Flexible Aggregate Nearest Neighbor Queries in Road Networks

Zhongpu Chen*, Bin Yao*, Xiaofeng Gao*, Shuo Shang†, Shuai Ma‡, Minyi Guo*
*Department of Computer Science and Engineering, Shanghai Jiao Tong University
800 Dongchuan Road, Shanghai, China
{chenzhongpu@, yaobin@cs., gao-xf@cs., guo-my@cs.}sjtu.edu.cn
†Computer Science Program, King Abdullah University of Science and Technology
Thuwal, Saudi Arabia
shuo.shang@kaust.edu.sa
‡School of Computer Science and Engineering, Beihang University
37 Xueyuan Road, Beijing, China
mashui@buaa.edu.cn

***Abstract*—Aggregate nearest neighbor (ANN) query has been studied in both the Euclidean space and road networks. The flexible aggregate nearest neighbor (FANN) problem further generalizes ANN by introducing an extra flexibility. Given a set of data points P, a set of query points Q, and a user-defined flexibility parameter ϕ that ranges in $(0, 1]$, an FANN query returns the best candidate from P, which minimizes the aggregate (usually *max* or *sum*) distance to any $\phi\,|Q|$ objects in Q. In this paper, we focus on the problem in road networks, and present a series of universal (i.e., suitable for both max and sum) algorithms to answer FANN queries, including a Dijkstra-based algorithm enumerating P, an approach that is made up of a list of queues and processes data points from-near-to-far, and a framework that combines *Incremental Euclidean Restriction* (IER) and kNN. We also propose a specific exact solution for *max*-FANN and a specific approximate solution for *sum*-FANN which can return a near-optimal result with a guaranteed constant-factor approximation. These specific algorithms are easy to implement and can achieve excellent performance in some scenarios. Besides, we further extend the FANN to k-FANN, and successfully adapt most of the proposed algorithms to answer k-FANN queries. We conduct a comprehensive experimental evaluation for the proposed algorithms on real road networks to demonstrate their superior efficiency and high quality.**

I. Introduction

The aggregate nearest neighbor (ANN) query [1]–[7] is a classic problem that has a large number of applications (e.g., location-based services) in spatial databases. Given a group of query points Q, ANN finds out a point in a set of data points P, which has the smallest aggregate distance to *all* points in Q. The aggregate function is usually either *max* or *sum*. The ANN problem has been studied in both the Euclidean space [1]–[3] and road networks [4]–[7].

In many cases, it is more desirable to take a fraction of query points Q into account. Consider the example of Fig. 1, where a set of data points $P = \{p_1, p_2, \dots, p_8, p_9\}$ (colored in black), and a set of query points $Q = \{q_1, q_2, q_3, q_4\}$ (colored in red). Note that p_4 and p_3, p_5 and p_4 are located at the same node, respectively. Some points are located at edges. For example, q_1 lies on (p_2, p_3), and q_2 lies on (p_3, p_6). Suppose Q is a set of small grain depots, and each depot can store 1 ton of grain, and P is a set of candidate locations to build a distribution center. If the designer of this distribution center wishes to collect *all* the grain from these depots while minimizing the aggregate traveling cost, this can be answered by an ANN query. The result of this *max*-ANN query is p_2 with the aggregate distance of 16 and the result of this *sum*-ANN query is also p_2 with the aggregate distance of 52. We can intuitively understand the result because p_2 is the geographical "center" of Q. However, if he or she wishes to collect only 2 tons of grain from some of these depots while minimizing the aggregate traveling cost, the result is different. In this case, we need to find the best place to build the distribution center that collects grain from only 50% of these depots. More precisely, a more general query is to allow users to specify a flexibility $\phi \in (0, 1]$, and the goal is to retrieve the best point from P that is the closest to *any* $\phi\,|Q|$ points in Q. We dub it as the flexible aggregate nearest neighbor (FANN) query. When ϕ is 50%, it is obvious that the result of this *max*-FANN query is p_3 with the aggregate distance of 2 and the result of this *sum*-FANN query is also p_3 with the aggregate distance of 4.

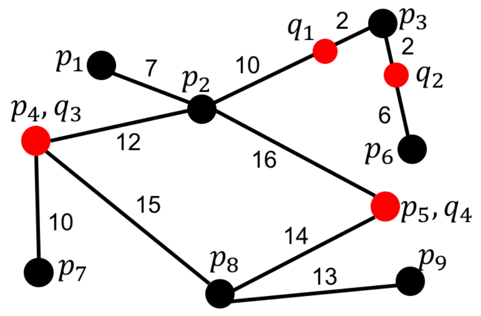


Fig. 1. Example of an FANN query in road networks

FANN queries [8], [9] are first studied in the Euclidean space. Compared with the Euclidean space, many operations in road networks are much more expensive. For example, determining the distance between any two points in the

Euclidean space can be solved in $O(1)$ time. However, the calculation process in road networks is much more complex, and the computation cost depends on the implementation of the shortest path distance algorithm. In order to further optimize the FANN algorithms in road networks, the topology nature of the road networks should be fully utilized in order to prune the unnecessary candidate points as many as possible. On the other hand, many geometrical properties in the Euclidean space will not be hold in road networks, and hence it is necessary to redesign the algorithms in the context of road networks.

To our best knowledge, there is no any research on FANN query problem in road networks. The most relevant study is the ANN query in road networks [4], [6], [7], but our study of FANN is not a trivial extension or adaption based on ANN query. The *IER* algorithm in [4] can reach the best performance, which uses R-tree to index the data objects, but it is not efficient when only partial points in Q are considered. Due to the newly introduced flexibility, the result of an FANN query is much more difficult to find. Because *any* $\phi\,|Q|$ objects in Q may be the potential targets, the number of all possible answers can achieve to the scale of $\binom{|Q|}{\phi|Q|}$. The algorithms in [6], [7] use a Voronoi diagram to partition the road network, but they can often cause unbalanced partitioning and thus are also inefficient in large road networks. Obviously, we can regard ANN query as a special case of FANN query when $\phi = 1$. Another relevant but different type of query is the *optimal meeting point* (OMP) query [5] in road networks, but the set P in OMP query is not determined in advance. However, the uncertainty of set P does not really increase the complexity. As proved in [5], [10], given an OMP query with a set of query points Q on a road network $G = (V, E)$, $V \cup Q$ contains an OMP. In other words, we can determine the set P in an implicit way. Hence, we can also regard the OMP query as a special case of the FANN query.

In this paper, we first focus on the universal methods for the FANN query problem in road networks, which means that they are applicable to both *max* and *sum*. We first design a Dijkstra-based algorithm enumerating P as our baseline. Secondly, we successfully modify the *List* algorithm [8], [9] in the context of road networks, which processes data points from-near-to-far. Thirdly, we design a general algorithm framework by combining *Incremental Euclidean Restriction* (IER) and kNN. Note that the IER here refers to general *Incremental Euclidean Restriction*, rather than the specific algorithm for ANN in [4]. To boost the efficiency, these general algorithms often require complex index structures which usually are infeasible for large dynamic road networks with respect to time and space. Motivated by this, we sacrifice some generalities and investigate a specific algorithm for *max*-FANN which can give an exact answer. Sometimes, it is also desirable to retrieve a near-optimal answer as fast as possible. Therefore, we further design an efficient specific 3-approximation algorithm for *sum*-FANN. We also prove that the approximation ratio can even reach to 2 when Q is a subset of P. These two specific methods have little dependence on index structure over road networks, and hence they can still give acceptable answers especially when road networks change frequently (or we cannot build an index over the whole road network easily). Besides, these specific algorithms are easy to implement. Finally, we further extend FANN to k-FANN, and adapt most of the proposed algorithms successfully to answer k-FANN queries in road networks.

The main contributions of our work can be summarized as follows:

- We firstly introduce an extra flexibility parameter to the classic ANN problem in road networks, and propose a series of methods to answer FANN queries in road networks.
- We design a Dijkstra-based algorithm to answer FANN queries, which is much better than adopting ANN as an independent module directly. Some researchers [8], [9] have tried the *List* to answer FANN queries in the Euclidean space. We modify this algorithm, by taking the features of road networks into account to solve FANN queries in road networks, and denote it as *R-List* (Road networks' *List*). We also combine IER with kNN, and then develop several algorithms based on the general algorithm framework for FANN queries.
- We propose a specific *Exact-max* (exact *max*-FANN) algorithm for *max*-FANN and *APX-sum* (approximate *sum*-FANN) algorithm for *sum*-FANN to get the exact and 3-approximation answer, respectively. We also prove that the approximation ratio of *APX-sum* can even reach to 2 if Q is the subset of P. These specific algorithms are easy to implement and can achieve excellent performance in some scenarios.
- We further extend FANN to k-FANN, and formulate it formally in road networks. We also successfully adapt most of the proposed algorithms to answer k-FANN queries.

