_i)$, over a road network G with data point set P.

For each query, C_i, the query graph is denoted as $G_{Q_i}(V_{Q_i}, E_{Q_i})$. Because of the property of query graphs we discussed in the previous subsection (recall Lemma 2), in

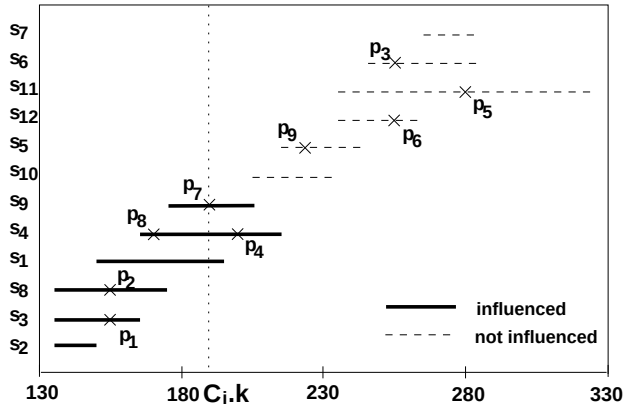


Fig. 5. Example for IRC and MTR

the following part, we can assume the aggregate function value at a given data point p w.r.t. query C_i can be computed in constant time (according to Equation (3)).

All edges in E_{Q_i} are sorted in the ascending order of the aggregate function lower bounds within the edges. The sorted edge list is denoted by $C_i.E$. A pointer is associated with the ordered list $C_i.E$, and four operations are defined: i) $head(C_i.E)$ – set the pointer to the first edge in $C_i.E$ and return this edge; ii) $current(C_i.E)$ – return the edge pointed by the pointer currently; iii) $next(C_i.E)$ – move the pointer to the next edge and return this edge (or return $emptyset$ if the pointer points to the end of $C_i.E$); iv) $prev(C_i.E)$ – move the pointer to the previous edge and return this edge.

Initial Top-k Result Computation: The algorithm IRC (Algorithm 1) computes the top-k_i data points for a query C_i. In line 1, $C_i.top$, used to keep the set of the top-k_i data points for C_i, is initialized as empty; $C_i.k$, used to record the k_i-th smallest aggregate value of the data points kept in $C_i.top$, is initialized as $+\infty$. In line 2, $head(C_i.E)$ returns the first edge in $C_i.E$. In the while statement (line 3-6), it computes the top-k_i data points for C_i by scanning the ordered list $C_i.E$. In line 3, $e \neq \emptyset$ means $C_i.E$ has not been scanned to the end yet, and $low(e)$ denotes the aggregate function's lower bound within the edge e.

The case $low(e) \leq C_i.k$, called *edge e is influenced*, indicates that there may be some data points on e, which can be included in $C_i.top$. In this case, the top-k list ($C_i.top$) and the k_i-th smallest aggregate value in $C_i.top$ are updated using all the data points on the edge e (line 4).

Figure 5 shows an example over the road network G (Figure 1), for the query $C_i = \mathsf{CANN}(\{q_1, q_2, q_3\}, 3, sum)$. The label for each segment, s_l, for $1 \leq l \leq 12$ is illustrated in Figure 4. The x-axis shows the aggregate function values and the y-axis shows the list of edges $C_i.E$. All edges are listed in ascending order of the aggregate function lower bound on them, and each data point is marked as a cross in edges. Suppose the current set of data points is $P = \{p_1, p_2, \cdots, p_7\}$($p_7$ that lies on s_9 is not drawn on Figure 1). After visiting edges from s_2 to s_4, the data points, p_1, p_2 and p_4, are added to $C_i.top$. In the next iteration, the edge s_9 is visited. Note: s_9 is on edge $e_4 = (n_3, n_4)$ over the road network G from the position 40 to 50. On position 40

