This is the Pre-Published Version

# Aggregate Nearest Neighbor Queries in Road Networks

Man Lung Yiu[†] Nikos Mamoulis[†] Dimitris Papadias[‡]

[†]Department of Computer Science
University of Hong Kong
Pokfulam Road, Hong Kong
{mlyiu2,nikos}@cs.hku.hk

[‡]Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
dimitris@cs.ust.hk

Correspondence Author:

Nikos Mamoulis
Department of Computer Science
University of Hong Kong
Pokfulam Road
Hong Kong
Tel (852)28578243
Fax (852)25598447
E-mail: nikos@cs.hku.hk

**Keywords:** H.2.4.h Query processing, H.2.4.k Spatial databases, H.2.8.o Spatial databases and GIS, J.9.a Location-dependent and sensitive

# Aggregate Nearest Neighbor Queries in Road Networks

Man Lung Yiu[†], Nikos Mamoulis[†], and Dimitris Papadias[‡]

[†]Department of Computer Science
University of Hong Kong
Pokfulam Road, Hong Kong
{mlyiu2,nikos}@cs.hku.hk

[‡]Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
dimitris@cs.ust.hk

**Abstract**

Aggregate nearest neighbor queries return the object that minimizes an aggregate distance function with respect to a set of query points. Consider, for example, several users at specific locations (query points) that want to find the restaurant (data point), which leads to the minimum sum of distances that they have to travel in order to meet. We study the processing of such queries for the case where the position and accessibility of spatial objects are constrained by spatial (e.g., road) networks. We consider alternative aggregate functions and techniques that utilize Euclidean distance bounds, spatial access methods, and/or network distance materialization structures. Our algorithms are experimentally evaluated with synthetic and real data. The results show that their relative performance depends on the problem characteristics.

## 1 Introduction

In many applications that manage spatial data (e.g., location-based services) the position and accessibility of spatial objects are constrained by spatial networks. In such cases the actual distance between two objects corresponds to the length of the shortest path connecting

them in the network. Recently, there has been an increasing interest in processing nearest neighbor queries over road networks [14, 12, 7, 17]. Given a set $P$ of interesting objects (e.g., facilities) and a location $q$, the *nearest neighbor* query returns the nearest object of $q$ in $P$. Formally, the query retrieves a point $p \in P$, such that $d(p, q) \leq d(p', q), \forall p' \in P$, where $d()$ is a distance function (i.e., the network distance in our setting).

In this paper, we study an interesting generalization of nearest neighbor search. Given a set $P$ of interesting objects, a set $Q$ of query points, and an aggregate function $f$ (e.g., sum, max) an *aggregate nearest neighbor* (ANN) query retrieves the object $p$ in $P$, such that $f\{d(p, q_i), \forall q_i \in Q\}$ is minimized. Consider the example of Figure 1, where a set of interesting objects $P = \{p_1, p_2, p_3, p_4\}$ (e.g., restaurants) and a set of query points $Q = \{q_1, q_2\}$ (e.g., users) lie on the edges of a road network. The numbers on the edges represent travel cost (in terms of distance, time etc.). An ANN with $f = \text{sum}$ as aggregate function retrieves the point $p_i \in P$ that minimizes the total cost required by $q_1, q_2$ to meet at $p_i$, when traveling along network edges. The result of this ANN query is $p_3$ with aggregate distance $(6+4+4)+(1+1) = 16$. Another important aggregate function is $f = \max$, which minimizes the maximum (as opposed to the total) distance traveled by any user. For instance, assume that the costs of the network edges correspond to travel time, and the two users want to meet as fast as possible at a restaurant $p_i$. The result of this ANN query is object $p_1$ with $\max\{d(p_1, q_i)\} = d(p_1, q_1) = 12$.

ANN queries are a natural way to express requests by *groups* of mobile users, who want to *optimize* their routes according to an aggregate function applying on the traveling distances. Apart from the meeting-restaurant example, other application instances include (i) establishing a meeting station for members of a new church, based on its distances from their homes, (ii) selecting the location of a touristic office based on its distances to attractions in a city. ANN queries find application in geographic information systems, location-based services, navigation systems, mobile computing systems, data mining (e.g., clustering objects in a road network [19]).

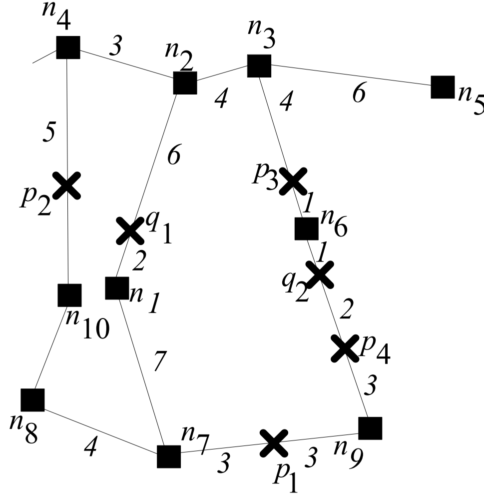



Figure 1: Example of ANN queries

The contributions of the paper can be summarized as follows:

- We propose and solve ANN query processing in the context of large road networks. To the best of our knowledge, this constitutes the first comprehensive study on this important problem.

- We develop three methods for ANN queries, utilizing connectivity information (preserved by the network) and spatial locality. The first algorithm can be applied when the Euclidean distance between any two network nodes lower bounds their network distance. It incrementally retrieves Euclidean ANN using a spatial index (R–tree) and then computes their aggregate *network* distance, until the query results are guarranteed to be found. We also propose two adaptations of top-$k$ algorithms for this problem, based on the observation that ANN queries combine distances from multiple sources (and therefore, they can be thought of as top-$k$ queries [3]).

- We conduct an extensive experimental study to evaluate the efficiency of the proposed algorithms on real road networks for various problem characteristics. In addition, we explore the efficiency of the proposed algorithms in the presence of data structures that

materialize shortest path distances between network nodes. The results show that the best technique depends on the problem input (e.g., underlying network, edge weights, aggregate function).

The rest of the paper is organized as follows. Section 2 defines the problem and discusses related work. Sections 3 and 4 present our methodology. Section 5 discusses interesting variants of ANN queries. The proposed methods are experimentally compared in Section 6. Finally, Section 7 concludes the paper.

# 2 Definitions and Background

We first present the network distance definitions that we follow throughout the paper. Then, we overview related work on shortest path algorithms, distance materialization, nearest neighbor search, and top-$k$ queries.

## 2.1 Problem definition

A *network* is an undirected weighted graph $G = (V, E, W)$ where $V$ is the set of vertices (i.e., nodes), $E$ is the set of edges, and $W : E \rightarrow \mathbb{R}^+$ associates each edge to a positive real number (i.e., the weight or cost of the edge). Interesting *objects* (i.e., data points) are located on edges $e \in E$. The position of an object $p$ lying on the edge $(n_i, n_j)$ can be expressed by the triplet $\langle n_i, n_j, pos \rangle$ where $pos \in [0, W(e)]$ is the distance of $p$ from node $n_i$. To ensure that the location of the object is expressed unambiguously by one triplet, we require that $n_i < n_j$ (assuming a total ordering of node labels). Figure 2 shows an example of a network, where nodes are denoted by squares, and every edge is associated with a distance label. Each object (denoted by a cross) lies on exactly one edge.[1] For instance, $p_2$ lies on $(n_1, n_3)$

[1] In real-life problems, some objects may not lie on edges of the network. In such cases, we assume that the object is represented by the closest position on the network [12]. If a data point is on the intersection of multiple edges (i.e., on a network node) it may have multiple equivalent representations, out of which only one is stored.

and it is 1.0 units away from $n_1$ along the edge. Therefore, its position can be expressed by $\langle n_1, n_3, 1.0\rangle$.

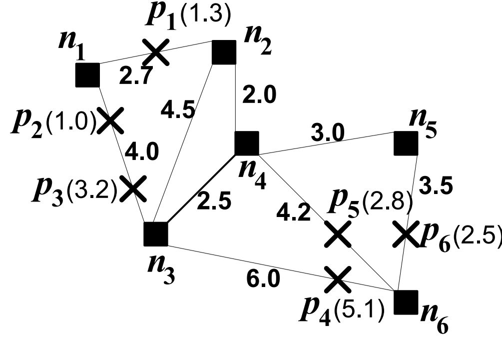



Figure 2: Example of a spatial network

Let $p_i$ and $p_j$ be two points at $\langle n_a, n_b, pos_{p_i}\rangle$ and $\langle n_c, n_d, pos_{p_j}\rangle$, respectively. If $n_a = n_c$ and $n_b = n_d$ (i.e., $p_i$ and $p_j$ lie on the same edge), the *direct distance* $d_L(p_i, p_j)$ between $p_i$ and $p_j$ is defined as $|pos_{p_i} - pos_{p_j}|$ ; otherwise, it is $\infty$. For instance, in Figure 2, $d_L(p_2, p_3) = 2.2$ (the points lie on the same edge) and $d_L(p_2, p_1) = \infty$. The direct distance between a point and a network node is defined only when the point lies on an edge adjacent to the node. Given a point $p$ with position $\langle n_a, n_b, pos_p\rangle$, the *direct distance* $d_L(p, n_a)$ between $p$ and $n_a$ is $pos_p$. Similarly, the direct distance $d_L(p, n_b)$ is $W(n_a, n_b) - pos_p$. For example, $d_L(p_1, n_1) = 1.3$ and $d_L(p_1, n_2) = 2.7 - 1.3 = 1.4$. Notice that the direct distance of two points on the same edge is not necessarily the shortest distance between them. For instance, consider an edge $(n_x, n_y)$ with $W(n_x, n_y) = 10$ and assume that $n_x$ and $n_y$ are connected to $n_z$, such that $W(n_x, n_z) = 2$ and $W(n_z, n_y) = 2$. For points $p_i = \langle n_x, n_y, 1.0\rangle$ and $p_j = \langle n_x, n_y, 9.0\rangle$, $d_L(p_i, p_j) = 8$, whereas the distance of the path from $p_i$ to $p_j$ via $n_x$, $n_z$, and $n_y$ is $1 + 2 + 2 + 1 = 6$. We assume that the edges are bidirectional and that the direct distance is symmetric, i.e., $d_L(p_i, p_j) = d_L(p_j, p_i)$ and $d_L(p, n_i) = d_L(n_i, p)$.