The rest of this paper is organized as follows: Section II defines the problem, discusses some related works, and shows an outline of the proposed approaches. Section III presents universal methods, including a Dijkstra-based algorithm, a modified *List* and a general algorithm framework. Section IV presents specific methods to answer *sum*-FANN and *max*-FANN queries respectively. Section V discusses k-FANN, which is an extension of FANN. We present the experimental results in Section VI and make a conclusion in Section VII.

II. PRELIMINARIES

We first present the road network definitions that we follow throughout the paper and formulate the FANN problem formally. Then, we review some related works. Finally, we overview the outline of the proposed methods in this paper.

A. Problem Formulation

A road network can be represented as an undirected weighted graph, $G = (V, E, W)$, where V is the set of nodes, E is the set of edges, and W is a weight function $E \rightarrow \mathbb{R}^+$, which maps an edge to a positive real number.

We use Q to denote the set of query nodes, and use P to denote the set of data nodes, and use ϕ to denote the flexibility parameter, and ϕ varies in the range of $(0, 1]$. For simplicity, we assume that query (or data) objects are at vertices, i.e., $P \subset V$, $Q \subset V$. If the query (or data) object is on an edge, we can use the two vertices on the edge to do FANN search and merge the answer sets of the two vertices to generate the final result. If the query (or data) object is outside the whole network, we can find the closest point in the network and use it to do FANN search. The similar assumption is also adopted in [11]. In the following paper, "node" ,"point" , "object", and "vertex" are interchangeably used if the context is clear.

Let δ be the distance function on G, and the network distance $\delta(v_i, v_j)$ between node v_i and v_j is defined as the minimum sum of weights of any path between them. We use $\delta^\epsilon(v_i, v_j)$ to denote the Euclidean distance between node v_i and v_j. g is an aggregate function, which can be defined on a single node p and a dataset $S \subset V$, and it is either *sum* or *max* in this paper:

$$g(p, S) = g\{\delta(p, v_1), \delta(p, v_2), ..., \delta(p, v_k)\}$$

where $k = |S|$, $v_i \in S$, for $i = 1, 2, \ldots, k$.

Then we can define the *flexible aggregate function* g_ϕ, which is the most critical operation for the FANN query problem in road networks.

Definition 1 (Flexible Aggregate Function)**.** The flexible aggregate function g_ϕ, is a function that takes a node $p \in P$ and the set Q as its input, and returns a pair (Q^p_ϕ, d^p) as the result, which satisfies:

$$\begin{cases} Q^p_\phi = \underset{Q_\phi \subset Q, |Q_\phi| = \phi|Q|}{\operatorname{argmin}} g(p, Q_\phi), \\ d^p = g(p, Q^p_\phi), \end{cases}$$

where Q_ϕ is the subset of Q with $|Q_\phi| = \phi|Q|$. Given a p, we denote the *optimal flexible subset* of Q as Q^p_ϕ, and denote its *flexible aggregate distance* to Q as d^p.

It is worth noting that we usually use $g_\phi(p, Q)$ to denote the *flexible aggregate distance* for simplicity in the following paper. Our goal is to retrieve a point p^* in P to minimize d^p. An FANN query can be formalized with the following definition:

Definition 2 (FANN query)**.** The input of an FANN query is a quintuple (G, P, Q, ϕ, g), which returns a triple (p^*, Q^*_ϕ, d^*) as its answer such that:

$$\begin{cases} (p^*, Q^*_\phi) = \underset{p \in P, Q_\phi \subset Q, |Q_\phi| = \phi|Q|}{\operatorname{argmin}} g(p, Q_\phi), \\ d^* = g(p^*, Q^*_\phi), \end{cases}$$

where p^* is the point in P that minimizes the flexible aggregate distance, Q^*_ϕ is the optimal flexible subset, and d^* is the flexible aggregate distance.

At the end of this section, we summarize all those aforementioned symbols in Table I.

TABLE I
FANN SYMBOL LIST

Symbol	Description
ϕ	flexibility parameter
$\delta(\cdot, \cdot)$	distance function of G
$\delta^\epsilon(\cdot, \cdot)$	Euclidean distance function
$g(\cdot, \cdot)$	aggregate function, may be *max* or *sum*
$g_\phi(\cdot, \cdot)$	flexible aggregate function
Q^p_ϕ	flexible optimal subset of p
d^p	flexible aggregate distance of p
(G, P, Q, ϕ, g)	input of an FANN query
(p^*, Q^*_ϕ, d^*)	answer of an FANN query

B. Related Works

The shortest path algorithm. The shortest path algorithm is one of the most fundamental operations in road networks, and it has been extensively studied during the past half century. The *Dijkstra* algorithm [12] and its variants (e.g., A^* algorithm [13]) have been widely applied in location based services. We can use either lower bounds (or other heuristic properties) or materialization techniques to accelerate the shortest path computation. The fully materialization of distances requires high storage cost, while *HiTi* [14] and *HEPV* [15] materialize distances partially to make it feasible for large graph. Currently, PHL [16] is the fastest method by decomposing a graph into the shortest paths and storing distances from each vertex to the shortest path in its labels. Note that many indexing techniques also reply on distance materializations, and we will discuss it in the following.

Indexing techniques and hierarchical structure. Indexing techniques and hierarchical structure [14], [15], [17]–[19] are also widely used in road networks. The basic idea is to partition the graph into subgraph recursively, and pre-compute some shortcuts within subgraph. *CH* [18] has a low memory overhead, but it has to traverse a large number of nodes when objects are relatively dispersed in the graph. The authors in [7], [20] transplanted Voronoi digram to the domain of road networks, and presented some effective algorithms. It is usually required to keep the hierarchical structure balanced for better performance. K. C. Lee et al. [19] used *ROAD* to index road networks in a hierarchical way, but it performs badly if the objects are sparse and road networks are large. G-tree [11], [21] has a superior performance while the cost of building index is acceptable.

kNN, ANN, and FANN in the Euclidean space. The k-nearest neighbor (kNN) query has been studied for decades. Many successful approaches have been developed to solve this problem [11], [22], [23]. With some pruning strategies and search techniques, the kNN queries can be answered efficiently. Abeywickrama et al. [24] studied the different in-memory algorithms for kNN queries, and they proved that IER has an excellent potential. The ANN problem has been studied in both the Euclidean space [1]–[3] and road networks [4]–[7], and successfully adapted to the top-k algorithms. The *IER* algorithm in [4] can reach the best performance, which uses R-tree to index the data objects, but it is not efficient when only partial points in Q are considered. The algorithms in [6], [7] use a Voronoi diagram to partition the road network, but they can often cause unbalanced partitioning and thus are also inefficient in large road networks. Y. Li et al. [8], [9] first introduced the flexibility parameter into the ANN query definition in the Euclidean space, which generalized the classical ANN problem and offered it richer semantics. They proposed a series of exact and approximation algorithms to address this problem, but those algorithms cannot used directly due to the complex topology in road networks.

C. Outlines of Proposed Approaches

In this paper, we design two kinds of algorithms which are denoted as the universal and specific methods respectively. The universal methods are able to deal with *any* g (i.e., suitable for both max and sum), and the specific methods can only solve the problem when g is either max or sum.

As for the universal methods, a naive way is regard the ANN an independent module. To be specific, we enumerate $\binom{|Q|}{\phi|Q|}$ options to determine Q^p_ϕ (as the query objects) first, and then apply the ANN routine (e.g., *IER* in [4]) directly. However, this method is always infeasible in practice since $\binom{|Q|}{\phi|Q|}$ is often too large to deal with. For example, the $\binom{|Q|}{\phi|Q|}$ can reach 2.39×10^{37} if we set $|Q|$ and ϕ to 128 and 0.5 respectively. To this end, we design a Dijkstra-based algorithm to compute g_ϕ (shown in Section III-A). Although it also searches in an enumeration way, it is much more efficient than the naive method. Inspired by the threshold algorithm [25], *List* [8], [9] is proposed to answer FANN queries in the Euclidean space. We modify this *List*, and implement the modified algorithm in a "parallel and switchable" way to answer FANN queries in road networks. We denote it as *R-List* (shown in Section III-B). The modified implementation also leads to much more efficient solution to *max*-FANN problems (shown in Section IV-A). As presented in [24], IER has an excellent potential when retrieving kNN. Hence, we further design a IER-kNN framework to answer FANN queries. Based on the general IER-kNN framework, we can have a family of algorithms, such as IER-GTree, IER2-PHL (shown in Section III-C).