Algorithm 2 MTR

```
1: let P_del be the set of removed data points;
2: let P_ins be the set of added data points;
3: for every data point p in P_del do
4:    suppose p lies on edge e;
5:    delete p from e (using an object index);
6:    for every C_i in that is influenced by e do
7:       if p in C_i.top then
8:          delete p from C_i.top;
9:          C_i.k ← +∞;
10: for every data point p in P_ins do
11:   suppose p lies on edge e;
12:   insert p into e (using object index);
13:   for every C_i that is influenced by e do
14:      update C_i.top and C_i.k using p;
15: for every C_i do
16:   if C_i.k is greater than its previous value then
17:      IRC(C_i);
```

and 50, its aggregate function values are 175 and 205, respectively. This information is recorded in $C_i.E$. Here, p_7 is over s_9, and therefore on e_4 in the road network G. Note: $pos_{e_4}(p_7) = 45$. IRC computes its value, for p_7, $175 + \frac{(205-175)\times(45-40)}{50-40} = 190$, which is smaller than the current $C_i.k = 200$ for the data point p_4. Therefore, p_4 is removed from $C_i.top$ and p_7 is added. The value $C_i.k$ is updated to be 190. Then, when visiting the next edge s_{10}, the smallest value is 205 which is larger than $C_i.k = 190$, and it stops. The top-3 for C_i is then $C_i.top = \{p_1, p_2, p_7\}$.

Monitor Top-k Result: Algorithm 2 shows top-k monitoring for a list of CANN queries. Here, the movement of a data point is considered as: first to delete it from P; then to insert a new data point into P. Let the set of deleted data points and the set of newly inserted data points be P_{del} and P_{ins}, respectively. (line 1-2). In Algorithm 2, in the first for statement (line 3-9), it updates $C_i.top$ if the deleted data points affects the top-k_i results. In the second for statement (line 10-14), it updates $C_i.top$ if the inserted data points affects the top-k_i results. In the first two for-statement, there is no need to scan $C_i.E$. In the third for-statement (line 15-17), if $C_i.k$ is changed and is greater than its previous $C_i.k$ value, it calls $IRC(C_i)$ to recompute the top-k_i results.

Reconsider the example (Figure 5) over the road network G (Figure 1), for the query $C_i = \mathsf{CANN}(\{q_1, q_2, q_3\}, 3, sum)$. First, suppose p_9 that lies on s_5 is inserted. The insertion of p_9 does not change the current top-3 results for C_i, as shown in Figure 5. Second, suppose p_7 is deleted which is in $C_i.top$. It leads to invoke $IRC(C_i)$ to recompute the top-k result. The new result is $C_i.top = \{p_1, p_2, p_4\}$. Then, suppose p_8 (lies on s_4) is inserted, which lies on the influenced edges (solid lines). It does not request recomputation. The new result is $C_i.top = \{p_1, p_2, p_8\}$.

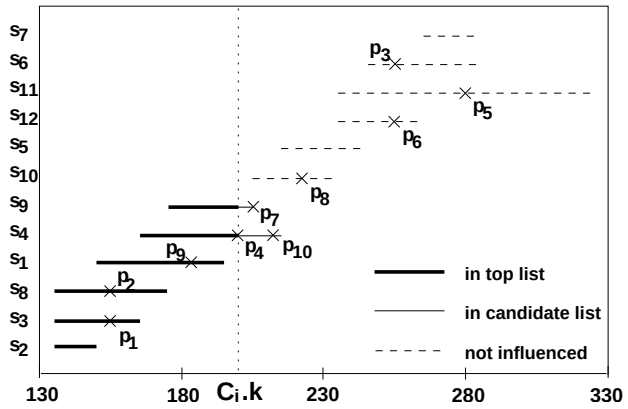


Fig. 6. Example for BUA

Algorithm 3 $ForwardUpdating(C_i)$

1: $e \leftarrow current(C_i.E)$;
2: **while** $e \neq \emptyset$ **and** $low(e) \leq C_i.k$ **do**
3: update $C_i.top$, $C_i.k$, $C_i.can$ with data points on e;
4: $e \leftarrow next(C_i.E)$;

4.3 Bidirectional Top-k Monitoring Algorithm

There are two drawbacks in the MTR algorithm. First, it needs to recompute top-k, for C_i, when $C_i.k$ increases (line 16-17) in MTR, which is time consuming. Second, it may scan some edges in $C_i.E$ which is unnecessary. In this section, we introduce a new incremental monitoring algorithm, to avoid the two drawbacks. The new algorithm keeps an additional structure called *candidate list*, denoted as $C_i.can$, for query C_i, which always stores the points lies on the influenced edges, but not in $C_i.top$. These points may be included in $C_i.top$, when some points in $C_i.top$ are deleted. As an example, consider Figure 6, for $P = \{p_1, p_2, \cdots, p_{10}\}$ (here p_7 to p_{10} is different from those in Figure 5) for a query $C_i = \mathsf{CANN}\ (\{q_1, q_2, q_3\}, 4, sum)$. Suppose $C_i.top = \{p_1, p_2, p_9, p_4\}$ and $C_i.can = \{p_7, p_{10}\}$. Below, we give two procedures, namely forward updating and backward updating, followed by the introduction to the new monitoring algorithm.