The *network distance* $d(n_i, n_j)$ of nodes $n_i$ and $n_j$, is defined as the minimum sum of weights of any path between them. In Figure 2, $d(n_2, n_6) = 6.2$. Given points $p_i$ and $p_j$, where $p_i$ lies on edge $(n_a, n_b)$ and $p_j$ lies on the edge $(n_c, n_d)$, the *network distance* $d(p_i, p_j)$ can be

computed as $\min_{x\in\{a,b\},y\in\{c,d\}}(d_L(p_i,n_x)+d(n_x,n_y)+d_L(n_y,p_j))$, if $p_i$ and $p_j$ lie on different edges; otherwise, $d(p_i,p_j)$ is the minimum of the previous quantity and $d_L(p_i,p_j)$. The network distance is symmetric and satisfies the inequality $d(p_i,p_j) \leq d(p_i,p_k)+d(p_k,p_j)$ (because $d(p_i,p_j)$ is the shortest distance between $p_i$ and $p_j$).

Let $p$ be a point and $Q$ be a set of query points that lie on the network. Then, an *aggregate network distance function* $d_{agg}(p,Q)$ is defined as $agg\{d(q_i,p),\forall q_i \in Q\}$, where $agg$ is an aggregate function that applies on sets of numbers (e.g., sum, max, etc.); $d_{agg}(p,Q)$ is *monotone* if $\forall p,p'$ $(\forall q_i \in Q, d(p,q_i) \geq d(p',q_i)) \Rightarrow d_{agg}(p,Q) \geq d_{agg}(p',Q)$. In this paper, we only consider monotone functions. We call each $d(p,q_i)$ a *component distance* (implying the query component $q_i$). Two popular aggregate functions are: $d_{sum}(p,Q) = \sum_{\forall q_i \in Q} d(p,q_i)$; and $d_{max}(p,Q) = \max_{\forall q_i \in Q} d(p,q_i)$. For instance, in Figure 1, for $Q=\{q_1,q_2\}$, $d_{sum}(p_1,Q)=20$ and $d_{max}(p_1,Q)=12$.

Given a set of *query points* $Q$, a set of *interesting objects* $P$ ($P$ and $Q$ are located on the network), and an aggregate distance function $d_{agg}(p,Q)$, an *aggregate* $k$*-nearest neighbor query* $k$-$ANN_{agg}(P,Q)$ retrieves $S \subset P$, such that $|S|=k$ and $d_{agg}(p,Q) \leq d_{agg}(p',Q), \forall p \in S, p' \in P-S$ for some $k<|P|$. E.g., in Figure 1, for $Q=\{q_1,q_2\}$, 1-$ANN_{sum}(P,Q)=\{p_3\}$ (with $d_{sum}(p_3,Q)=16$). Although ANN queries can have multiple results with the same quality, only one of them is reported for simplicity.

## 2.2 Related work

Our problem is closely related to shortest path computation in large graphs. Given a source $n_s$ and a destination node $n_d$, Dijkstra's algorithm [2] expands the network from $n_s$ until $n_d$ is reached. A priority queue $H$ is used to organize the neighbors of the nodes found so far, so that intermediate nodes from $n_s$ to $n_d$ are visited in increasing order of their distances from $n_s$. A shortcoming of the algorithm is that it may visit many nodes far from the shortest path. A* search (e.g., see [15]) alleviates this effect using lower distance bounds. Assume that the Euclidean distance $d^{\mathcal{E}}(n_i,n_j)$ lower-bounds the network distance $d(n_i,n_j)$.

A* organizes the nodes $n_i$ to be visited by $L_d(n_i) = d(n_s, n_i) + d^{\mathcal{E}}(n_i, n_d)$. $L_d(n_i)$ restricts the shortest path distance from $n_s$ to $n_d$, via $n_i$. The node with the minimum $L_d(n_i)$ is visited next and its neighbors are added on the heap $H$. The process continues until the destination node $n_d$ is popped from $H$.

Shortest path search can be accelerated by materializing the network distance between every pair of nodes. The high storage cost of fully materialized distances makes this approach infeasible even for networks of moderate sizes. For instance, for a graph of $|V|$=100K nodes we need to store $|V|(|V|-1)/2 \cong 5 \times 10^9$ distances. *HiTi* [9] and *HEPV* [8] avoid the extreme space requirements by *partial* materialization. *HiTi* first partitions the network into subgraphs that are small enough to fit in memory. These subgraphs can be abstracted as network nodes which are recursively grouped at the higher level. At each level, all the edges that connect *boundary* nodes of different subgraphs at the lower level are explicitly stored together with the corresponding distance. To compute a shortest path distance between two given nodes, it suffices to find the most detailed subgraphs that contain the nodes and use the materialized information stored in higher-level nodes of the two search paths. *HEPV* performs a similar hierarchical partitioning, but pre-computes and stores more network distances.

ANN queries are also closely related to nearest neighbor search and related forms of spatial information processing over networks. [12] propose a storage scheme for objects that lie on a network, as well as algorithms for range selections, nearest neighbor queries, and distance joins. The Euclidean distance is employed (like in A* search) to guide search and prune parts of the network. In addition, R–trees are used to efficiently compute Euclidean distance bounds. [14] transform the spatial network to a high dimensional space and use simple distance functions to approximate the network distance. However, the query results are only approximate and the storage overhead high, so that the method cannot handle large networks. [7] discuss nearest neighbor queries for objects moving in a network. [17] study the problem of finding nearest neighbors along a given route instead from a single query

point. [5] propose methods for solving shortest path queries with spatial constraints.

[11] solve ANN queries considering only Euclidean distance and the sum function. The methods proposed there utilize R–trees and distance bounds to converge to the result, by minimizing the I/O and computational cost. In this paper, we study the problem considering the network distance and additional aggregate functions. Our work is essentially different due to the non-trivial computation of the network distances. For instance, given a point $p$ on the plane and a set of query objects $Q$, it takes $O(|Q|)$ time to compute the aggregate Euclidean distance from $p$ to $Q$. On the other hand, the aggregate network distance requires expensive network traversal.

Finally, since our techniques aggregate distances from multiple sources, they are related to top-$k$ queries [3]. Consider a database that contains multiple orderings for a given set $P$ of objects (e.g., images) with respect to their similarity to a query object $q$, based on different criteria (e.g., color, texture). For example, ordering $O_1$ could rank the objects in $P$ based on color similarity with $q$, and $O_2$ could rank them based on texture similarity with $q$. The top-$k$ query retrieves the $k$ objects with the maximal aggregate similarity to $q$. We can express ANN queries as top-$k$ queries by sorting the objects in $P$ based on their distances from each $q_i \in Q$ and then combine the sorted streams to derive the final result.

# 3 Storage and Distance Computation

Before presenting our techniques for processing ANN queries, we briefly describe the storage architecture for the network. Next, we show how to compute aggregate distances of points on network edges. Finally, we propose a transformation that allows the application of Euclidean distance bounds for pruning the search space.

## 3.1 Disk-based storage of the network

We use a disk-based storage model that groups network nodes based on their connectivity and distance, as in [16, 12, 19]. Figure 3 contains a graphical illustration of the files and indexes for the network of Figure 2. Adjacency lists and points are stored in two separate flat files. The header of the adjacency list file contains, for each node $n$, a pointer to the corresponding list. Furthermore, recall that the A* algorithm requires computation of the Euclidean distance between network nodes. For this purpose, the header also stores the coordinates of $n$. The adjacency list of $n$ keeps the neighboring nodes of $n$ together with their edge weight. Points on the same edge $(n_x, n_y)$ form a *point group* and are kept together in the file for data points. In addition, this file stores the node ids $n_x$, $n_y$ and the direct distance $d_L(p, n_x)$ of every point on $(n_x, n_y)$. Each edge in the adjacency list has a pointer to the corresponding *point group* (if any). In this way, network traversal algorithms can efficiently retrieve the points on the adjacent edges (and nodes) to a given node $n$. Finally, a sparse B$^+$–tree is built on top of the *point file*. Thus, given a point $p$, we can efficiently find (i) the edge where $p$ lies and (ii) all other points on this edge.

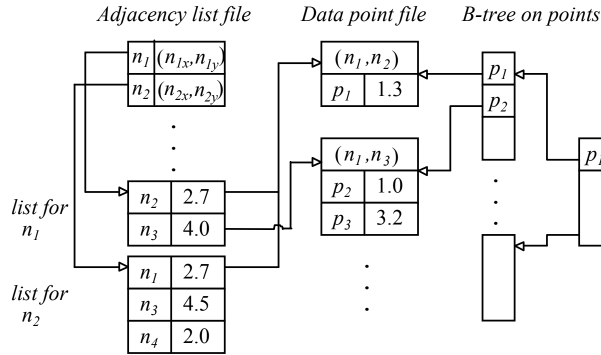



Figure 3: Disk-based storage representation

An issue that deserves further clarification refers to the grouping of lists in the adjacency list file. In particular, the lists of neighboring nodes should be stored in the same disk-page in order to minimize the I/O cost during the graph traversal. [16, 18] propose several methods for disk-based organization of network nodes and/or adjacency lists. However,

these algorithms (based on graph partitioning) are complex and expensive. Instead, we follow a simple technique which, as we conjecture, should achieve similar performance. We first partition the graph using the spatial coordinates of the nodes (e.g., by a $c \times c$ grid) such that each partition fits in memory. For every partition, a random node $n$ is first chosen and its adjacency list is written into the current disk page. The process is repeated for $n$'s neighbors; i.e., the part of the network in the partition is traversed in a breadth-first manner, packing adjacency lists until the page is full. For the next page, breadth-first traversal is re-initialized for the next unpacked node in order, etc. In this way, nodes in the same page are close in the network with high probability.

Finally, some of the proposed algorithms require the efficient indexing of points based on their spatial coordinates and their clustering on the network edges. For this purpose, we may need to build an R–tree on top of the point file, as elaborated later.