Although the universal methods can generally achieve good performance, they reply on sophisticated indexing techniques. The construction cost of index can often be very high especially for frequently changing road networks. Motivate by this, we design a specific algorithm when g is max, which can return an exact answer, and denote it as *Exact-max* (shown in Section IV-A). *Exact-max* follows the basic data structure of *R-List*, but it often outperforms any other methods when it is index-free. What is more, it is often desirable to obtain a near-optimal result. To this end, we design a *APX-sum* algorithm to answer *sum*-FANN queries in road networks, which can return a 3-approximation result. We further prove that it can even return a 2-approximation result if Q is the subset of P.

III. UNIVERSAL METHODS

In this section, we present the universal solutions to answer both *sum*-FANN and *max*-FANN queries.

A. The Dijkstra-based Algorithm

The Dijkstra-based algorithm is based on the following observation. Recall how *Dijkstra* routine runs: at every step of its expansion, it chooses an unvisited nearest node of the source node to visit and updates its neighbors' distances to the source node. This behavior also makes sense in running $g_\phi(p, Q)$. First, let p be the source node, we call a *Dijkstra*-like routine on it. Then we keep the path expanding until $\phi|Q|$ nodes in Q are labeled as visited. Hence, the $\phi|Q|$ is exactly Q^p_ϕ.

At a high level, the difference between the naive method mentioned in Section II-C and the Dijkstra-based algorithm is that: the former is to construct a Q_ϕ first, and then to determine the correct p and its flexible aggregate distance with respect to Q_ϕ; the latter is to choose a p first, and then retrieve the correct Q^p_ϕ and the flexible aggregate distance with respect to p.

Improvement for Terminate Early. Given a $g_\phi(p, Q)$, we use D to denote the set of distances of visited nodes with respect to p. We can maintain the last computed d^p as a bound and pass it to the next iteration in order to terminate early, and this strategy is always useful for most algorithms in this paper. To be specific, assume that it has expanded to some q, then the algorithm should terminate if:

- $\delta(p, q) \geq d^p$ when g is max.
- $\sum_{q_i \in D}\{\delta(p, q_i)\} + (\phi|Q| - |D|) \times \delta(p, q) \geq d^p$ when g is sum.

Intuitively, there are two ways to improve this baseline algorithm: prune nodes in P as many as possible (i.e., reduce the invocation of g_ϕ), or improve the implementation of g_ϕ. We will discuss related improving techniques in the following sections.

B. The R-List Algorithm

The *R-List* is actually queue-based. The basic idea of *R-List* algorithm is to construct a threshold for early terminating (as mentioned in Section III-A). We create $|Q|$ queues, and each queue corresponds to an object in Q, and processes data points in a from-near-to-far way. Suppose *head* is a function get the head element from L_i, and we denote the threshold as τ which is a lower bound of d^*.

At the initialization stage, we set d^* to infinity and create a *list* of queues (line 1-2). For q_i in Q, we denote the queue corresponding to q_i is L_i. L_i contains *all* nodes in P initially, and those nodes are sorted by their distances to q_i in ascending order. During every iteration, we collect $\phi|Q|$ heads with the smallest distances from the queues (line 4). Let τ be the *sum* or *max* of these $\phi|Q|$ smallest distances (line 5). If the $\tau \geq d^*$ holds, we can terminate this algorithm safely (line 6-7). Otherwise, we examine the queue whose head has the smallest distance, and denote it as L_{min} (line 8). If $head(L_{min})$ has not been visited ever, we call g_ϕ on it, and update the result if necessary, and then label it as visited (line 10-13). After that, we pop the head from L_{min} (line 14). If this queue is empty now, we can also terminate the algorithm safely (line 15-16). We show this process in Algorithm 1.

Algorithm 1: The *R-List* Algorithm

```
Input: $G$, $P$, $Q$, $\phi$, $g$
Output: $p^*$, $Q^*_\phi$, $d^*$
1  $d^* \leftarrow \infty$
2  initialize $|Q|$ queues
3  while true do
4      $D \leftarrow \phi|Q|$ smallest values of
         $\{\delta(head(L_i), q_i), \forall q_i \in Q\}$
5      $\tau \leftarrow sum$ or $max$ of $D$
6      if $\tau \geq d^*$ then
7          break
8      $L_{min} \leftarrow$the queue whose head has the smallest
         distance
9      if $head(L_{min})$ has not been visited before then
10         $(Q^{min}_\phi, d^{min}) \leftarrow g_\phi(head(L_{min}), Q)$
11         if $d^{min} < d^*$ then
12             $p^* \leftarrow head(L_{min})$, $Q^* \leftarrow Q^{min}_\phi$, $d^* \leftarrow d^{min}$
13         label $head(L_{min})$ as visited
14     pop $head(L_{min})$ from $L_{min}$
15     if $L_{min}$ is empty then
16         break
```

Although Algorithm 1 shares the basic idea with *List* [8], [9], the implementation details of constructing the *list* of queues are different in the context of road networks. What is more, the modified implementation can lead to a much more efficient solution to *max*-FANN problems. To this end, we will discuss the detailed implementation together with *Exact-max* algorithm in Section IV-A.

C. IER-kNN Framework

In this section, we propose a simple rather powerful algorithm framework for FANN query. Firstly, we adopt the Incremental Euclidean Restriction (IER) to prune the unnecessary nodes in P as many as possible. Let e be an entry of an R-tree that indexes P and b its *minimum bounding rectangle* (MBR). We can calculate the minimum possible distance from b to a point q, denoted as $mdist(b, q)$. Let e' be another R-tree node, and we also use $mdist(b, e')$ to denote the minimum possible distance from b to the MBR of e' for simplicity. We denote g^ϵ as the *Euclidean aggregate function*, and we have

$$g^\epsilon(p, Q) = g\{\delta^\epsilon(p, q_1), \ldots, \delta^\epsilon(p, q_{|Q|})\}.$$

Similarly, $g^\epsilon(e, Q)$ can be defined by $g\{mdist(b, q_i), \forall q_i \in Q\}$. Like Definition 1, we can define the *flexible Euclidean aggregate function* (Euclidean FANN) by $g^\epsilon_\phi(p, Q)$ if replacing $g(p, Q_\phi)$ with $g^\epsilon(p, Q_\phi)$. Following the similar definition, we also have $g^\epsilon_\phi(e, Q)$ with respect to an MBR e. Now, we can prove the following lemma:

Lemma 1. Let Q be a set of query points and e an R-tree node entry. For any point p indexed under e, $g^\epsilon_\phi(e, Q)$ cannot be larger than $g_\phi(p, Q)$.

Proof: It is true due to $g^\epsilon_\phi(e, Q) \leq g^\epsilon_\phi(p, Q)$ and $g^\epsilon_\phi(p, Q) \leq g_\phi(p, Q)$. ■

Based on Lemma 1, we can solve the FANN query using an R-tree built on P. We show this process in Algorithm 2. Initially, the root of the R-tree is pushed into a priority queue which is sorted by $g^\epsilon_\phi(e, Q)$ in *ascending* order (line 2). For each iteration, we first check whether $g^\epsilon_\phi(e, Q)$ is larger than or equal to current best candidate result (line 5). If so, we terminate the algorithm (line 6); otherwise, we check whether the dequeue item is an R-tree node (line 8). If so, we push all entries under this node into the priority queue (line 9-10); otherwise, we run $g_\phi(p, Q)$ and update the result if necessary (line 12-14). Note that in line 9, the entry $\hat{e}$ is a data point in P if e is a leaf node, and it is an R-tree node if e is a non-leaf node.