Forward Updating: As shown in Algorithm 3, this procedure is similar to that of IRC (Algorithm 1). The differences are as follows. First, it does not need the initialization step. Second, the candidate list is updated in line 3. The forward updating procedure repeat updating $C_i.top$, $C_i.k$, and $C_i.can$ when not all the influenced edges are visited.

Backward Updating: This procedure, as shown in Algorithm 4, removes from $C_i.can$ the data points on every edge e in $C_i.E$, if e is not influenced any more.

The BUA Algorithm: Our new incremental bidirectional updating algorithm (BUA) is shown in Algorithm 5. We explain it using the example in Figure 6. Suppose initially, the set of data points is $P = \{p_1, p_2, \cdots, p_8\}$, for a query $C_i = \mathsf{CANN}\ (\{q_1, q_2,$

Algorithm 4 $BackwardUpdating(C_i)$

```
1: e ← prev(C_i.E);
2: while low(e) > C_i.k do
3:    delete data points on e from C_i.can;
4:    e ← prev(C_i.E);
5: next(C_i.E);
```

Algorithm 5 BUA

```
1: let P_del be the set of removed data points;
2: let P_ins be the set of added data points;
3: for every point p in P_del do
4:    suppose p lies on edge e;
5:    delete p from e (using an object index);
6:    for every C_i where e is influenced do
7:       if p in C_i.top or p in C_i.can then
8:          update C_i.top, C_i.k, C_i.can by deleting p;
9: for every point p in P_ins do
10:   suppose p lies on edge e;
11:   insert p into e (using the object index);
12:   for every C_i where e is influenced do
13:      update C_i.top, C_i.k, C_i.can by inserting p;
14: for every C_i do
15:   if low(current(C_i.E)) ≤ C_i.k then
16:      ForwardUpdating(C_i);
17:   else
18:      BackwardUpdating(C_i);
```

$q_3\}$, 3, sum). After $ForwardUpdating(C_i)$ for initialization, we can get the initial result $C_i.top = \{p_1, p_2, p_4\}$ and $C_i.can = \{p_7\}$. Then, suppose $P_{ins} = \{p_{10}\}$ and $P_{del} = \{p_4\}$. When inserting p_{10}, it lies on the influenced edge s_4 but has an aggregate function value that is less than $C_i.k$. So p_{10} is inserted into the candidate list of C_i, $C_i.can$. When deleting p_4, it is in the $C_i.top$. So it is removed from the $C_i.top$ and p_7 will be moved from $C_i.can$ to the $C_i.top$. At this time, the lower bound of the current edge $low(s_{10}) \leq C_i.k$ (the aggregate function value of p_7). So the forward updating is invoked, s_{10} becomes influenced in C_i. The data point p_8 that lies on s_{10} is also added to $C_i.can$. The current result becomes $C_i.top = \{p_1, p_2, p_7\}$ and $C_i.can = \{p_8, p_{10}\}$. Note that in case of the MTR algorithm, the result of C_i have to be recomputed because $C_i.k$ increases. In the next time stamp, suppose $P_{ins} = \{p_9\}$ and $P_{del} = \phi$. After p_9 is used to update the result of C_i, it is added into $C_i.top$ and p_7 is moved from $C_i.top$ to $C_i.can$. At this time, we have the lower bound of current edge $low(s_5) > C_i.k$ (the aggregate function value of p_9). So the backward updating is invoked, and s_{10} is not influenced any more. The data point p_8 that lies on s_{10} is also removed from $C_i.can$. The result becomes $C_i.top = \{p_1, p_2, p_9\}$ and $C_i.can = \{p_7, p_{10}\}$.