## 3.2 Aggregate distances on edges

Given a network edge $(n_x, n_y)$ and the component distances of $n_x$ and $n_y$, we can compute $d_{agg}(p, Q)$ for each point on $(n_x, n_y)$. While solving ANN queries, it is useful to know the minimum possible $d_{agg}(p, Q)$ for any $p$ on $(n_x, n_y)$, so that we can prune the edge if it cannot contain any better ANN (without accessing the point file). Towards this goal, we study the possible range of $d_{agg}(p, Q)$ as a function of $d(p, q_i)$, $\forall q_i \in Q$. We first discuss how $d(p, q_i)$ ranges depending on (i) whether $q_i$ lies on $(n_x, n_y)$ and (ii) $q_i$'s distance from $n_x$ and $n_y$. Consider for example, a part of the network, as shown in Figure 4a and three query points $q_1, q_2, q_3$. We have three distance distributions for $d(p, q_i)$ all of which are piecewise linear functions. In the first case, $q_i$ is not on the edge and $d(n_y, q_i) = d(n_x, q_i) + W(n_x, n_y)$. In other words, the shortest path from $q_i$ to $n_y$ passes through $n_x$ and the distance of any point $p$ along $(n_x, n_y)$ increases linearly and monotonically. In the example of Figure 4, $q_1$ corresponds to such a query point. In the symmetric case, $d(n_x, q_i) = d(n_y, q_i) + W(n_x, n_y)$ and $d(p, q_i)$ linearly decreases. The second case applies when $q_i$ lies on edge $(n_x, n_y)$ (e.g., $q_2$),

so that $d(p, q_i)$ first decreases and then increases linearly. Finally, the third case applies for $|d(n_y, q_i) - d(n_x, q_i)| < W(n_x, n_y)$ (e.g., $q_3$), where $d(p, q_i)$ first increases and then decreases linearly.

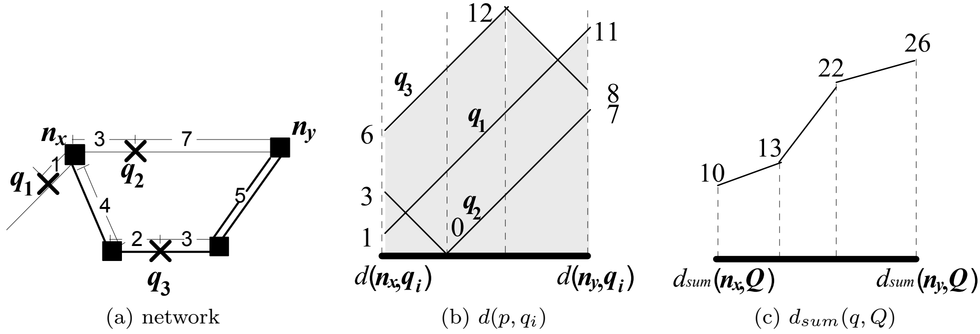



Figure 4: $d_{agg}(p, Q)$ along edge $(n_x, n_y)$

Thus, given an edge $(n_x, n_y)$ and a set of query points $Q$, we can define a sequence of *split points* on $(n_x, n_y)$, consisting of the edge endpoints and the extrema of the second and third cases for each $q_i \in Q$. In Figure 4b, the split points are denoted by the dotted lines. By using the split points, we can find the position on the edge that minimizes $d_{sum}$ and $d_{max}$. Figure 4c shows the distribution of $d_{sum}(p, Q)$ for our example. Note that the distance function between two adjacent split points can remain constant, increase or decrease linearly. Therefore, the optimal location can be found by computing only the aggregate distances at the split points. $d_{max}(p, Q)$ along edge $(n_x, n_y)$ corresponds to the upper contour of the shaded area in Figure 4b.

## 3.3 Using Euclidean bounds

If the edge weights correspond to the Euclidean distance of the connected nodes, the network distance $d(n_i, n_j)$ between any pair of nodes is lower bounded by their Euclidean distance $d^{\mathcal{E}}(n_i, n_j)$. We can generalize this property for any pair of points on the network.

**Lemma 1** *Let $G = (V, E, W)$ be a network, such that for each edge $(n_x, n_y) \in E$, $d^{\mathcal{E}}(n_x, n_y) \leq W(n_x, n_y)$. For any two points $p$, $q$ on the network, $d^{\mathcal{E}}(p, q) \leq d(p, q)$.*

**Proof.** The network distance $d(p, q)$ is defined by the sum of weights along the shortest path from $p$ to $q$. This quantity is no smaller than the sum of the Euclidean distances of the edges, which in turn is no smaller than $d^{\mathcal{E}}(p, q)$ (triangular inequality). ∎

In addition, we can show that the Euclidean aggregate distance $d^{\mathcal{E}}_{agg}(p, Q)$ from a data point to a set of query points is a lower bound for the corresponding aggregate network distance $d_{agg}(p, Q)$. This property is used by our first ANN algorithm.

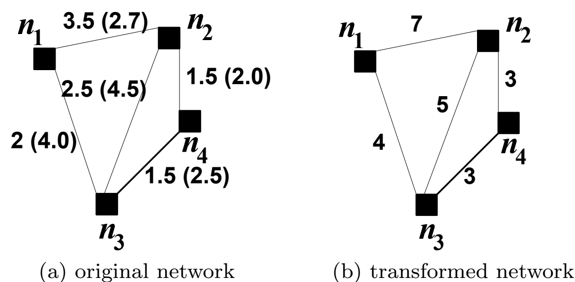
**Lemma 2** *Let $G = (V, E, W)$ be a network, such that for each edge $(n_x, n_y) \in E$ $d^{\mathcal{E}}(n_x, n_y) \leq W(n_x, n_y)$. Let $Q$ be a set of query points, $p$ be a point on the network and agg be a monotone aggregate function. The Euclidean aggregate distance $d^{\mathcal{E}}_{agg}(p, Q)$ is a lower bound of the network aggregate distance $d_{agg}(p, Q)$.*

**Proof.** The lemma holds since $d^{\mathcal{E}}(p, q_i) \leq d(p, q_i)$ for each $q_i \in Q$ (by Lemma 1) and by the definition of monotone aggregate functions. ∎

Nevertheless, edge weights do not necessarily correspond to Euclidean distances between the connected nodes. For instance, consider a city road network, where weights correspond to the cost of traveling along the corresponding road segments. Roads that cross tunnels or bridges are more expensive due to toll fees. As another example, assume that the weights reflect the time required to cross a road segment. In this case, traffic-congested roads have much higher weight than freeways of the same length.

We deal with networks of arbitrary weights, by normalizing them so that (i) for each edge $(n_x, n_y) \in E$, $d^{\mathcal{E}}(n_x, n_y) \leq W(n_x, n_y)$, (ii) the correctness of shortest path and nearest neighbor computations is not affected by the change, and (iii) the Euclidean distance lower bounds are as tight as possible. This normalization is based on the *scaling ratio* defined as: $r = \max\{\frac{d^{\mathcal{E}}(n_x, n_y)}{W(n_x, n_y)}, \forall (n_x, n_y) \in E\}$. Figure 5a shows an example of a network. The edges are tagged by weights, followed by the Euclidean distance between the corresponding nodes enclosed in parentheses. Note that the weights of the network are not proportional to the Euclidean distances. First we compute $r = \max\{\frac{d^{\mathcal{E}}(n_x, n_y)}{W(n_x, n_y)}\} = \frac{d^{\mathcal{E}}(n_1, n_3)}{W(n_1, n_3)} = 2$ and then we

multiply all weights by $r$, resulting in the network of Figure 5b. The shortest path between any pair of points on the network remains the same, since all weights are multiplied by the same factor. In addition, the Euclidean distance bounds hold, permitting the application of Lemmas 1 and 2 by shortest path and (aggregate) nearest neighbor search algorithms. Note, however, that if the Euclidean distance length of edges is not proportional to their weights, Euclidean distance (after normalization) becomes a loose lower bound, degrading the performance of ANN search methods that use it, as we verify later.




(a) original network (b) transformed network

Figure 5: Transformation of network weights

# 4 Algorithms for ANN queries

Given a set of interesting points $P$ and a set of query points $Q$ on a network, a brute-force method to evaluate a $k$-$ANN_{agg}(P,Q)$ query is to traverse the network from each $q_i \in Q$, compute $d(p,q_i)$ for each $p \in P$ and, finally, sort the points in $P$ to find those with the $k$ smallest $d_{agg}(p,Q)$. This method is expensive, since the entire network has to be traversed $|Q|$ times. In this section, we propose three algorithms that solve ANN queries by effectively minimizing network traversal. The first one utilizes Euclidean lower bounds and an incremental Euclidean ANN method. The other two algorithms are motivated by aggregate top-$k$ query processing techniques.

## 4.1 Incremental Euclidean Restriction (IER)

The *Incremental Euclidean Restriction* (IER) paradigm was first applied for conventional nearest neighbor queries in road networks [12]. Assume that we have a single query point $q$ and we wish to find its NN in a set of points $P$, lying on a network. An R–tree that indexes $P$ is used to incrementally retrieve the Euclidean nearest neighbors of $q$ in $P$ using the algorithm of [4]. For each point $p$ retrieved, $d(p, q)$ is computed using a shortest path algorithm. If the Euclidean distance $d^{\mathcal{E}}(p, q)$ of the next point $p$ is larger than or equal to the network distance of the best NN found, IER terminates since (due to Lemma 1) no closer neighbor can be found. The IER ANN algorithm is based on this paradigm. Specifically, IER (i) incrementally retrieves points $p \in P$ based on their $d^{\mathcal{E}}_{agg}(p, Q)$, (ii) for each point $p$, it computes $d_{agg}(p, Q)$ and (iii) updates the result accordingly, until the aggregate Euclidean distance of the next point exceeds the network aggregate distance of the NN. In addition, we apply two optimizations for avoiding redundant shortest path computations.