Algorithm 2: IER-kNN Framework

```
Input: $G$, $P$, $Q$, $\phi$, $g$, $R$
Output: $p^*$, $Q^*_\phi$, $d^*$
1  $d^* \leftarrow \infty$, $H \leftarrow$ new priority queue
2  $H.push(R.root, g^\epsilon_\phi(R.root, Q))$
3  while $H$ is not empty do
4      $e \leftarrow H.top()$
5      if $g^\epsilon_\phi(e, Q) \geq d^*$ then
6          break
7      $H.pop()$
8      if $e$ is an R-Tree node then
9          foreach R-Tree entry $\hat{e}$ under $e$ do
10             $H.push(\hat{e}, g^\epsilon_\phi(\hat{e}, Q))$
11     else
12         $(Q^e_\phi, d^e) \leftarrow g_\phi(e, Q)$
13         if $d^e < d^*$ then
14             $p^* \leftarrow e$, $d^* \leftarrow d^e$, $Q^*_\phi \leftarrow Q^e_\phi$
```

Re-visitation of $g_\phi(p, Q)$. Now we revisit the implementation of $g_\phi(p, Q)$. As implied in Section III-A, given a p, Q and ϕ, $g_\phi(p, Q)$ is exactly an *incremental network expansion* (INE), which is also a kNN query where p is a query node and Q is the set of data objects and $k = \phi|Q|$.

Note that [4] uses A* to compute the shortest path distance due to its better performance compared to Dijkstra. However, A* is necessarily no better than INE, as proved in our experiments. To boast fast speed of computing the shortest path distance, we apply two state-of-the-art techniques. The first is G-tree [11], [21], which materializes distance matrix for each tree node. The second is *pruned highway labelling* (PHL)

[16], which accelerates the shortest path distance queries by decomposing a graph into shortest paths and storing distances from each vertex to the shortest path in its labels. We denote the IER-kNN framework combined with the INE algorithm as IER-INE. Similarly, we also have IER-A*, IER-GTree and IER-PHL to represent the IER-kNN framework combined with A*, G-tree and PHL respectively.

$g_\phi(p, Q)$ itself can also be implemented with IER and any shortest path distance algorithm when Q is indexed under R-tree. In this way, we denote the IER-kNN method whose g_ϕ is implemented with IER-A* as IER2-A* (there are double IER routines). The IER-A* here means the kNN (or g_ϕ) method, instead of the FANN approach mentioned above. Similarly, if we replace A* here with GTree (or PHL), we also have IER2-GTree (or IER2-PHL). Note that the "GTree" in IER-GTree is the kNN algorithm presented in [11], [21], which is based on an *occurrence list* (Occ) over Q, and the "GTree" in IER2-GTree denates the shortest path distance algorithm based on G-Tree index.

The index techniques of different implementations for g_ϕ are summarized in Table II, and they will be further studied in the experimental section.

TABLE II
ROAD NETWORK INDEX OF g_ϕ

Algorithm Name	G-tree	PHL	R-tree	Occ
INE	✗	✗	✗	✗
A^*	✗	✗	✗	✗
GTree	✓	✗	✗	✓
PHL	✗	✓	✗	✗
IER-A^*	✗	✗	✓	✗
IER-GTree	✓	✗	✓	✗
IER-PHL	✗	✓	✓	✗

In addition, we can also use another bound in line 5 of Algorithm 2 when we apply IER2-A^*, IER2-GTree, or IER2-PHL, which is not as tight as $g_\phi^\epsilon(e, Q)$, but can be computed cheaply. The new bound $d(p, Q)$ can be defined by:

- $d(p, Q) = mdist(b_Q, p)$ when g is max.
- $d(p, Q) = \phi|Q| \times mdist(b_Q, p)$ when g is sum.

where b_Q is the MBR of Q, and p can be either a data point or an R-tree node. It is obvious that we can terminate the algorithm if $d(e, Q) \geq d^*$ holds.

IV. SPECIFIC METHODS

In this section, we propose two specific algorithms to solve *sum*-FANN and *max*-FANN queries respectively.

A. The Exact-max Algorithm

We present this method in Algorithm 3, and call it *Exact-max*, which shares the similar idea and data structure of Algorithm 1. The main difference is that we add a counter for every point in P. Initially, these counters are set to 0 (line 2). During every iteration, we get the head node with the smallest distance (line 4), and then increase the counter associated with the head node by one (line 5). If the counter associated with the head node reaches $\phi|Q|$, the head node is exactly p^*, and then we can terminate the algorithm safely (line 6-9). Hence, we run the time-consuming g_ϕ only once (line 8). This is why *E-max* can be efficient. Besides, this also indicates that the different implementations of g_ϕ have little influence on *Exact-max*. In other words, we may get an adorable performance even if we do not build a road network index over the whole G. This property is appealing when road networks change frequently.

Algorithm 3: The *Exact-max* Algorithm

```
Input: $G$, $P$, $Q$, $\phi$, $\delta$, $g$
Output: $p^*$, $Q_\phi^*$, $d^*$
1  foreach $p \in P$ do
2      $count[p] \leftarrow 0$
3  while true do
4      $L_{min} \leftarrow$the queue whose head has the smallest
         distance
5      $count[head(L_{min})] \leftarrow count[head(L_{min})] + 1$
6      if $count[head(L_{min})] \geq \phi|Q|$ then
7          $p^* \leftarrow head(L_{min})$
8          $(Q_\phi^*, d^*) \leftarrow g_\phi(head(L_{min}), Q)$
9          break
10     pop $head(L_{min})$ from $L_{min}$
```

Implementation Details. Now we discuss the implementation details of *R-List* and *Exact-max*. The *list* of queues is constructed in an implicit way, or it will violate the memory limit if $O(|P||Q|)$ is larger enough. To be specific, We set the $|Q|$ query nodes instead of p as the multiple sources initially, and then execute *Dijkstra*-like routine on them simultaneously and independently. As shown in Algorithm 1 or 3, the queue operations are alternately performed on different queues, which implies that we can implement this *multi-source Dijkstra* procedure in a "parallel and switchable" way. All data structures related to different queues should be well preserved when the *Dijkstra* routine switches away, thus the interrupted search process can be reloaded and resumed when it switches back.

The correctness of this algorithm is easy to validate: the nature of the *Dijkstra* algorithm guarantees that the closer nodes to the source are, the earlier they will be visited. When a node is first visited by exact $\phi|Q|$ sources, it means that this node is the closest one to the $\phi|Q|$ sources. Thus Algorithm 3 can answer *max*-FANN queries correctly. Besides, there is a trade-off between space and time to boast speed. If the memory is permitted, we can maintain a distance dictionary for the visited data points of each query point in Q. Hence, we can directly obtain the flexible aggregate distance from those distance dictionaries without calling $g_\phi(h_\rho, Q)$.

Note that basic idea of *Exact-max* is different from g_ϕ which is implemented with *Dijkstra* or *INE*. The latter is to regard P as sources and then obtaining the kNN (i.e., the nodes in

Q are destinations), while the direction of expansion is quite the reverse: we regard the nodes in Q are sources and then expanding to P (i.e., the nodes in P are destinations). It is not hard to understand that *Exact-max* is very efficient when P is dense and $\phi|Q|$ is relatively small. In most reality cases, this is true because $|Q|$ is much smaller than $|P|$.

A running example. Let us see how *Exact-max* finds the max-FANN in the graph of Figure 1 whose $\phi = 50\%$. The expanding paths of $\{q_1, q_2, q_3, q_4\}$ are $\{p_3, \dots\}$, $\{p_3, \dots\}$, $\{p_4, \dots\}$ and $\{p_5, \dots\}$ respectively. It is obvious that the counter of p_3 will reach $\phi \times 4 = 2$ first. Hence the result of this max-FANN query is $p^* = p_3$, $d^* = 2$ and $Q^*_\phi = \{q_1, q_2\}$.

TABLE III
A COUNTER EXAMPLE OF SUM-FANN

Source	**Expanding**			
q_1	(0, q_1)	(4, p_2)	(12, p_3)	-
q_2	(0, q_2)	(2, p_1)	(10, p_2)	-
q_3	(0, q_3)	(11, p_1)	-	-
q_4	(0, q_4)	(14, p_4)	-	-
q_5	(0, q_5)	(15, p_2)	-	-

It is worth noting that this method cannot be used to answer *sum*-FANN queries. Table III illustrates a counter example. Suppose query nodes set Q is $\{q_1, q_2, q_3, q_4, q_5\}$ (any query node does not belong to $\{p_1, p_2, p_3, p_4, p_5\}$), and $\phi = 40\%$. Hence, $\phi|Q| = 2$. If we follow the idea of Algorithm 3, we would examine the queue of q_2 first, and visit p_1. Secondly, we would examine the queue of q_1, and visit p_2. Thirdly, we would examine the queue of q_2 again, and hence visit p_2 again. At this moment, the counter of p_2 reaches 2, and we terminate the algorithm. In this way, we have $p^* = p_2$, $d^* = 4+10 = 14$, and $Q^*_\phi = \{q_1, q_2\}$. However, the correct answer is $p^* = p_1$, $d^* = 2 + 11 = 13$, and $Q^*_\phi = \{q_2, q_3\}$.