4.4 Analysis

Suppose there are n nodes and m edges in the network, for each query $\mathsf{CANN}(Q, k, h)$, there are s segments in the query graph on average, and the average number of objects on each segment is o, the buffer size for each query is b. The average number of segments that influence the result of a query is r, we have $o \cdot r \geq k$. We assume that the objects are uniformly distributed on all edges and the portion of objects that changes at each timestamp is $\lambda (0 \leq \lambda \leq 1)$. For convenience, we ignore the cost for operations on the object index, which is not the dominate cost.

Lemma 4. *In the IRC algorithm, for each query, the time complexity to compute the initial results is $O(o \cdot r \cdot \log k)$, the memory used is $O(k + b)$ and the I/O cost is $O(\frac{r}{b})$.*

Proof. To compute the initial top-k result of a query, we need to retrieve all the objects that lie on the influence segments(i.e., the first r segments in the segment list of the query). The number of objects to be retrieved is $O(o \cdot r)$. Each object is used to update the top-k results, which can be implemented as a heap of size k. Each update can be done in $O(\log(k))$ time, so the total time complexity is $O(o \cdot r \cdot \log(k))$. For the memory cost, we need $O(b)$ to buffer the segment list, and $O(k)$ to store the results, so the total memory used is $O(k + b)$. We visit the first r segments in the segment list sequentially, so the I/O cost is $O(\frac{r}{b})$. □

Lemma 5. *In the MTR algorithm, with a probability of 0.5, the result of a query is needed to be recomputed at each timestamp. For the query that does not need to be recomputed, the time complexity for updating at each time stamp is $O(\lambda \cdot o \cdot r \cdot \log k)$ and no I/O operation is needed. The memory used for each query is the same as in IRC.*

Proof. The result of a query needs to be recomputed iff after the deletion and insertion steps, the new top-k result expires, or $C_i.k$ value for the query C_i increases. This case happens when, for the two sets P_{del} and P_{ins}, P_{del} contains more objects with cost smaller than the former $C_i.k$. The probability of this situation is 0.5 for the uniformly distributed objects. For each query that does not need re-computation, the time cost is the updates of $\lambda \cdot o \cdot r$ objects that lie on the influence segments, each update cost $\log(k)$ time as the same in the IRC algorithm, so the total time complexity for updating at each timestamp is $O(\lambda \cdot o \cdot r \cdot \log k)$. The influence segments keeps the same after the updating steps, so no I/O operation is needed on the segment list. The memory cost is also the same as the IRC algorithm. □

Lemma 6. *For the BUA algorithm, no re-computation is needed to update the result of a query at each timestamp, the time complexity for each query is $O(\lambda \cdot o \cdot r \cdot \log (o \cdot r))$. The memory used for each query is $O(o \cdot r + b)$. The I/O cost is $O(\frac{\lambda \cdot r}{b})$ in the worst case.*

Proof. For the BUA algorithm, it uses an extra candidate list for each query to record the candidate objects that lie on the influence segments but not in the top-k result of a query. For the $\lambda \cdot o \cdot r$ changed objects that lie on the influence segments, the cost for updating each object is $O(\log o \cdot r)$ by using a heap to record all the objects that lie on

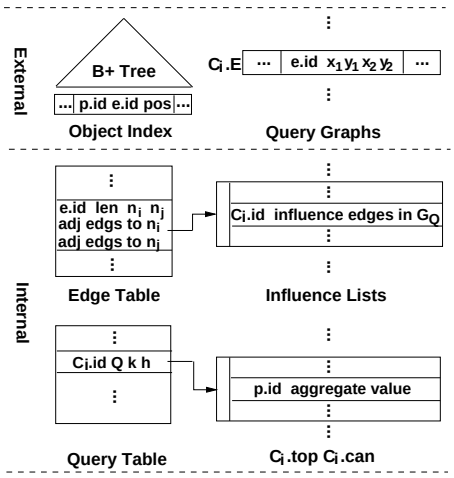


Fig. 7. Internal and External Structures

the influence segments(i.e., all the objects in the top-k result and candidate list). The total time complexity is $O(\lambda \cdot o \cdot r \cdot \log(o \cdot r))$. For the memory cost, in addition to the $O(b)$ buffer size, we need $O(o \cdot r)$ cost to record all the objects that lie on the influence segments. The total memory cost is $O(o \cdot r + b)$. For the I/O cost, consider the worst case, when $\lambda \cdot o \cdot r$ objects move out of the influence segments and no object moves in, or $\lambda \cdot o \cdot r$ objects move into the influence segments and no object moves out. In the first case, we need to visit $O(\frac{\lambda \cdot o \cdot r}{o}) = O(\lambda \cdot r)$ segments which cost $O(\frac{\lambda \cdot r}{b})$ I/O operations for the forward updating. In the second case, we also need to visit $O(\lambda \cdot r)$ that cost $O(\frac{\lambda \cdot r}{b})$ I/O operations for the backward updating. So the I/O cost is $O(\frac{\lambda \cdot r}{b})$ in the worst case. □