### 4.1.1 The generic algorithm

IER requires an incremental algorithm for Euclidean ANN search. The existing Euclidean ANN algorithms [11], however, are not incremental and focus exclusively on the $d^{\mathcal{E}}_{sum}$ aggregate distance function. Thus, we first propose a generalized, incremental technique that works for any monotone aggregate function. Let $e$ be an entry of an R–tree that indexes $P$ and $e.B$ its *minimum bounding box*. Let $mindist(e, q_i)$ be the minimum Euclidean distance between $e.B$ and $q_i$, reflecting the minimum possible distance of any point $p \in P$, indexed under $e$, from $q_i$. Given a monotone aggregate function $agg$, $d^{\mathcal{E}}_{agg}(e, Q)$ can be defined by $agg\{mindist(e, q_i), \forall q_i \in Q\}$. Now, we can prove the following lemma:

**Lemma 3** *Let $Q$ be a set of query points and $e$ an R–tree node entry. For any point $p$ indexed under $e$, $d^{\mathcal{E}}_{agg}(e, Q) \leq d^{\mathcal{E}}_{agg}(p, Q)$ holds.*

**Proof.** True, due to $mindist(e, q_i) \leq d^{\mathcal{E}}(p, q_i)$ and the fact that $agg$ is monotone. ∎

Based on Lemma 3, we can solve ANN queries in the Euclidean space as follows. Initially, all entries of the R–tree root are added to a heap $H$ sorted on $d^{\mathcal{E}}_{agg}(e, Q)$. The top element $e$ of $H$ is dequeued. If it is a directory entry, then the corresponding R–tree node is accessed and its entries are added to $H$. If it is a point (i.e., leaf entry), it is returned as the next Euclidean ANN. Thus, this method allows us to incrementally retrieve points in ascending order of their aggregate Euclidean distance from $Q$.

Figure 6 presents the pseudocode for network ANNs by combining the above method with the multi-step processing framework [13]. IER retrieves Euclidean ANNs and computes their actual aggregate distance by shortest path queries (SPQs). The $k$-ANN set is updated and the process continues until the $k$-th network ANN found so far has distance smaller than or equal to the Euclidean distance of the next entry from $H$. For SPQs, we apply A* due to its superior performance compared to Dijkstra's algorithm. IER can output the network ANNs incrementally (i.e., without prior knowledge of $k$) by re-inserting in $H$ each point after computing its network aggregate distance. If the dequeued element from $H$ is a point $p$ whose network distance has already been computed, $p$ is reported as the next NN.

Algorithm **IER**($G$,$R$,$P$,$Q$)
1. *best_dist*:=$\infty$; $H$:=new priority queue;
2. **for each** entry $e$ in root(R) $enqueue(H, e, d^{\mathcal{E}}_{agg}(e, Q))$;
3. **while** ($notempty(H)$)
4. $e$:=$dequeue(H)$;
5. **if** ($d^{\mathcal{E}}_{agg}(e, Q) \geq best_dist$) **then** stop;
6. **if** ($e$ is an object in $P$) **then**
7. compute $d_{agg}(e, Q)$ using SPQs;
8. update best-$k$ results and *best_dist* if necessary;
9. **else** // $e$ is a R–tree node
10. **for each** entry $e'$ in R–tree node pointed by $e$
11. $enqueue(H, e', d^{\mathcal{E}}_{agg}(e', Q))$;

Figure 6: The IER ANN algorithm

For instance, assume that we want to find the 1-$ANN_{sum}(P, Q)$ in the graph of Figure 7a (let $Q = \{q_1, q_2\}$ and $P = \{p_1, p_2, p_3, p_4\}$) using IER. Figure 7b shows the two leaf nodes

of a 2-level R–tree that indexes $P$. Initially, the R–tree root entries are enqueued and $H = \{(M_2, 5), (M_1, 11)\}$. $M_2$ is then dequeued, its points are enqueued and $H = \{(p_1, 7), (p_3, 8), (M_1, 11)\}$. After $p_1$ is dequeued, IER applies SPQs to find its network distance and $p_1$ becomes the current ANN with $best_dist = 10$. Next, $p_3$ is dequeued and since $d^{\mathcal{E}}_{sum}(p_3, Q) < best_dist$, we compute its network distance $d_{sum}(p_3, Q) = 14 > best_dist$. Finally, $M_1$ is dequeued. Since $d^{\mathcal{E}}_{sum}(M_1, Q) = 11 \geq best_dist$, the algorithm terminates reporting $p_1$ as the ANN.

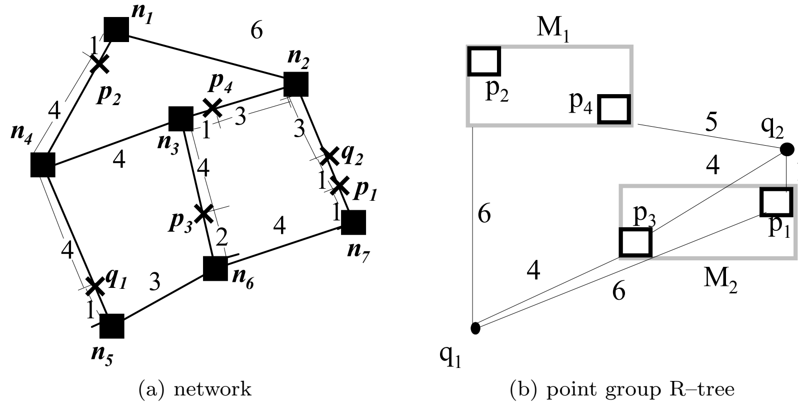



Figure 7: ANN search example

#### 4.1.2 Optimizations of IER

If multiple points appear on an edge $(n_x, n_y)$, IER has to apply SPQs for $n_x$ and $n_y$ several times. In addition, each time a point $p$ is popped from $H$, the point B$^+$–tree of Figure 3 must be accessed for finding the edge where $p$ lies. In order to minimize the shortest path computations and avoid visiting the same edge multiple times, we apply the following optimization. Whenever a Euclidean ANN $p$ is popped, $d_{agg}(p', Q)$ is computed for *all* points $p'$ in the same point group as $p$. The effectiveness of this optimization requires a modification of the R–tree structure because whenever we access a data point, we also need to retrieve all the other points lying on the same edge. Thus, we first create, for each edge populated by some $p \in P$, a minimum bounding box containing all the data points on the edge. Then, the R–tree is built on these bounding boxes (instead on the data points).

As a motivation for the second optimization, note that all SPQs have a common set of source nodes (i.e., the query points $Q$). Thus, for each query point $q_i$, we can re-use information about network nodes visited by previous SPQs that originated from $q_i$. In particular, the network nodes (and their distances) discovered by every SPQ are stored in a hash table $T_{q_i}$. In addition, for each $q_i$, we maintain the heap contents so that the network expansion can continue from its previous state. Assume a new SPQ with source $q_i$ and destination $n_x$. If $n_x$ is in $T_{q_i}$, we directly use the network distance stored in $T_{q_i}$. Otherwise, we use the previous state of the A$^*$ heap to resume search until $n_x$ has been reached (recording any newly visited nodes in $T_{q_i}$).

## 4.2 The threshold algorithm (TA)

Our second algorithm is based on the observation that the network traversal from each $q_i \in Q$ visits the nodes in increasing order of their distances from $q_i$. Thus, the *network node* with the minimum aggregate distance from $Q$ can be found by (i) concurrently and incrementally expanding the network around each $q_i \in Q$, (ii) applying some top-$k$ aggregate query processing technique [3] to guide and terminate search when the $k$ nodes with the minimum $d_{agg}(n, Q)$ are guaranteed to be found.

Note that there is a subtle difference between this problem and the ANN problem we study in this paper. The above method will derive the ANN only if $V = P$, i.e., the network nodes correspond to the points in $P$, but not in the general case, where points in $P$ lie on (arbitrary) edges. Consider the network of Figure 7a; even though $n_6$ minimizes $d_{sum}(P, Q)$, the ANN in $P$ is $p_1$, which is not close to $n_6$. Therefore, we need extensions of top-$k$ algorithms that consider the special nature of the problem.

The threshold algorithm (TA) takes its name from the corresponding technique used for top-$k$ queries [3]. Figure 8 shows the pseudocode of TA. For each query point $q_i$, TA first computes $d_{agg}(p, Q)$ for all points on the edge $(n_x, n_y)$ containing $q_i$. In addition, $n_x$ and $n_y$ are added to a heap $H$ that stores triplets $(n_x, d(q_i, n_x), q_i)$. $H$ keeps nodes $n_x$ visited by

some $q_i$ ordered on $d(q_i, n_x)$. Thus, its top element corresponds to the next nearest node from any $q_i$. TA iteratively pops nodes from $H$, computes $d_{agg}(p, Q)$ on their adjacent edges, and adds their adjacent nodes to $H$ (if they have not been visited from the same $q_i$ before). During the process, a set of $k$-ANNs retrieved so far is maintained.

Let $best_dist$ be the distance of the $k$-th ANN found. TA terminates when the next node popped from $H$ has distance larger than or equal to a threshold $\theta$. $\theta = \frac{best_dist}{|Q|}$ for $d_{sum}$ and $\theta = best_dist$ for $d_{max}$. If this condition is met, then no unexamined edge can contain better solutions, as guaranteed by the following lemma.

**Lemma 4** *Let $(n_x, n_y)$ be an edge that does not contain any point $q_i \in Q$. For any $\theta \geq 0$, if $\forall q_i \in Q, (d(n_x, q_i) \geq \theta \wedge d(n_y, q_i) \geq \theta)$ then $\forall p$ on $(n_x, n_y)$, $d_{sum}(p, Q) \geq |Q| \cdot \theta$ and $d_{max}(p, Q) \geq \theta$.*

**Proof.** Since no $q_i$ lies on $(n_x, n_y)$, the shortest path from each $q_i$ to any $p$ on the edge should be an extension of one of the shortest paths from $q_i$ to $n_x$ or from $q_i$ to $n_y$. Thus, for each $q_i$, $d(p, q_i) \geq \min\{d(n_x, q_i), d(n_y, q_i)\} \geq \theta$ and, as a result, $d_{sum}(p, Q) \geq |Q| \cdot \theta$, $d_{max}(p, Q) \geq \theta$. ∎

Consider the network of Figure 7a and assume that we want to find the 1-$ANN_{sum}(P, Q)$ using TA. First, the edges where $q_1$ and $q_2$ lie are examined. Since edge $(n_7, n_2)$ is populated, TA applies a SPQ (e.g., by using A*) from $q_1$ to find $d(q_1, n_2)$ and $d(q_1, n_7)$. The aggregate distance of $p_1$ (on $(n_7, n_2)$) can directly be computed and $p_1$ becomes the current ANN with $best_dist = 10$. The threshold is $\theta = 5$ ($best_dist/2$). Now the expansion heap is initialized to $H = \{(n_5, 1, q_1), (n_7, 2, q_2), (n_2, 3, q_2), (n_4, 4, q_1)\}$. Entries $(n_5, 1, q_1)$ and $(n_7, 2, q_2)$ are then popped in this order, not affecting the result since the (unexamined) edges adjacent to $n_5$ and $n_7$ are not populated. Thus, TA continues by popping $n_2$, examining edge $(n_2, n_3)$, and rejecting $p_4$ with $d_{sum}(p_4, Q) > 10$. Next, entries $(n_4, 4, q_1)$ and $(n_6, 4, q_1)$ are popped and $p_2$ and $p_3$ are also rejected after computing their aggregate distances. TA finally terminates when entry $(n_6, 6, q_2)$ is dequeued, since its distance is greater than $\theta$.