B. The APX-sum Algorithm

For the *sum*-FANN problem in road networks, we present an approximation approach *APX-sum* in Algorithm 4. This algorithm is extremely simple, but it has a very good approximation quality in both theory and practice. Instead of considering the whole P, we only examine those data points which are the nearest neighbors of those query nodes in Q (line 2-4). Then we regard the candidate set as P, and run the FANN algorithm (whose g is sum) mentioned above (line 5). Hence, we reduce the number of candidate data points to $|Q|$, which is usually much smaller than $|P|$. This is why it can remarkably improve the search efficiency. Actually, it is even possible that the size of candidate set is smaller than $|Q|$, since different query points may have the same nearest data point neighbor. One of the most appealing properties of *APX-sum* is the stability when varying P because it is only affected by Q generally. Another appealing property *APX-sum* of is the good approximation quality. We can prove that the approximation ratio d^α/d^* of this algorithm is no more than 3.

Algorithm 4: The *APX-sum* Algorithm

```
  Input: $G$, $P$, $Q$, $\phi$, $\delta$, $g$
  Output: $p^\alpha$, $Q^\alpha_\phi$, $d^\alpha$
1 candidate ← ∅
2 foreach $q \in Q$ do
3     $p$ ← the nearest neighbor of $q$ in $P$
4     candidate.insert($p$)
5 FANN ($G$, candidate, $Q$, $\phi$, $sum$)
```

Theorem 1. Algorithm 4 returns a 3-approximation answer to any *sum*-FANN query in road networks.

Proof: Given an *sum*-FANN query, Algorithm 4 returns an approximate answer $(p^\alpha, Q^\alpha_\phi, d^\alpha)$, and the true optimal answer is (p^*, Q^*_ϕ, d^*). Let q^τ be the nearest node in Q^*_ϕ to p^*, and p^τ be the nearest node in P to q^τ. We have:

$$\delta(p^\tau, q^\tau) \le \delta(p^*, q^\tau) \tag{1}$$

And for any $q \in Q^*_\phi$, we have:

$$\delta(p^*, q^\tau) \le \delta(p^*, q) \tag{2}$$

$$\Rightarrow \phi M \cdot \delta(p^*, q^\tau) \le \sum_{q \in Q^*_\phi} \delta(p^*, q) = d^* \tag{3}$$

It is obvious that p^τ cannot be "better" than p^α. If the result of $g_\phi(p^\tau, Q)$ is (Q^τ_ϕ, d^τ), we have:

$$\begin{aligned}
d^\alpha &\le d^\tau \\
&= \sum_{q \in Q^\tau_\phi} \delta(p^\tau, q) \\
&\le \sum_{q \in Q^*_\phi} \delta(p^\tau, q) \\
&\le \sum_{q \in Q^*_\phi} (\delta(p^\tau, p^*) + \delta(p^*, q)) \qquad \text{(triangle inequality)} \\
&= \sum_{q \in Q^*_\phi} \delta(p^\tau, p^*) + d^* \\
&= \phi M \cdot \delta(p^\tau, p^*) + d^* \\
&\le \phi M \cdot (\delta(p^\tau, q^\tau) + \delta(q^\tau, p^*)) + d^* \\
&\le 2\phi M \cdot \delta(q^\tau, p^*) + d^* \qquad \text{(by Eqation 1)} \\
&\le 2d^* + d^* \qquad \text{(by Eqation 3)} \\
&= 3d^*
\end{aligned}$$

■

A running example. Let us see how *APX-sum* finds the sum-FANN in the graph of Figure 1 whose $\phi = 50\%$. First, we can easily obtain the candidates of data points $\{p_3, p_4, p_5\}$. Since the true optimal p^* belongs to the set of candidates, *A-sum* returns $p^* = p_3$, $d^* = 4$ and $Q^*_\phi = \{q_1, q_2\}$ as the final result.

It should be pointed out that, 3 is only a theoretical bound. As shown in our experiments, the approximation quality of Algorithm 4 is far less than 3 in practice: it never exceeds 1.2 in our experiments. Note that p^τ is also q^τ if Q is the subset of P, because in this case the nearest node in P to q^τ is q^τ itself. Hence, $\delta(p^\tau, q^\tau) = 0$ holds, and we can further prove that $d^\alpha \leq 2d^*$ holds. What is more, it will also avoid the cost of expanding to $|Q|$ nearest neighbors. In the following, we propose a new theorem without proof.

Theorem 2. Algorithm 4 returns a 2-approximation answer to any *sum*-FANN query in road networks if Q is a subset of P.

V. EXTENSION TO THE K-FANN PROBLEM

In this section, we define the k-FANN query, which is a further extension of FANN. We can regard FANN as a special case of k-FANN when k is 1.

Definition 3 (k-FANN query)**.** A k-FANN query is a six-tuple (G, P, Q, ϕ, g, k), which returns a k-element vector X as its answer. Each element of X is a pair (p_i, r_i), in which $i = 1, 2, \cdots, k$, $p_i \in P$ and r_i is the flexible aggregate distance from p_i to Q. For any node $p_0 \in P \backslash \{p_1, p_2, \cdots, p_k\}$, assume its flexible aggregate distance to Q is r_0, we have $r_0 \geq \max\{r_1, r_2, \cdots, r_k\}$.

Obviously, it is not necessary for $Q_\phi^{p_i}$ to be the same, where $i = 1, 2, \cdots, k$. With some minor modifications, all of the algorithms in this paper can be easily adapted to answer the k-FANN query except the *APX-sum* algorithm. The basic idea of those most modifications is mainly to add a priority queue to record the k best results found so far.

The Dijkstra-based Algorithm. To get the top-k, we need to update the queue when enumerating the P. Finally, the queue is our final result.

The *R-List* Algorithm. The threshold τ is calculated in the same way as the original *R-List* algorithm. Instead of comparing τ with the smallest distance, we compare it with the k-th smallest distance in the priority queue. And if τ is larger, we can terminate the algorithm and return the priority queue as the k-FANN answer.

The *IER-kNN* Framework. Like the adaptation for the *R-List* algorithm, instead of comparing $g_\phi^\epsilon(e, Q)$ with the smallest distance, we compare it with the k-th smallest distance in the priority queue. And if $g_\phi^\epsilon(e, Q)$ is larger, we can terminate the algorithm and return the priority queue as the k-FANN answer.

The *Exact-max* Algorithm. For the k-FANN problem, we should expand the paths until k different counters reach to $\phi|Q|$. And then, we can terminate the search routine and return the k corresponding query nodes and their flexible distances as the final answer to the k-FANN query.

VI. EXPERIMENTS

A. Setup

We implemented all the algorithms mentioned in this paper by standard C++, and executed our experiments on a Linux machine. All of our road network datasets come from the real word[1]. Note that the original datasets have many errors, such as unconnected components or self-loops, and we have cleaned it up at the preprocessing stage. We show them in Table IV.

TABLE IV
ROAD NETWORK DATASETS

Name	Description	# nodes	# edges
DE	Delaware	48,812	119,004
ME	Maine	187,315	412,352
COL	Colorado	435,666	1042,400
NW	Northwest USA	1,089,933	2,545,844
E	Eastern USA	3,598,623	8,708,058
CTR	Central USA	14,081,816	33,866,826
USA	Full USA	23,947,347	57,708,624

Given a road network, there are many factors that will affect the cost of an FANN query. In our experiments, we mainly focus on those in the following:

- d, the density of P
- A, the coverage ratio of Q
- M, the size of Q (i.e., $|Q|$)
- C, the number of clusters of Q
- ϕ, the flexibility parameter

where d controls the generation of P, and A, M and C affect the generation of Q.

Uniform data points. Parameter d reflects the ratio of the size of P to the size of nodes in whole graph (i.e., $|P|/|V|$). We assume that P is generated randomly in the road network given a density.