For the I/O cost of the BUA algorithm, in the average case, the number of objects that move into the influence segments is almost the same to the number of objects that move out, so the average I/O cost is very small in practice.

5 Implementation Details

In this section, we introduce the details for implementation including the data structures used and the storage model.We introduce three types of data structures that are constructed over the road network, data objects and queries respectively.

Edge Table For every edge e in the road network, we store in the edge table two part of information. The first part is about the network structure, i.e., the edge $e.id$, the two nodes n_i and n_j it connects, the length of the edge $len(e)$, and the lists of edges to n_i and n_j, this part can be used to construct the query graph G_Q of a CANN query. The second part is the influence list of e maintaining a set of queries that e influence along with the set of influence edges in G_Q. Using this part of information, we can fast retrieve all queries that is influenced by e.

Object Index Each object point p in the network can be represented as $(e.id, pos)$, where $e.id$ is the id of the edge it lies on, and pos is its position on e, i.e., $pos = pos_e(p)$. We use a index of a balanced tree to store all the object points in the network. It allows to retrieve all the objects that lies in a certain interval on a given edge e, or retrieve all the objects that over a certain edge s in a query graph of a CANN query. When the size of objects are large, the index can be stored external and a B+ tree can be used for storage.

Query Table The query table stores the set of queries. For every query C_i in the query table, tree parts of information are stored. The first part is the query descriptor, i.e., $C_i.id$, Q, k and h. The second part is the list of top-k objects $C_i.top$ along with $C_i.k$ and the candidate list $C_i.can$. The third part is the sorted edge list in the query graph $C_i.E$, which is a external data structure on which only sequential access and read operation is allowed. Each edge in the list is represented as $s = (e.id, x_1, y_1, x_2, y_2)$, where e is the edge on the original graph G, x_1 and x_2 are the start and end positions of s on e, y_1 and y_2 are the aggregate values on x_1 and x_2 respectively. Based on the sequential property, a buffer can be used for each query when processing.

The main internal and external data structures used for processing are illustrated in Figure 7.