Algorithm **TA**($G$,$P$,$Q$)
1. $best_dist$:=$\infty$; $H$:=new priority queue;
2. **for each** query point $q_i \in Q$
3. let $q_i$ lie on the edge $(n_x, n_y)$;
4. **if** $(n_x, n_y)$ is populated **then**
5. compute $d_{agg}(p, Q)$ by SPQ for all $p$ on $(n_x, n_y)$;
6. update best-$k$ results (and $best_dist$) if necessary;
7. create new queue entries $B_x, B_y$;
8. $B_x.node$:=$n_x$; $B_x.qid$:=$q_i$; $B_x.dist$:=$d_L(q_i, n_x)$;
9. $B_y.node$:=$n_y$; $B_y.qid$:=$q_i$; $B_y.dist$:=$d_L(q_i, n_y)$;
10. $enqueue(H, B_x)$; $enqueue(H, B_y)$;
11. **while** $(notempty(H))$
12. $B$:=$dequeue(H)$;
13. **if** $B.dist \geq \theta$ **then** terminate;
14. **if** ($B.node$ not visited by $B.qid$ before) **then**
15. **for each** adjacent node $n_z$ of $B.node$
16. **if** $(B.node, n_z)$ is populated and not checked **then**
17. compute $d_{agg}(p, Q)$ by SPQ for all $p$ on $(B.node, n_z)$;
18. update best-$k$ results (and $best_dist$) if necessary;
19. create a new queue entry $B'$;
20. $B'.node$:=$n_z$; $B'.id$:=$B.id$;
21. $B'.dist$:=$B.dist + W(e(n_z, B.node))$;
22. $enqueue(H, B')$;

Figure 8: The threshold ANN algorithm

TA uses the optimizations of IER to reduce redundant shortest path computations and to avoid multiple SPQs from the same query points. Furthermore, in order to minimize accesses to the point file, TA first computes $lb(n_x, n_y)$, a lower bound for any possible $p$ on $(n_x, n_y)$, according to the methodology described in Section 3.2. If $lb(n_x, n_y) \geq best_dist$, the corresponding group of points does not need to be accessed.

## 4.3 Concurrent expansion (CE)

The *concurrent expansion* (CE) algorithm is similar to TA in that it concurrently and incrementally expands the network around each $q_i \in Q$. However, unlike TA, it does not

perform SPQs to compute *all* component distances $d(q_i, n_x)$ and $d(q_i, n_y)$ when a populated edge $(n_x, n_y)$ is visited, but waits until the edge has been seen from all $q_i$ during the concurrent expansion. Only then, CE can derive the aggregate distance for any points on $(n_x, n_y)$. Thus, CE avoids shortest path computations, which can be expensive, because they traverse network nodes in a less systematic way, incurring more random I/Os.

When a populated edge $(n_x, n_y)$ is visited by CE, some component distances $d(q_i, n_x)$ and $d(q_i, n_y)$ might not be known yet. However, such an edge may contain solutions. For instance, in Figure 7a, the ANN (point $p_1$) is near $q_2$, but it is far from $q_1$. Thus, CE maintains a set $S$ of populated edges, which have been visited and may contain data points with aggregate distance smaller than *best_dist*. Before CE can terminate, all edges in $S$ have to be visited from all $q_i \in Q$ in order to compute $d_{agg}(p, Q)$ for each point $p$ on them and verify whether $p$ is in the $k$-ANN set.

Figure 9 shows a pseudo-code for CE. Initially, the nodes that form the edges where each $q_i$ lies are pushed on a common heap $H$, labeled using the id of the corresponding query point (i.e., $q_i$), and organized based on their distance from it (lines 2–9). CE iteratively pops elements from $H$ while the heap is not empty and a termination condition (to be discussed shortly) is not met. After popping an entry $n_x$ that has not been visited before from the same query point, CE visits all neighbors $n_z$ of $n_x$ and enqueues them in $H$ (by adding $W(n_x, n_z)$ to $n_x$'s distance from its expansion source $q_i$). At the same time, it checks whether $n_x$ or $n_z$ have been visited from all query points $q_i$ (lines 20–22). In this case, we can derive an aggregate distance for all points in $P$ that lie on $(n_x, n_z)$.[2] Thus all $p \in P$ on $(n_x, n_z)$ (if any) are visited and potentially added on the best $k$ results found so far.

CE terminates if at some point we know that no better solution than the $k$-th best so far can be found. This can happen when (i) $S = \varnothing$ and (ii) the distance of the last popped node exceeds $\theta$ (the same threshold as TA). The first condition ensures that we have visited and eliminated all edges that may contain a better solution based on the (partial) component

[2]Note that these aggregate distances are just upper bounds, since edge $(n_x, n_z)$ can be later visited again (via another path) and the distances points on it can be improved.

Algorithm **CE**($G$,$P$,$Q$)

```
1. H:=new priority queue; best_dist:=∞; S:=∅;
2. for each query point q_i ∈ Q
3.   let q_i lie on the edge (n_x, n_y);
4.   if (edge (n_x, n_y) is populated) then
5.     add (n_x, n_y) to S;
6.   create new queue entries B_x, B_y;
7.   B_x.node:=n_x; B_x.qid:=q_i; B_x.dist:=d_L(q_i, n_x);
8.   B_y.node:=n_y; B_y.qid:=q_i; B_y.dist:=d_L(q_i, n_y);
9.   enqueue(H, B_x); enqueue(H, B_y);
10. while (notempty(H))
11.   B:=dequeue(H);
12.   if (S = ∅ ∧ B.dist ≥ θ) then terminate;
13.   if (B.node not visited by B.qid before) then
14.     for each adjacent node n_z of B.node
15.       if (n_z not visited by B.qid before) then
16.         create a new queue entry B';
17.         B'.node:=n_z; B'.qid:=B.qid;
18.         B'.dist:=B.dist + W(e(n_z, B.node));
19.         enqueue(Q, B');
20.       if (∀q_i ∈ Q,
21.           (B.node or n_z have been visited by q_i)
22.           ∨ (q_i on edge (B.node, n_z))) then
23.         for each p ∈ (B.node, n_z) compute d_agg(p, Q)
24.           update best-k results (and best_dist) if necessary;
25.       if (edge (B.node, n_z) is populated) then
26.         compute lb(B.node, n_z);
27.         if lb(B.node, n_z) < best_dist then
28.           add (B.node, n_z) to S;
29.         else if (B.node, n_z) in S;
30.           delete (B.node, n_z) from S;
```

Figure 9: The concurrent expansion ANN algorithm

distances available. When we visit an adjacent edge $(n_x, n_z)$ to the currently popped node $n_x$, we add it in $S$ if it is populated and $lb(n_x, n_z)$, the *lower* aggregate distance bound of any point on it, is smaller than $best_dist$ (lines 25–28). $lb(n_x, n_z)$ is computed by the methodology described in Section 3.2; for each $q_i$, if $n_x$ ($n_z$) has been visited by $q_i$ we use

the actual $d(q_i, n_x)$ ($d(q_i, n_z)$), else we use a lower bound for $d(q_i, n_x)$ ($d(q_i, n_z)$), which is equal to the last distance popped from $H$ (i.e., $B.dist$). If $(n_x, n_z)$ is populated and already in $S$, but now $lb(n_x, n_z) \geq best_dist$, we remove it from $S$; no point in $(n_x, n_z)$ can be in the $k$-ANN set (lines 29–30). Thus $S$ initially grows (when no $best_dist$ is available) and later shrinks as the ANN bound becomes tighter. For all unvisited populated edges, we know that their end-nodes are further than $\theta$ from all query points. Thus, due to Lemma 4 they may not contain any points better than the current solution. When the two conditions are met, CE terminates with the correct results.

Let us see how CE finds the 1-$ANN_{sum}(P, Q)$ in the graph of Figure 7a. Network nodes are concurrently visited from different query points in ascending order of their component distance to the nearest query point. Thus, $(n_5, 1, q_1)$, $(n_7, 2, q_2)$, $(n_2, 3, q_2)$ and $(n_4, 4, q_1)$ are dequeued from $H$ in this order; first $n_5$ will be visited (from $q_1$), then $n_7$ (from $q_2$), etc. $n_7$, $n_2$, and $n_4$ will add to $S$ populated edges $(n_2, n_7)$, $(n_2, n_3)$, and $(n_1, n_4)$ respectively, since currently $best_dist = \infty$. Next, $(n_6, 4, q_1)$ is dequeued and $(n_3, n_6)$ is added to $S$. Then, when $(n_6, 6, q_1)$ is dequeued and edge $(n_3, n_6)$ is checked, note that the condition of lines 20–22 is met; we can compute an upper distance bound for all points on $(n_3, n_6)$ (since $n_6$ has been reached from all query points). This gives us the first ANN $p_3$ with $d_{sum}(p_3, Q) \leq 14$. Note that 14 is just an upper bound for $d_{sum}(p_3, Q)$, since it is possible to visit $p_3$ via another path (i.e., via node $n_3$) and find a smaller value. Now, $best_dist$ is updated to 14 and $(n_3, n_6)$ is removed from $S$, since from the information so far $d(n_6, q_1) = 4$, $d(n_6, q_1) = 6$, $d(n_3, q_1) \geq 6$, and $d(n_3, q_2) \geq 6$, $d_{sum}(p_3, Q)$ cannot be improved. When $(n_3, n_6)$ is later visited (at dequeuing $(n_3, 7, q_2)$), it is added to $S$ for the same reason. CE continues this way, and when $(n_7, 8, q_1)$ is popped, the actual ANN $p_1$, with $d_{sum}(p_1, Q) = 10$ is found. Then, $(n_2, n_7)$ is removed from $S$, and, eventually, CE terminates when $S$ becomes empty.