Uniform query points. Parameter A reflects aggregation degree of the query nodes in Q. We first randomly select a node in V as a source node (i.e., *seed* node), and calculate the shortest path distances from it to all other nodes in V. We denote the maximum one as the *radius* of G. Then we randomly choose M nodes from G, whose distances to the seed node are no more than $A \times radius$. Note that we assume the size of nodes in such region is always larger than or equal to M. If there is no enough objects in the region, we simply expand outward until the size reaches M.

Clustered query points. In some scenarios, the query points are not uniformly distributed. Some locations, such as schools, often occur in clusters. After we determine a region by A in the road network, we select C central nodes in the selected region, and choose M/C nodes in the vicinity of each central node by expanding from it.

We use the NW as our default road network. We vary d from 0.0001 to 1, and vary A from 1% to 20%, and vary M from 64 to 1024, and vary C from 1 to 8, and vary ϕ from 0.1 to 1. The default values of d, A, M, C and ϕ are set to 0.001, 10%, 128, 1 and 0.5 respectively. These values are set stochastically and others are also adoptable. By default, both P and Q are generated uniformly. In order to minimize the randomness, we average the results of algorithms over 100 queries. The performances of R-tree and G-tree are dependent on the value

[1]http://www.dis.uniroma1.it/challenge9/download.shtml

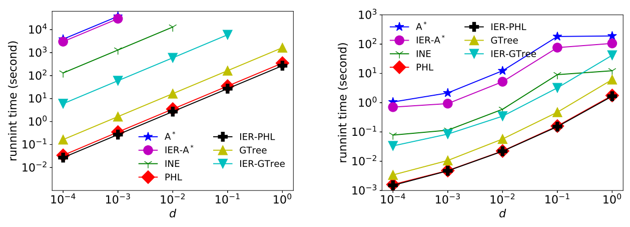
of fanout f (both R-tree and G-tree) and maximum number of points in a leaf node τ (G-tree). In our experiments, we set $f = 4$. And for G-tree, we set τ to 64 (DE), 128 (ME, COL), 256 (NW, E), and 512 (CTR, USA) respectively.

In the following paper, we always use baseline to denote the generalized Dijkstra-based algorithm, since the *Dijkstra* here can be regarded as INE and it can be replaced with other methods (e.g., A*, PHL, and G-tree). Note that we only show the results of *max*-FANN for universal methods (i.e., baseline, *R-List*, and IER-kNN) due to the space limit except for the evaluation for approximation ratio in Section VI-C. Fortunately, the running time of *sum*-FANN and *max*-FANN is very close given a specific input.

B. Efficiency

Now we focus on the efficiency of different algorithms. The experimental results of their running time are shown in the following, with respect to the varying parameters of d, A, M, C and ϕ.

Varying d. We use $0.0001, 0.001, 0.01, 0.1, 1$ to vary d. Firstly, we present the experimental results of baseline algorithms and IER-kNN which are implemented by different g_ϕ routines in Fig. 2. Note that the legends here refer to g_ϕ methods. We can find that PHL and IER-PHL always preform best while A* and IER-A* are worst among these different implementations. Another important conclusion is that there is a linear (or sub-linear) relationship between the running time and density of P for baseline (or IER-kNN) algorithms. The IER version of A* is slightly better than A* in generally. Comparing Fig. 2(a) with 2(b), it is obviously that IER-kNN outperforms baseline the efficiency by 1-3 orders of magnitude with respect to the same g_ϕ. From Fig. 2(a), we can conclude that baseline algorithm is often infeasible if it is combine with A*, IER-A*, or INE. And since PHL (or IER-PHL) is the most efficient implementation of g_ϕ, we choose PHL as the default implementation of g_ϕ in the latter experiments. It is worth noting that the reason why the baseline algorithm can also achieve acceptable performance is that it adopts PHL, which is fast enough. For example, if we use INE as the implementation of g_ϕ, we can observe the difference between baseline and *R-List* with respect to the running time clearly. We will discuss more about this in Section VI-F.



(a) Baseline (b) IER-kNN

Fig. 2. Efficiency of baseline and IER-kNN implemented by different g_ϕ

Fig. 3 shows the efficiency of all algorithms presented in this paper. With the increasing of d, IER-PHL performs best

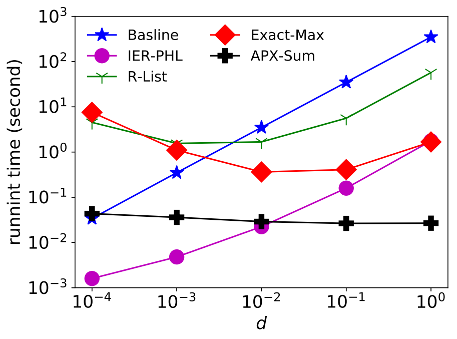
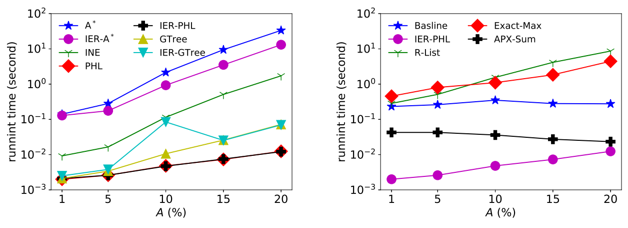


Fig. 3. Efficiency of all algorithms

at first and *APX-sum* outperforms any other method when d is larger than 0.01. We also find that *R-List* is better than the baseline algorithm when d is large. We validate the stability of *APX-sum* when varying d, and this is because *APX-sum* depends much on Q instead of P. Both *APX-sum* and *Exact-max* have the tendency that they cost less time when d is larger. This is because the expanding routine from Q to P is faster when the data points is denser. The reason why *Exact-max* decreases first and then increases is due to the trade-off between the expanding overhead and convenience brought by d. To be specific, a larger d will lead to a higher expanding overhead in general, while it will also make the terminating condition to be fulfilled earlier.

Varying A. Now we study the efficiency with different coverage ratios of Q. We use $1\%, 5\%, 10\%, 15\%, 20\%$ to vary M. We show the results in Fig. 4.



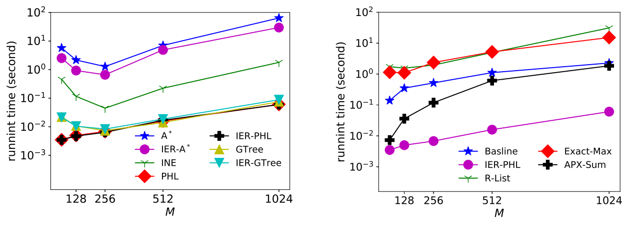
(a) IER-kNN (b) All algorithms

Fig. 4. Efficiency when varying A

We can conclude that IER-kNN can reflect how the efficiency is affected by different g_ϕ routines when varying parameters from the last experiment. Fig. 4(a) shows the results of IER-kNN implemented by different g_ϕ when we vary A. Clearly, PHL and IER-PHL outperform other methods. And all will cost more running time with the increase of A. This is because larger A means that Q is sparser in road networks, and hence it will travel within a larger region in general. As shown in Fig. 4(a), the slopes of A*, IER-A* and INE are larger than others. This behavior is consistent with the their nature of "expanding".

Further, we show the results of all algorithms in Fig 4(b). Clearly, *APX-sum* is stable when we vary A. This validates our conclusion that *APX-sum* has little dependence on Q. As for *R-List* and *Exact-max* algorithm, their performances are bad if Q is sparse so that expanding from Q to P is slow. We can also find the baseline algorithm is stable for different A. This is not surprising, as the baseline method depends much on P. Finally, *R-List* and *Exact-max* are not better than the baseline algorithm here, and this is verified by the efficiency in Fig. 3 when $d = 0.001$.

Varying M. Our next experiment investigates the efficiency when varying the size of Q. We use $64, 128, 256, 512, 1024$ to vary M. The results are shown in Fig. 5. As for IER-kNN methods, they have the tendency that larger M leads to worse efficiency in general. Note that the running time decreases first (from $M = 64$ to $M = 256$) for most IER-kNN methods. This is due to the trade-off between M and the sparsity of Q. To be specific, the smaller M results in a larger sparsity given a coverage ratio of Q . We also find that IER version of A* can improve the performance generally. Besides, the differences among PHL, GTree, IER-PHL, and IER-GTree are minor in Fig. 5(a). And Fig. 5(b) shows the results of all algorithms presented in this paper. We can clearly find that *APX-sum* increases with the increase of M, and this verifies our conclusion that *APX-sum* depends much on the size of Q.