6 Experimental Studies

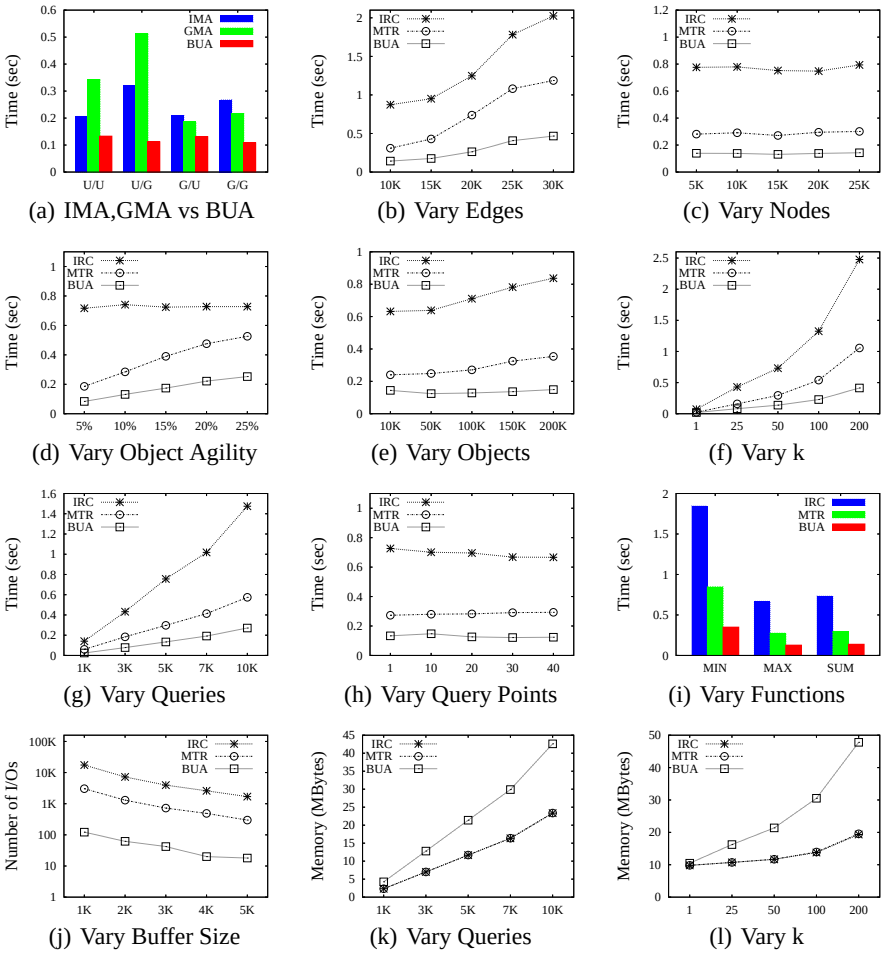
Parameter	Default	Range
Number of edges	25K	10, 15, 20, 25, 30 (K)
Number of nodes	20K	5, 10, 15, 20, 25 (K)
Number of queries	5K	1, 3, 5, 7, 10 (K)
Number of query points	20	1, 10, 20, 30, 40
Number of objects	100K	10, 50, 100, 150, 200 (K)
Query distribution	Uniform	Gaussian, Uniform
Object distribution	Uniform	Gaussian, Uniform
Top-k	50	1, 25, 50, 100, 200
Object agility	10%	5, 10, 15, 20, 25 (%)
Buffer size	2K	1, 2, 3, 4, 5 (K)
Function	SUM	MIN, MAX, SUM

Table 1. Parameters

We conducted extensive experimental studies to test the performance of our algorithms. All algorithms are implemented using C++. We use the road-map in the Maryland State in US extracted from US Census Bureau 2005 TIGER/Line.[3]. All the parameters including default values and ranges are listed in Table 1. Here, number of query points means the number of points in Q (i.e., $|Q|$) for each query, the query distribution is distribution of all query points, and the object agility is the percentage of objects that is changed per time stamp. The default graph is a subgraph of the above network with $20K$ nodes and $25K$ edges. When number of nodes varies, we use a subgraph of the

[3] Topologically Integrated Geographic Encoding and Referencing system: http://www.census.gov/geo/www/tiger/

network with the provided node number. When number of edges varies, we fix the node number to be 10K and generate a graph with the provided edge number. For each test that is to monitor the k-NN result, we process for 100 time stamps by generating the moving objects using the generator proposed in [3].We record the average performance for every time stamp. For the IRC algorithm, we mean to recompute the top-k result from scratch for every time stamp. Unless specified, we will use the default value for testing. All tests are conducted on a 2.8GHz CPU/1G memory PC running XP.



(a) IMA,GMA vs BUA (b) Vary Edges (c) Vary Nodes

(d) Vary Object Agility (e) Vary Objects (f) Vary k

(g) Vary Queries (h) Vary Query Points (i) Vary Functions

(j) Vary Buffer Size (k) Vary Queries (l) Vary k

Fig. 8. Experimental Results

Query Graph Construction: We first test the time to construct the query graph for each query. We vary the number of edges, number of nodes and number of query points, and record the time to construct the query graph in each test. The result is shown in Table 2, the time to construct query graph is small (less than 0.7 second) for all tests. As each of the three parameters increases, the response time will increase steadily.

$\|E\|(K)$/T(ms)	10/114	15/214	20/319	25/408	30/505
$\|N\|(K)$/T(ms)	5/64	10/155	15/254	20/336	25/437
$\|Q\|$/T(ms)	1/26	10/178	20/343	30/506	40/675

Table 2. Time to construct query graph

IMA,GMA vs BUA: With $|Q|$ fixed to be 1, we test the efficiency for IMA, GMA, and BUA algorithms. For each algorithm, we combine the different distributions(i.e., Gaussian and Uniform distribution) for queries and objects (e.g., U/G means the queries are uniformly distributed and the objects are in Gaussian distribution) with all other parameters setting to be the default values. As illustrated in Figure 8(a), our BUA algorithm always performs best and changes for distribution of both the queries and objects will not influence the efficiency of BUA algorithm much.