The points of an edge are examined at most $|Q|$ times and a node can be enqueued at most $|Q|$ times. In addition, CE does not use Euclidean distance bounds, thus it can be applied in the general case, where there are no relationships between edge weights and Euclidean distances

between the corresponding nodes. Nevertheless CE can be adapted to use Euclidean bounds in the computation of $lb(B.node)$ (see line 26) by considering the $\max\{d^{\mathcal{E}}(B.node, q_i), B.dist\}$ for each $q_i$ where from $B.node$ has not been seen.

# 5 Variants of ANN queries

An interesting variant of the ANN query takes as input a set of query points $Q$ and finds the location $l$ on the network that minimizes the aggregate function, without requiring $l$ to be an object. For instance, suppose that the mobile users want to meet at the best location in the graph, without caring whether there is a particular facility there. We call such queries *aggregate center* (AC) queries. Euclidean AC queries can be solved efficiently using numerical methods. However, in a spatial network, it is not trivial to find the center of a group of query points. Aggregate center queries can directly be processed by the proposed algorithms. All edges are treated as populated. In addition, there are no accesses to any points, but the virtual points on the edges that minimize the aggregate function are computed as discussed in Section 3.2. IER in this case, employs an R–tree that indexes the edges of the network.

Concerning ANN queries, the $d_{sum}$ and $d_{max}$ functions have some interesting *weighted* variants. Assume, for instance, that every $q_i$ is the position of some vehicle carrying $w_i$ passengers and the goal is to find the facility $p$ that minimizes the total distance traveled by all passengers (as opposed to vehicles), i.e., $sum\{w_i d(q_i, p), \forall q_i \in Q\}$. Similarly, if each vehicle has average travel speed $v_i$, then the facility $p$ that leads to the earliest meeting time is the one that minimizes $max\{d(q_i, p)/v_i, \forall q_i \in Q\}$, i.e., $w_i = 1/v_i$. Our algorithms can be easily adapted for weighted queries. For IER, we can show that the weighted Euclidean component distance lower bounds the corresponding weighted network component distance. In addition, the Euclidean ANN algorithm used by IER is tuned to return incrementally weighted ANN. Finally, in TA and CE, the network nodes are visited in order of their weighted distance from any query point and the termination conditions and lower bounds are adjusted accordingly.

Finally, our algorithms can be used for complex ANN queries, carrying *selection constraints* or *preferences* on attributes of the interesting points other than their locations. For instance, consider three users who want to meet at the nearest restaurant which serves Mexican food. In this case, the interesting points (e.g., restaurants) carry some non-location information (e.g., the food they serve). During search, our algorithms can filter out from consideration those points that do not qualify the selection conditions (e.g., restaurants that do not serve Mexican food). As an example of another ANN query that carries non-location preferences, consider three users who want to meet at the nearest and cheapest restaurant. In this case, the aggregate function also contains non-location components (e.g., restaurant price) for the interesting points. For such queries, incremental versions of our ANN algorithms could be used in combination with incremental ranking algorithms for the non-location components to retrieve the combined top-$k$ results (e.g., using the rank-join operators of [10, 6]).

# 6 Experimental Evaluation

In this section, we evaluate the efficiency of the proposed IER, TA, and CE algorithms. We also developed three SPQ variants that use none, partial, or full materialization of network distances. The first is the A* algorithm. For the second SPQ variant, we implemented a 2-level $HiTi$ graph for each network as suggested in [9]. The number of subgraphs in $HiTi$ was tuned to $10^2$, after comparing versions of $5^2, 10^2, 15^2$, and $20^2$ subgraphs. For the third SPQ algorithm, we constructed a secondary memory array that materializes all $O(V^2)$ distances. Each access to a materialized network distance costs a disk page access.[3] The algorithms were developed in C++ and the experiments were executed on a PC with a Pentium 4 CPU of 2.3GHz. We used an LRU memory buffer of 1Mb and the page size was set to 4Kb.

[3]Network distances from the same source node can span across an average of $\frac{1}{2}|V|/1024$ pages (assuming that a float takes 4 bytes).

## 6.1 Experimental setup

Table 1 contains the real road networks of the evaluation. NA and CN were downloaded from `www.maproom.psu.edu/dcw/`. SF, TG, and OL were obtained from [1]. Since the original networks were not connected, we extracted the largest connected components from them. The coordinates of the nodes are normalized in the domain $[0, 10000]^2$. The weight of each edge was set to the Euclidean distance between the end-points, multiplied by a random number chosen from the range $[1, F]$. In this way, the Euclidean lower bound of Lemma 1 holds, whereas, $F$ reflects the factor by which the actual weight may deviate from the Euclidean distance.

| Id | Description | # nodes | # edges |
|---|---|---|---|
| NA | road segments in North America | 175,813 | 179,179 |
| CN | rail-roads of China | 32,925 | 33,120 |
| SF | San Francisco road map | 174,956 | 223,001 |
| TG | San Joaquin County road map | 18,263 | 23,874 |
| OL | Oldenburg road map | 6,105 | 7,035 |

Table 1: Real datasets used in the experiments

We uniformly generated points on the network edges. In order to control the density of the generated points, we set the distance between adjacent points to a parameter $G$. On the CN network, in specific, we used a real point dataset, which is a hypsography supplemental point dataset (51,663 points) obtained from the same source. The points in the query set $Q$ are generated randomly on edges of a random connected sub-network covering $A\%$ of the network edges.

Unless otherwise stated, each query has $|Q| = 8$ query points randomly generated in $A = 4\%$ of the network and the number of aggregate nearest neighbors $k$ is set to 10. The default values for the other parameters are $F = 1$ and $G = 0.1\overline{W}$ where $\overline{W}$ is the average edge weight. For each experimental setting, we averaged the results of the algorithms over 10 queries in order to reduce the randomness effect.

## 6.2 Performance study

Table 2 shows the performance of the algorithms on the SF network, using the default data and query generation parameters. Each row corresponds to an ANN algorithm, the SPQ variant used by it, and the aggregate distance function ($d_{sum}$ or $d_{max}$). We show the results of SPQ implementations that use partial and full materialization for IER. Materialization is expected to produce similar results for TA, whereas it is inapplicable for CE. Note that the full materialization approach is too slow in terms of I/O and response time, since each network distance computation incurs a random access. Although the *HiTi* implementation incurs few page accesses, its execution cost is high because *HiTi* search cannot be used incrementally, as opposed to A*. The reason is that *HiTi* computes each time the shortest path distance, without going through the intermediate nodes. Thus, any distance for these nodes (if required later) has to be computed from scratch at a non-negligible computational cost. On the other hand, the optimized version of A*, discussed in Section 4.1, caches the distances of intermediate nodes in path computations, and uses the heap of the previous search incrementally to compute the next shortest path(s) efficiently. Since the full materialized approach and the *HiTi* search have high execution cost, we omit them from the remainder of the evaluation and consider A* as a standard implementation for SPQs.

Observe that CE performs better for $d_{max}$ than for $d_{sum}$. This is attributed to the fact that the lower bound used for pruning edges is much tighter for $d_{max}$ compared to $d_{sum}$. For $d_{max}$, in order for a visited edge to be inserted or maintained in $S$, it should be closer than *best_dist* from *all* query points. On the other hand, TA performs better for $d_{sum}$ than for $d_{max}$ because the termination condition holds earlier for $d_{sum}$ than for $d_{max}$. IER is fast and has fewer page accesses than the other methods. Also, its performance is stable over different aggregate functions. The last two rows of Table 2 show the performance of IER for aggregate center queries (described in Section 5). Note that AC queries have similar performance to ANN queries in this setting, since most edges are populated.

Table 3 shows the effect of different networks on the number of page accesses by the algo-

| Method | I/O | time (sec) | network node accesses |
|---|---|---|---|
| $d_{sum}$ | | | |
| CE | 1286 | 12.06 | 35995 |
| TA-A* | 569 | 1.38 | 5000 |
| IER-A* | 530 | 1.66 | 6343 |
| IER-HiTi | 543 | 189.51 | 1.1e6 |
| IER-Full | 70320 | 140.64 | 4395 |
| $d_{max}$ | | | |
| CE | 707 | 2.22 | 8380 |
| TA-A* | 736 | 2.68 | 9568 |
| IER-A* | 515 | 1.41 | 5575 |
| IER-HiTi | 479 | 44.25 | 2.9e5 |
| IER-Full | 35808 | 71.62 | 2238 |
| ANN center queries | | | |
| IER-A*, $d_{sum}$ | 510 | 1.65 | 6338 |
| IER-A*, $d_{max}$ | 507 | 1.38 | 5557 |

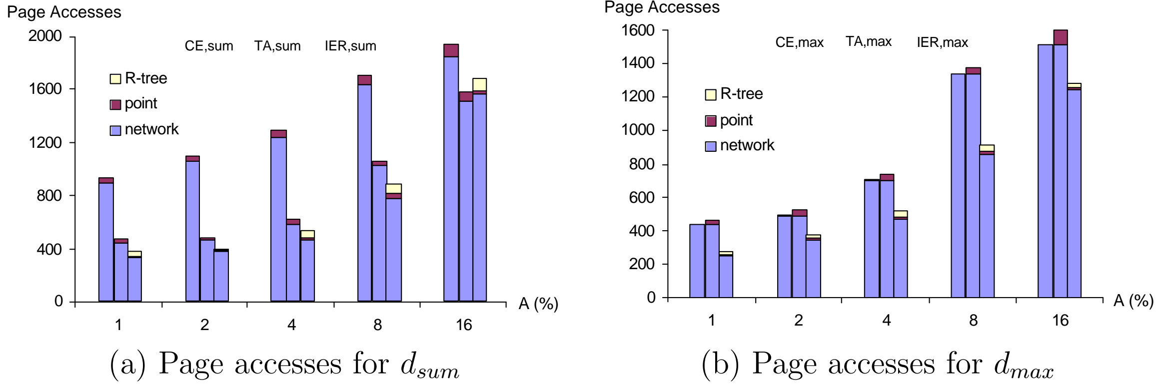
Table 2: Comparisons of different algorithms

| Method | CE,sum | TA,sum | IER,sum | CE,max | TA,max | IER,max |
|---|---|---|---|---|---|---|
| $NA$ | 421 | 210 | 367 | 262 | 290 | 283 |
| $SF$ | 1286 | 569 | 530 | 707 | 736 | 515 |
| $TG$ | 213 | 115 | 99 | 139 | 153 | 119 |
| $OL$ | 64 | 28 | 41 | 34 | 43 | 34 |
| $CN$ | 135 | 67 | 130 | 114 | 114 | 147 |