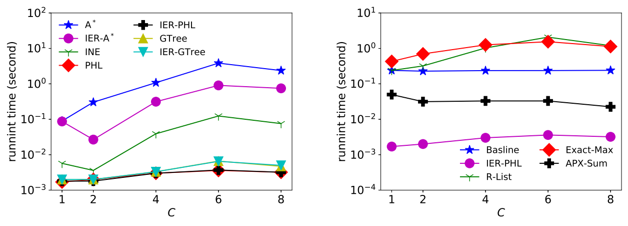


(a) IER-kNN (b) All algorithms

Fig. 5. Efficiency when varying M

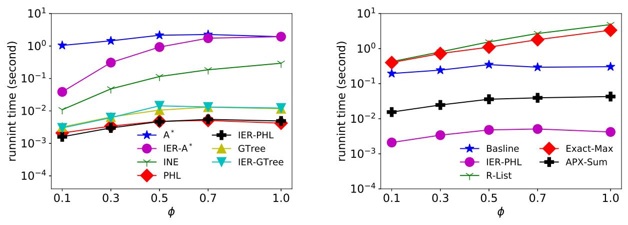
Varying C. We evaluate the effect of C if Q is generated in clusters, rather than uniformly. We use $1, 2, 4, 6, 8$ to vary C. Fig. 6 shows the results. Clearly, the larger C leads to worse performance in general, and this tendency is more serious for "expanding-based" methods in Fig. 6(a). When C is larger enough, the performance will be stable and it will approximate to that when Q is generated uniformly. For example, the running time of IER-A* is 2.16 seconds if the Q is generated uniformly, while the cost is 2.37 seconds when $C = 8$. As shown in Fig. 6(b), *R-List* and *Exact-max* are more effected by C due to their similar mechanisms.

Varying ϕ. The final part of this section is to study the effect of ϕ, i.e., the flexibility parameter. We use $0.1, 0.3, 0.5, 0.7, 1.0$ to vary ϕ. Fig. 7 shows our results. There is an obvious positive correlation with the ϕ. This is reasonable because the larger ϕ means that more destinations need to be visited. We can find that R-tree over Q has less improvement for A* with the increasing of ϕ, however, the improvements is very obvious when ϕ is relatively small. Fig. 7(b), we can find that *R-List* and *Exact-max* are more effected by ϕ.



(a) IER-kNN (b) All algorithms

Fig. 6. Efficiency when varying C



(a) IER-kNN (b) All algorithms

Fig. 7. Efficiency when varying ϕ

C. *Approximation Quality of* A-sum

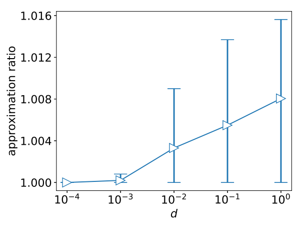
The *APX-sum* algorithm proposed in Section IV-B is an approximate algorithm. We show its approximation quality in Fig. 8. The y-error in error bars presents the standard deviation. The results show that *APX-sum* algorithm can achieve excellent approximation quality in all of the experimental cases, and the average ratio of r over r^* is always less than 1.2. Clearly, we notice that this approximate algorithm is always quite stable when we vary d, A, M, C and ϕ.

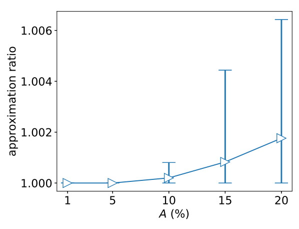
D. *Road Network Index Cost*

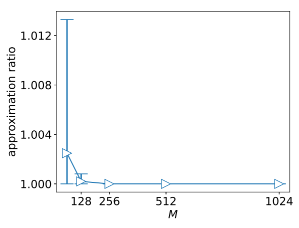
Here we measure the construction time and size of road network index used in our algorithms. The different index techniques are presented in Table II. For the index over whole road networks, we focus on G-tree and PHL. As for the query objects, we focus on R-tree and Occ. We show the experimental results in Fig. 9.

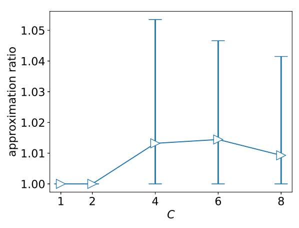
Index Size of G-tree and PHL. Fig. 9(a) shows the index size of G-tree and PHL for different datasets. Generally, G-tree costs less storage than PHL. Note that PHL only can build index for the first 5 datasets before exceeding the memory capacity. We observe that the size of PHL for E is 6.27GB while the size of G-tree for USA is only 4.35GB. This experimental result suggests that G-tree is the only choice for large road network (e.g. USA, whose size of nodes is 23,947,347) if the memory is limited, although PHL is generally faster than G-tree.

Construction Time of G-tree and PHL. Fig. 9(b) shows the result of index construction time for different road networks. Since PHL cannot build index for CTR and USA due to memory limit, the data of construction time for CTR and USA is not showed. It is obvious that they are very close generally.


(a) Varying d


(b) Varying A


(c) Varying M


(d) Varying C

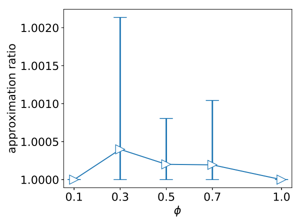
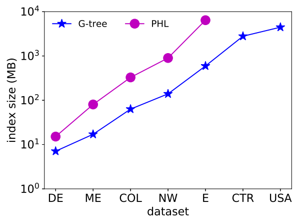
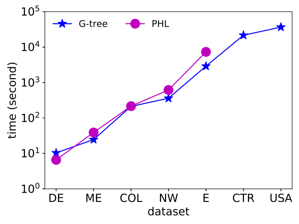
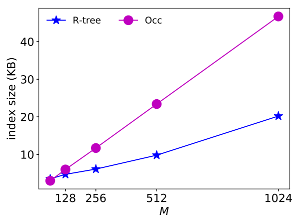

(e) Varying ϕ

Fig. 8. Approximation quality of *A-sum*


(a) Index size of G


(b) Construction time of G


(c) Index size of Q

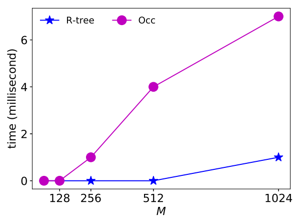

(d) Construction time of Q

Fig. 9. Index cost of different road networks

Hence, the difference of index construction time is trivial. Based on the compared experiments of index size, we can conclude that the major considerations are running efficiency and memory capacity rather than the index construction time when choosing from G-tree and PHL.

Index Cost of R-tree and Occ. Fig. 9(c) and 9(d) show the index size and construction time of R-tree and Occ when varying the size of Q. We can find that although Occ always costs more time and size than R-tree, the differences are still trivial compared with the running cost. Hence, the index cost of Q can be ignored when we need to make a choice between GTree and IER-GTree to implement g_ϕ. The running cost is the only consideration.

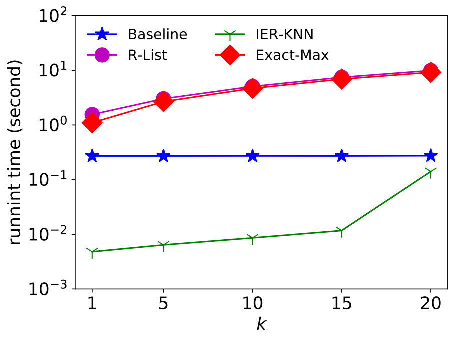


Fig. 10. Efficiency of k-FANN

E. *Evaluation for k-FANN Queries*

We now study performance of the k-FANN queries of different algorithms. We use $1, 5, 10, 15, 20$ to vary k. Fig. 10 shows the results. It is obvious that the query time will increase with the increasing of k in k-FANN problem except for the baseline algorithm, which is very stable. As mentioned above, the stability of baseline is due to its dependence on P, and thus the computation cost is similar with the case when $k = 1$. We can also notice that both the *Exact-max* and *R-List* algorithms are more sensitive to the increasing k than others. This is because they need more expanding overheads when k increases.