Network: We vary the number of edges and number of nodes for the network with an average of 4 objects on each edge and test the average processing time for IRC, MTR and BUA algorithms in each time stamp. We report our result in Figure 8(b) and Figure 8(c). For each test, the MTR algorithm is about 2-3 times faster than the IRC algorithm, and the the BUA algorithm is 2-4 times faster than the MTR algorithm. When the number of edges increases, the processing time for all three algorithms will increase, because as the network becomes denser, the number of influence edges will increase. When the number of nodes increases, the processing time for all three algorithms do not change much, because both the density of network and density of objects will not change as the network increases.

Objects: Figure 8(d) and Figure 8(e) shows the average processing time per time stamp for IRC, MTR and BUA algorithms when the object agility or the number of objects varies. When the object agility increases, the processing time for both MTR and BUA will increase steadily while IRC is not influenced because it will always computes each query from scratch at every time stamp. When the number of objects increases, the density of objects becomes larger, which increases the processing time. But the number of influenced edges will decrease, which decreases the processing time. We can see from Figure 8(e) that when the object number is larger than $50K$, the processing time for all the three algorithms all increase slowly.

Queries: There are mainly 4 parameters for the query: the top-k value, number of queries, number of query points in each query(i.e., $|Q|$), and the type of aggregate function for the query. In Figure 8(f) and Figure 8(g), when k increases or the number of queries increases, the processing time for IRC, MTR and BUA algorithms will increase steadily. In Figure 8(h), when the number of query points in each query increases, the processing time for all three algorithms will not influence much, because at one hand, the number of edges in each query's query graph in will increase which raise the complexity of algorithm; at the other hand, the length of edges in each query's query graph becomes shorter, and the objects lies on the influence edges become less, which lower the complexity. In Figure 8(i), we see that the MIN function consumes more for all three algorithms. It is because for MAX and SUM function, the best objects retrieved is more centralized for each query, while in the MIN function, each query point in a query can be considered as a center for the distribution of the top objects.

Total I/Os: We vary the buffer size used for every query C in the corresponding edge list $C.E$, and study the number of I/Os for IRC, MTR and BUA algorithms for each time stamp. As shown in Figure 8(j), as the buffer size increases, the number of I/Os will decrease steadily. The MTR costs about $\frac{1}{5}$ I/Os of IRC while BUA costs about $\frac{1}{20}$ of MTR, which is rather small, because the pointer for each query only moves forward of backward incrementally.

Memory: We finally test the memory used for algorithms of IRC, MTR and BUA. When the number of queries and top-k value vary, the result is shown in Figure 8(k) and Figure 8(l). As the query number or k increases, the memory used will increase steadily, for all three algorithms. The memory cost of IRC and MTR is the same as analyzed in Section 4.4. The memory cost for BUA is about 1.1 to 2.4 times of IRC.

7 Related Work

In this section, we survey k-NN search over road networks in two categories, namely, snapshot approaches and continuous monitoring approaches.

Snapshot approaches: Shahabi et al. in [4] applied an embedding technique to transform a road network to a high dimensional space, and used the Minkowski metrics for distance measurement in the embedded space. Jensen et al. in [5] proposed a foundation data model and a system prototype for k-NN queries in road networks. Shekhar et al. in [6] addressed the problem of finding the in-route nearest neighbor (IRNN). Papadias et al. in [7] proposed an architecture that integrates network and Euclidean information for query processing in spatial network databases. Tao et al. in [8] studied the time-parameterized k-NN queries when query points and objects change in certain speed and directions. Kolahdouzan et al. in [9] proposed to find the nearest points of interest to all the points on a path over road networks. They also performed k-NN over spatial networks in [10] based on the pre-computed first order Voronoi diagram. Yiu et al. in [17] first studied the aggregate nearest neighbor query in road networks, which explored the network around the query points until the aggregate nearest neighbors are discovered. UNICONS [11] developed a search algorithm which answers NN queries at any point of a given path. Huang et al. in [12] presented a versatile approach to k-NN computation in spatial networks using the island which is a sub-network in a certain area. Hu et al. in [13] proposed an approach that indexes the network topology based on a set of interconnected tree-based structures. Huang et al. in [14] focused on caching the query results in main memory and subsequently reusing these for query processing when there are multiple k-NN queries over a road network. Almeida et al. in [15] proposed a storage schema with a set of index structures to support Dijkstra based algorithms for k-NN queries in road networks. Deng et al. in [16] considered the problem of efficient multi-source skyline query processing in road networks.