Table 3: Page accesses on different networks

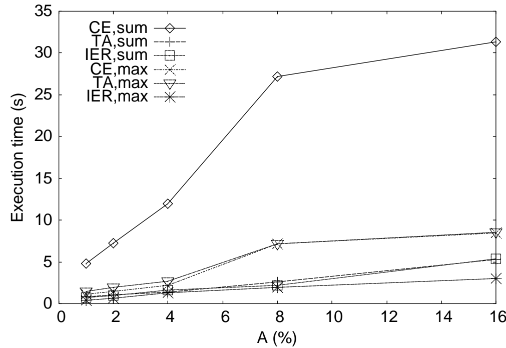
rithms. The cost of IER is linear to the size of the network (i.e., the number of edges). On the other hand, the costs of TA and CE increase superlinearly with the network density. For instance, CE's cost on SF is nearly triple compared to its cost on NA, even though the two networks have similar number of nodes. Since TA and CE traverse the network around the query points exhaustively, in dense networks, the same edges and nodes are visited from multiple paths, greatly increasing the complexity. On the other hand, IER is based mainly on shortest path computations, which are not very sensitive to the network density. As in the previous experiment, CE performs better for $d_{max}$ than $d_{sum}$, TA performs better for $d_{sum}$ than $d_{max}$, and IER has stable and best overall performance for different aggregate functions.

In the next experiment, we compare the three algorithms on the SF network, as a function




(a) Page accesses for $d_{sum}$

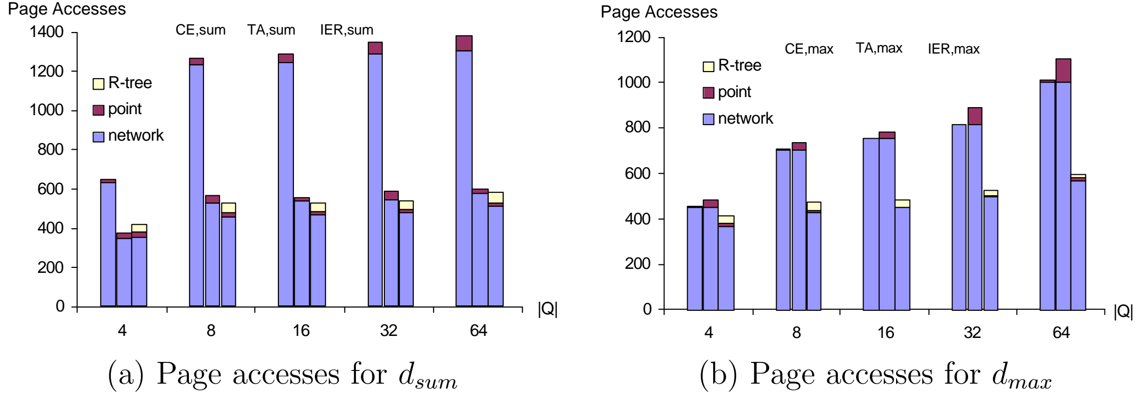
(b) Page accesses for $d_{max}$




(c) Execution time

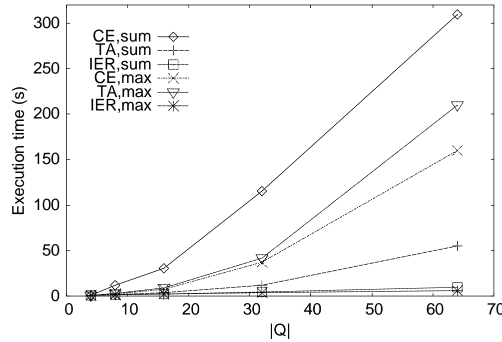
Figure 10: Cost as a function of query area $A$

of $A$, i.e., the sub-network area where the query points lie. Figure 10 shows the performance in terms of page accesses and response time. The I/O cost is decomposed to accesses on the network adjacency list file, on the point file, and on the R–tree. As expected, the cost increases with $A$, since the query points span a wider range of the network. For $d_{sum}$ queries, TA and IER have similar performance, with IER being marginally better in most cases. On the other hand, CE has consistently worse performance than the other methods. For $d_{max}$ queries, IER consistently outperforms CE and TA, but the difference is not large. CE is marginally faster than TA in this case. The execution times, in general, agree with the I/O figures, with the exception of CE for $d_{sum}$ queries, which is much slower than the other methods due to the large part of the network it has to explore from all query points. The difference is not as high in terms of I/O due to buffering effects.




(a) Page accesses for $d_{sum}$

(b) Page accesses for $d_{max}$

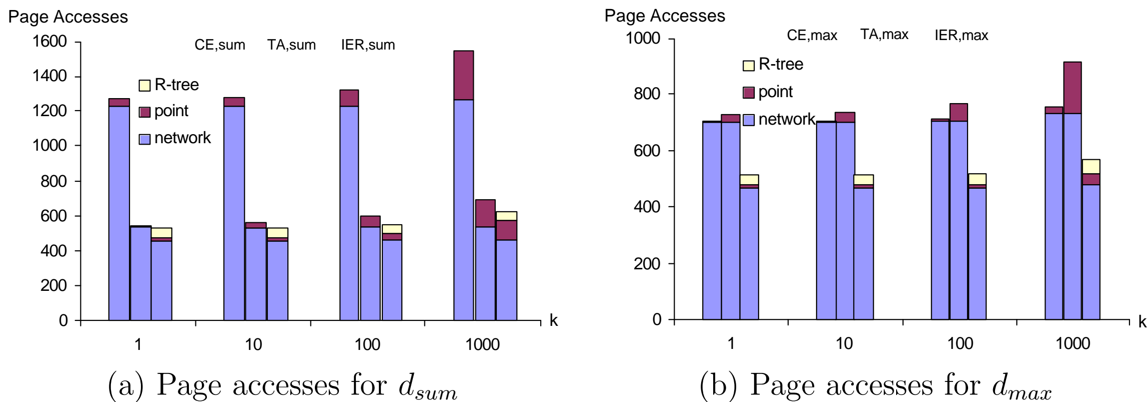



(c) Execution time

Figure 11: Cost as a function of the number of query points $|Q|$

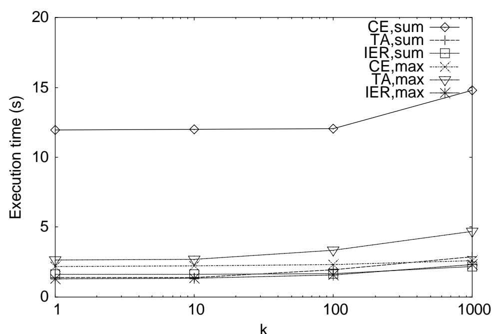
Figure 11 shows the effect of the number $|Q|$ of query points on the performance of the algorithms. Observe that the number of page accesses converges as $|Q|$ increases because the query points are distributed in the same query area. Similar to the previous experiments, CE does not perform well for $d_{sum}$ queries, whereas it outperforms TA for $d_{max}$ queries. The cost of CE increases fast with $|Q|$, for $d_{sum}$ queries, because the same edges and their point groups are checked multiple times (at most $|Q|$) and the cost of each check is directly proportional to $|Q|$. For $d_{max}$ queries the effect is smoother, due to the stricter pruning condition. TA's execution cost is also high due to the larger number of SPQ queries it has to perform. On the other hand, the cost of IER is affected less by $|Q|$ because, with the help of the tree, it can discover fast a good *best_dist* which prunes the search space effectively.

In Figure 12, we compare the algorithms by varying the number $k$ of ANN to be retrieved.




(a) Page accesses for $d_{sum}$

(b) Page accesses for $d_{max}$



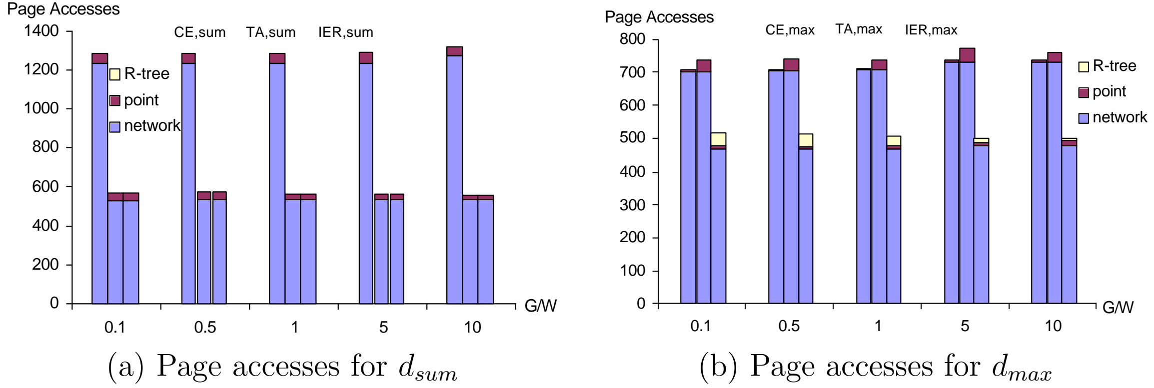

(c) Execution time

Figure 12: Cost as a function of $k$

Observe that the number of network and R–tree I/Os is insensitive to this parameter. On the other hand, the accesses on the point file increase linearly with $k$. Figure 12c shows the effect on the execution time. Since the dominating cost is the network access, the performance scales well with $k$ and the execution time increases slowly as $k$ increases.