F. *Revisit Efficiency of Baseline Algorithm*

As shown in experimental results above, we can find that the baseline is even better than *R-List* (or *Exact-max*) in some scenarios if g_ϕ is implemented by PHL. This is because PHL is fast enough such that the advantage of *R-List* (or *Exact-max*) is shadowed.

First, we show the experimental results of *Exact-max* implemented by different g_ϕ routines in Table V. Although we find that g_ϕ methods vary much in Fig. 2, it has little influence on *Exact-max* since there is only one invocation of g_ϕ. Clearly, *Exact-max* outperforms the baseline method by 2 orders if g_ϕ is implemented by A* even if $d = 0.0001$. The result indicates that we can answer *max*-FANN by *Exact-max* even if there is no any precomputed index over the whole road network. We can conclude that the baseline algorithm is often infeasible without PHL or G-tree.

Second, we use $0.0001, 0.001, 0.01, 0.1, 1$ to vary d and compare the *R-List* with the baseline algorithm with respect to the efficiency when g_ϕ is implemented by INE, and Fig. 11 shows the result. We can also conclude that *R-List* is much better than the baseline algorithm if there is no any precomputed

TABLE V
EFFICIENCY OF *E-max* WITH DIFFERENT g_ϕ (SECOND)

g_ϕ \ d	0.0001	0.001	0.01	0.1	1
A*	7.26	1.24	0.65	0.69	2.05
IER-A*	7.07	1.05	0.47	0.50	1.79
INE	6.81	0.94	0.35	0.38	1.62
PHL	7.56	1.10	0.36	0.41	1.67
IER-PHL	7.59	1.07	0.36	0.40	1.68
GTree	6.77	0.92	0.34	0.37	1.60
IER-GTree	6.78	0.93	0.34	0.38	1.64

index over the whole road network, and there is no doubt that the baseline algorithm highly depends on g_ϕ.

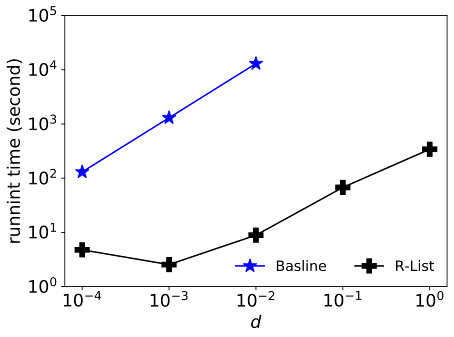


Fig. 11. Efficiency of *List* and baseline

VII. CONCLUSION

FANN query is a further extension of the classic ANN problem. It introduces a user-defined flexibility parameter and supports a more flexible searching in many applications. In this paper, we have studied the interesting problem of FANN queries in road networks. We proposed a series of algorithms, including a Dijkstra-based algorithm, the *R-List* and a general algorithm framework (IER-kNN), to solve this problem, and made a comprehensive study about the efficiency and characteristics of them. Combined with the state-of-the-art shortest path distance computing techniques, the proposed algorithms can achieve much better performance than the naive solution. Our specific approaches sacrifice some generalities, but are much more efficient than those universal approaches especially when the road networks change frequently such that it is infeasible to build a road index in the running time. In other words, both *Exact-max* and *APX-sum* can achieve acceptable performance even if there is no any index over a road network. The *Exact-max* can give an exact answer, while the *APX-sum* algorithm can return a near-optimal result with a guaranteed 3-approximation, and the approximation ratio can even reach to 2 if the set of query objects is the subset of the set of data objects. Besides, these specific methods are much easier to implement than the universal methods. Finally, we extend FANN to k-FANN and successfully adapt most of the proposed algorithms to answer k-FANN queries.

REFERENCES

[1] D. Papadias, Q. Shen, Y. Tao, and K. Mouratidis, "Group nearest neighbor queries," in *ICDE*. IEEE, 2004, pp. 301–312.
[2] D. Papadias, Y. Tao, K. Mouratidis, and C. K. Hui, "Aggregate nearest neighbor queries in spatial databases," *ACM TODS*, vol. 30, no. 2, pp. 529–576, 2005.
[3] F. Li, B. Yao, and P. Kumar, "Group enclosing queries," *IEEE TKDE*, vol. 23, no. 10, pp. 1526–1540, 2011.
[4] M. L. Yiu, N. Mamoulis, and D. Papadias, "Aggregate nearest neighbor queries in road networks," *IEEE TKDE*, vol. 17, no. 6, pp. 820–833, 2005.
[5] D. Yan, Z. Zhao, and W. Ng, "Efficient algorithms for finding optimal meeting point on road networks," *PVLDB*, vol. 4, no. 11, 2011.
[6] M. Safar, "Group k-nearest neighbors queries in spatial network databases," *JGS*, vol. 10, no. 4, pp. 407–416, 2008.
[7] L. Zhu, Y. Jing, W. Sun, D. Mao, and P. Liu, "Voronoi-based aggregate nearest neighbor query processing in road networks," in *SIGSPATIAL*. ACM, 2010, pp. 518–521.
[8] Y. Li, F. Li, K. Yi, B. Yao, and M. Wang, "Flexible aggregate similarity search," in *SIGMOD*. ACM, 2011, pp. 1009–1020.
[9] F. Li, K. Yi, Y. Tao, B. Yao, Y. Li, D. Xie, and M. Wang, "Exact and approximate flexible aggregate similarity search," *VLDBJ*, vol. 25, no. 3, pp. 317–338, 2016.
[10] Z. Xu and H.-A. Jacobsen, "Processing proximity relations in road networks," in *SIGMOD*. ACM, 2010, pp. 243–254.
[11] R. Zhong, G. Li, K.-L. Tan, and L. Zhou, "G-tree: An efficient index for knn search on road networks," in *CIKM*. ACM, 2013, pp. 39–48.
[12] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1959.
[13] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Trans. Syst. Sci. Cybern.*, vol. 4, no. 2, pp. 100–107, 1968.
[14] N. Jing, Y.-W. Huang, and E. A. Rundensteiner, "Hierarchical encoded path views for path query processing: An optimal model and its performance evaluation," *IEEE TKDE*, vol. 10, no. 3, pp. 409–432, 1998.
[15] S. Jung and S. Pramanik, "An efficient path computation model for hierarchically structured topographical road maps," *IEEE TKDE*, vol. 14, no. 5, pp. 1029–1046, 2002.
[16] T. Akiba, Y. Iwata, K.-i. Kawarabayashi, and Y. Kawata, "Fast shortest-path distance queries on road networks by pruned highway labeling," in *ALENEX*. SIAM, 2014, pp. 147–154.
[17] H. Bast, S. Funke, and D. Matijević, "Transit: ultrafast shortest-path queries with linear-time preprocessing," in *9th DIMACS Implementation Challenge—Shortest Path*, 2006.
[18] R. Geisberger, P. Sanders, D. Schultes, and D. Delling, "Contraction hierarchies: Faster and simpler hierarchical routing in road networks," in *Experimental Algorithms*. Springer, 2008, pp. 319–333.
[19] K. C. Lee, W.-C. Lee, B. Zheng, and Y. Tian, "Road: A new spatial object search framework for road networks," *IEEE TKDE*, vol. 24, no. 3, pp. 547–560, 2012.
[20] M. Kolahdouzan and C. Shahabi, "Voronoi-based k nearest neighbor search for spatial network databases," in *VLDB*, 2004, pp. 840–851.
[21] R. Zhong, G. Li, K. Tan, L. Zhou, and Z. Gong, "G-tree: An efficient and scalable index for spatial search on road networks," *IEEE TKDE*, vol. 27, no. 8, pp. 2175–2189, 2015.
[22] G. R. Hjaltason and H. Samet, "Distance browsing in spatial databases," *ACM TODS*, vol. 24, no. 2, pp. 265–318, 1999.
[23] N. Roussopoulos, S. Kelley, and F. Vincent, "Nearest neighbor queries," in *SIGMOD*, vol. 24, no. 2. ACM, 1995, pp. 71–79.
[24] T. Abeywickrama, M. A. Cheema, and D. Taniar, "K-nearest neighbors on road networks: a journey in experimentation and in-memory implementation," *PVLDB*, vol. 9, no. 6, pp. 492–503, 2016.
[25] N. Bruno and H. Wang, "The threshold algorithm: From middleware systems to the relational engine," *IEEE TKDE*, vol. 19, no. 4, pp. 523–537, 2007.