Continuous monitoring approaches: In recent years, more works focused on continuous monitoring of NN queries over road networks. Ku et al. in [17] studied the adaptive NN queries in travel time networks. It developed a local-based greedy nearest neighbor algorithm and a global-based adaptive nearest neighbor algorithm that both utilize real-time traffic information to maintain the search results. Mouratidis et al. in [1] focused on monitoring nearest neighbors in highly dynamic scenarios.

8 Conclusion

In this paper, we studied a new problem (CANN query) that is to monitor k-NN objects over a road network from multiple query points to minimize an aggregate distance function with respect to the multiple query points. In order to reduce the cost of network distance computing, we proposed a new approach that computes a query graph offline for a CANN query. With the help of the query graph, the cost of computing aggregate function values for any possible data points on the road network is significantly reduced. In addition, we proposed two algorithms to monitor CANN queries. We conducted extensive experimental studies over large road networks and confirmed the efficiency of our algorithms.

Acknowledgment: This work was supported by a grant of RGC, Hong Kong SAR, China (No. 418206).

References

1. Mouratidis, K., Yiu, M.L., Papadias, D., Mamoulis, N.: Continuous nearest neighbor monitoring in road networks. In: VLDB. (2006) 43–54
2. Yiu, M.L., Mamoulis, N., Papadias, D.: Aggregate nearest neighbor queries in road networks. IEEE Trans. Knowl. Data Eng. **17**(6) (2005) 820–833
3. Brinkhoff, T.: A framework for generating network-based moving objects. GeoInformatica **6**(2) (2002) 153–180
4. Shahabi, C., Kolahdouzan, M.R., Sharifzadeh, M.: A road network embedding technique for k-nearest neighbor search in moving object databases. In: ACM-GIS. (2002) 94–10
5. Jensen, C.S., Kolárvr, J., Pedersen, T.B., Timko, I.: Nearest neighbor queries in road networks. In: GIS. (2003) 1–8
6. Shekhar, S., Yoo, J.S.: Processing in-route nearest neighbor queries: a comparison of alternative approaches. In: GIS. (2003) 9–16
7. Papadias, D., Zhang, J., Mamoulis, N., Tao, Y.: Query processing in spatial network databases. In: VLDB. (2003) 802–813
8. Tao, Y., Papadias, D.: Spatial queries in dynamic environments. ACM Trans. Database Syst. **28**(2) (2003) 101–139
9. Kolahdouzan, M.R., Shahabi, C.: Continuous k-nearest neighbor queries in spatial network databases. In: STDBM. (2004) 33–40
10. Kolahdouzan, M.R., Shahabi, C.: Voronoi-based k nearest neighbor search for spatial network databases. In: VLDB. (2004) 840–851
11. Cho, H.J., Chung, C.W.: An efficient and scalable approach to cnn queries in a road network. In: VLDB. (2005) 865–876
12. Huang, X., Jensen, C.S., Saltenis, S.: The islands approach to nearest neighbor querying in spatial networks. In: SSTD. (2005) 73–90
13. Hu, H., Lee, D.L., Xu, J.: Fast nearest neighbor search on road networks. In: EDBT. (2006) 186–203
14. Huang, X., Jensen, C.S., Saltenis, S.: Multiple k nearest neighbor query processing in spatial network databases. In: ADBIS. (2006) 266–281
15. de Almeida, V.T., Güting, R.H.: Using dijkstra's algorithm to incrementally find the k-nearest neighbors in spatial network databases. In: SAC. (2006) 58–62
16. Deng, K., Zhou, X., Shen, H.T.: Multi-source skyline query processing in road networks. In: ICDE. (2007)
17. Ku, W.S., Zimmermann, R., Wang, H., Wan, C.N.: Adaptive nearest neighbor queries in travel time networks. In: GIS. (2005) 210–219