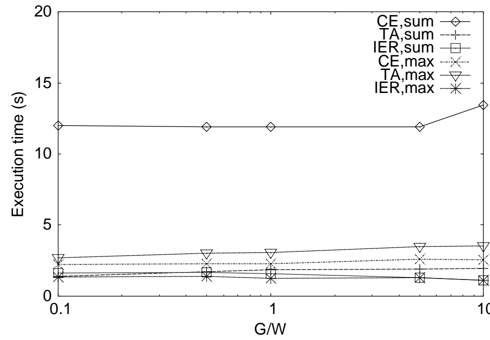
The next comparison factor is the density $G/\overline{W}$ of the data points on the network (Figures 13). Note that the density of the points is high when $G/\overline{W}$ is small and vice versa. As $G/\overline{W}$ increases, The number of network pages accessed increases slowly, since more edges become empty. On the other hand, the accesses to the point file and the R–tree (by IER) decrease, since $P$ becomes smaller. In general, the performance of all algorithms is insensitive to this parameter.

In the next experiment, we test the effect of points distribution in the network. We generated




(a) Page accesses for $d_{sum}$

(b) Page accesses for $d_{max}$




(c) Execution time

Figure 13: Cost as a function of the ratio $G/\overline{W}$

four clusters of 100K points each, according to the methodology of [19]. For each cluster, a random point is generated as its first point. Then, the network is traversed from this point. Whenever an edge is met for the first time, points are generated on it. The approximate distance between two consecutive generated points is initially $G$ and increases as the network is expanded to reach $G \times spread$ for the final point. Large values of *spread* generate more uniform distributions, whereas small values generate more skewed data. Figure 14 shows the performance of the algorithms as a function of the *spread* parameter. As expected, for skewed distributions, the algorithms are slower, since they have to explore larger parts of the network until they find the solutions.

We also compared the algorithms for *weighted* ANN queries, described in Section 5. Figure 15 shows the performance of the algorithms as a function of the skew in the distribution of

weights to query points. Note that only the performance of CE for $d_{sum}$ queries is affected by the skew on the weights. The more skewed the weights are, the larger part of the network CE has to explore from a single source. All visited populated edges are then added on $S$, which cannot be removed until visited by all other points.

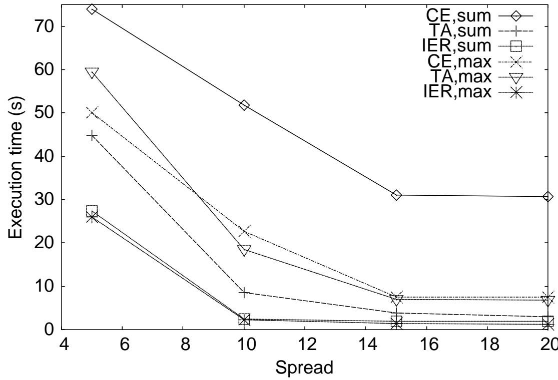



Figure 14: Effect of points skew

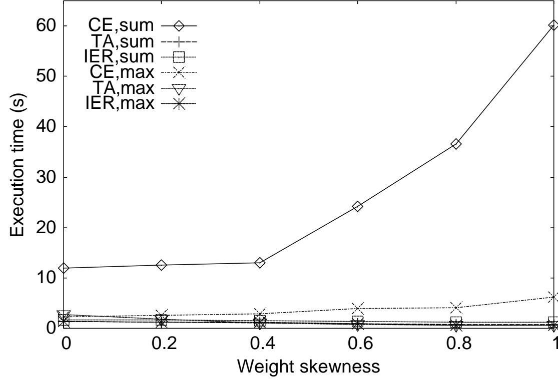
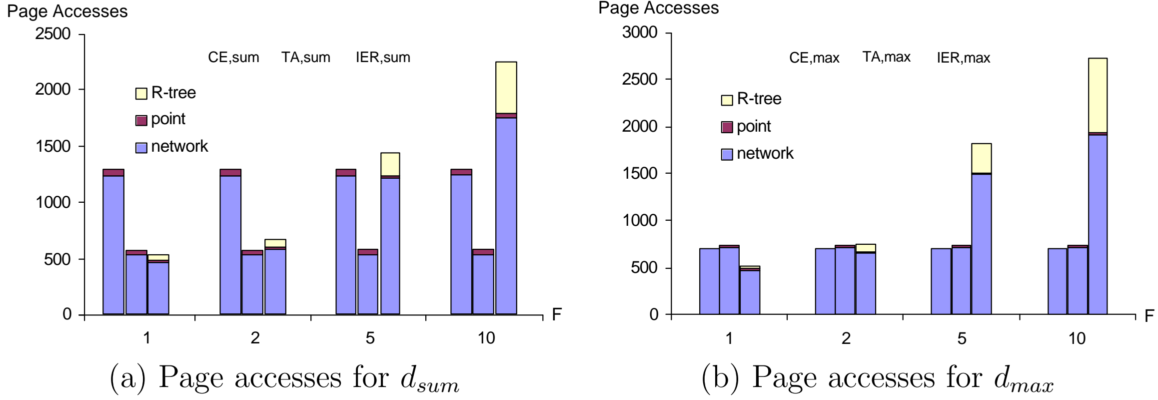



Figure 15: Weighted ANN

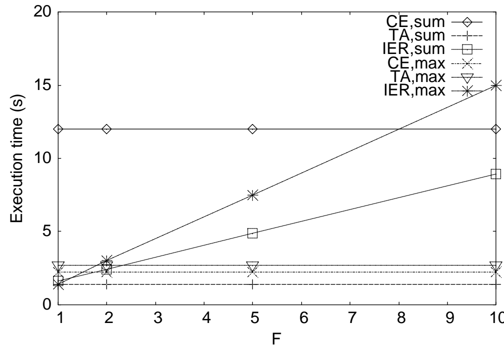
So far, IER dominates over TA and CE in almost all tested cases. Nonetheless, IER is not the best method when the weights of the edges are not proportional to their lengths as the next experiment suggests. We compared the algorithms after distorting the edge weights by different factors $F$ (Figure 16). Observe that TA and CE are not affected by this parameter. On the other hand, the performance of IER degrades with $F$, especially for $d_{max}$ queries, since the algorithm is based on the effectiveness of Euclidean distance as a lower bound of the network distance. As the edge weights become less proportional to the Euclidean distance, the Euclidean ANN distances become looser as a bound and IER explores a large number of edges and points before the result is guarranteed to be found. Thus, for large values of $F$ TA becomes the best method for $d_{sum}$ queries and CE dominates for $d_{max}$ queries.

# 7 Conclusion

In this paper, we have studied the interesting problem of aggregate nearest neighbor queries in road networks. Processing ANN queries in road networks cannot be achieved by straight-




(a) Page accesses for $d_{sum}$ (b) Page accesses for $d_{max}$




(c) Execution time

Figure 16: Effect of $F$

forward applications of previous approaches for the Euclidean space [11] due to the complexity of shortest path computations as opposed to geometric distances. We presented three algorithms that consider this inherent difficulty of the problem. IER incrementally retrieves Euclidean aggregate nearest neighbors and computes their network distance by shortest path queries, until the result cannot be improved. TA and CE explore the network around the query points until the aggregate nearest neighbors are discovered. Our techniques can be applied for various aggregate distance functions (sum and max). In addition, they can be combined with spatial access methods and shortest path materialization techniques. A thorough experimental study suggests

that their relative performance depends on the problem characteristics. IER is the best algorithm when the edge weights are proportional to their lengths, since in that case Euclidean

distance becomes a quite tight lower bound of the actual network distance. Nevertheless the performance of IER degrades fast as the weights are less reflected by the edge lengths. For such cases, TA is the most appropriate method for sum queries, whereas CE is the best approach for max queries. In addition, TA and CE are the only choices when the interesting points are not indexed by R-trees, or when the Euclidean distance bounds may not be used (e.g., in non-spatial networks). In the future, we plan to study the applicability of our techniques for problems where the set of query points is very large (i.e., it does not fit in memory), considering appropriate memory management techniques.

# References


[1] T. Brinkhoff. A framework for generating network-based moving objects. *GeoInformatica*, 6(2):153–180, 2002.

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

[3] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, 2001.

[4] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *TODS*, 24(2):265–318, 1999.

[5] Y.-W. Huang, N. Jing, and E. A. Rundensteiner. Integrated query processing strategies for spatial path queries. In *ICDE*, 1997.

[6] I. F. Ilyas, W. Aref, and A. Elmagarmid. Supporting top-k join queries in relational databases. In *VLDB*, 2003.

[7] C. S. Jensen, J. Kolar, T. B. Pedersen, and I. Timko. Nearest neighbor queries in road networks. In *ACM GIS*, 2003.

[8] N. Jing, Y. W. Huang, and E. A. Rundensteiner. Hierarchical encoded path views for path query processing: An optimal model and its performance evaluation. *TKDE*, 10(3):409–432, 1998.

[9] S. Jung and S. Pramanik. An efficient path computation model for hierarchically structured topographical road maps. *TKDE*, 14(5):1029–1046, 2002.

[10] A. Natsev, Y.-C. Chang, J. R. Smith, C.-S. Li, and J. S. Vitter. Supporting incremental join queries on ranked inputs. In *VLDB*, 2001.

[11] D. Papadias, Q. Shen, Y. Tao, and K. Mouratidis. Group nearest neighbor queries. In *ICDE*, 2004.

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

[13] T. Seidl and H.-P. Kriegel. Optimal multi-step k-nearest neighbor search. In *ACM SIGMOD*, 1998.

[14] C. Shahabi, M. R. Kolahdouzan, and M. Sharifzadeh. A road network embedding technique for k-nearest neighbor search in moving object databases. In *ACM GIS*, 2002.

[15] S. Shekhar, A. Kohli, and M. Coyle. Path computation algorithms for Advanced Traveller Information System (ATIS). In *ICDE*, 1993.

[16] S. Shekhar and D. Liu. CCAM: A connectivity-clustered access method for networks and network computations. *TKDE*, 19(1):102–119, 1997.

[17] S. Shekhar and J. S. Yoo. Processing in-route nearest neighbor queries: a comparison of alternative approaches. In *ACM GIS*, 2003.

[18] S. H. Woo and S. B. Yang. An improved network clustering method for I/O-efficient query processing. In *ACM GIS*, 2000.

[19] M. L. Yiu and N. Mamoulis. Clustering objects on a spatial network. In *ACM SIGMOD*, 2004.