Sun WW, Chen CN, Zhu L *et al.* On efficient aggregate nearest neighbor query processing in road networks. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 30(4): 781–798 July 2015. DOI 10.1007/s11390-015-1560-z

On Efficient Aggregate Nearest Neighbor Query Processing in Road Networks

Wei-Wei Sun[1,2] (孙未未), *Senior Member, CCF, ACM*, Chu-Nan Chen[1,2] (陈楚南), Liang Zhu[1,2] (朱 良)
Yun-Jun Gao[3] (高云君), *Senior Member, CCF, Member, ACM, IEEE*
Yi-Nan Jing[1,2] (荆一楠), *Member, ACM, IEEE*, and
Qing Li[4] (李 青), *Senior Member, IEEE, Member, ACM*

[1]*School of Computer Science, Fudan University, Shanghai 201203, China*
[2]*Shanghai Key Laboratory of Data Science, Fudan University, Shanghai 201203, China*
[3]*College of Computer Science, Zhejiang University, Hangzhou 310027, China*
[4]*Department of Computer Science, City University of Hong Kong, Kowloon, Hong Kong, China*

E-mail: {wwsun, chenchunan, zhuliang}@fudan.edu.cn; gaoyj@zju.edu.cn; jingyn@fudan.edu.cn; itqli@cityu.edu.hk

Received February 1, 2015; revised May 12, 2015.

Abstract An aggregate nearest neighbor (ANN) query returns a point of interest (POI) that minimizes an aggregate function for multiple query points. In this paper, we propose an efficient approach to tackle ANN queries in road networks. Our approach consists of two phases: searching phase and pruning phase. In particular, we first continuously compute the nearest neighbors (NNs) for each query point in some specific order to obtain the candidate POIs until all query points find a common POI. Second, we filter out the unqualified POIs based on the pruning strategy for a given aggregate function. The two-phase process is repeated until there remains only one candidate POI, and the remained one is returned as the final result. In addition, we discuss the partition strategies for query points and the approximate ANN query for the case where the number of query points is huge. Extensive experiments using real datasets demonstrate that our proposed approach outperforms its competitors significantly in most cases.

Keywords ANN query, spatial database, road network

1 Introduction

Nearest neighbor (NN) search and its variants in road networks have been well-studied in the past few years, due to their importance in a wide spectrum of applications. The aggregate nearest neighbor (ANN) query, one of NN query variations, returns the point of interest (POI) that minimizes an aggregate function with respect to a set of query points[1]. This type of queries is often raised in our real life. For example, some friends at different locations would like to find a meeting place (e.g., a restaurant) on weekend. Different from conventional NN queries, in an ANN query, there are multiple query points, and the query result depends on the specified aggregate function, which can usually be sum, max, or min. Take the above problem of finding a meeting place as an example, the sum function makes sure that the total distance (or time) traveled by all persons is minimum, while the max/min function aims to minimize the maximum/minimum traveling distance (or time) requested by any person.

An example of a road network is depicted in Fig.1. The points n_1, n_2, …, n_7 represent the nodes in the road network, and the number on each edge is the distance (or time) to travel the corresponding road segment. A set of POIs $P = \{p_1, p_2, p_3\}$ (e.g., restaurants) and a set of query points $Q = \{q_1, q_2\}$ (e.g., persons) are located on the edges of the road network.

Regular Paper

Special Section on Data Management and Data Mining

This research is supported in part by the Shanghai Natural Science Foundation of China under Grant No. 14ZR1403100, the Shanghai Science and Technology Development Funds of China under Grant Nos. 13dz2260200 and 13511504300, and the National Natural Science Foundation of China under Grant No. 61073001.

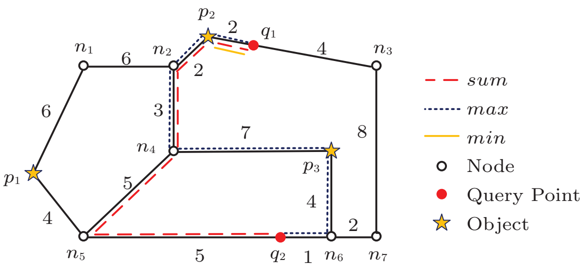


Fig.1. ANN queries for different aggregate functions.

Example 1. If the aggregate function is *sum*, the ANN for Q is p_2. Similarly, if the aggregate function is *max*, the ANN for Q is p_3, and if the aggregate function is *min*, the ANN for Q is p_2. The shortest paths from all query points to the ANNs for different aggregate functions are illustrated in Fig.1.

As shown in the aforementioned example, it is more complex to handle ANN queries in road networks than those in Euclidean spaces, because the location and the accessibility of the POIs are restricted by the computation of network distance (i.e., the length of the shortest path connecting two points). Existing algorithms for processing ANN queries in road networks[1] have two limitations. 1) They are not applicable when the edge weights are not proportioned to their physical lengths. For example, in some real applications like traffic networks, the weight of edges is represented by the traveling time. 2) The query performance is not very efficient when the number of query points is huge, as demonstrated by our experimental results.

In this paper, we propose an efficient algorithm for ANN queries in road networks, which avoids the shortcomings mentioned above. With effective pruning techniques, our approach filters out the unqualified candidate POIs, and outperforms the current state-of-the-art competitors in most cases. In general, most of POIs such as restaurants and hotels are usually stationary on the road network. Therefore, in this paper, we will not consider moving objects, which will be studied in our future work. Thus we tackle the ANN query by employing the network Voronoi diagram (NVD). Specifically, our approach employs the Voronoi-based kNN query processing algorithm[2] to expand the search space gradually, and enables effective pruning strategies to speed up the query processing. Furthermore, we present an approximate algorithm for processing ANN queries in road networks (with extremely small cost) when the number of query points is very large. The key contributions of this paper are summarized as follows.

- We solve the ANN query in a road network using NVD. To the best of our knowledge, this is the first work to deal with the ANN query based on the NVD.
- We develop effective pruning strategies for *sum* aggregate function and *max* aggregate function respectively to prune the unqualified POIs that cannot be ANNs.
- We present an efficient NVD-based algorithm for processing ANN queries in road networks, prove its correctness, and analyze its time complexity.
- We propose an approximate algorithm for ANN queries with a large number of query points in road networks.
- We conduct extensive experiments with real datasets to demonstrate the effectiveness of our presented strategies and the efficiency of our proposed algorithms.

A preliminary report of this study appears in [3], where only ANN queries with *sum* function are considered. In this paper, we extend the preliminary version by 1) studying ANN queries with *max* function; 2) proposing an approximate algorithm for ANN queries with a large number of query points to speed up the query processing; 3) conducting a more comprehensive experimental evaluation to examine more performance aspects.

The rest of this paper is organized as follows. Section 2 reviews the related work. Section 3 presents preliminaries. Section 4 elaborates our approach to tackle ANN queries in road networks, proves its correctness, and analyzes its time complexity. Section 5 discusses the approximate ANN query in road networks. Extensive experimental results and our findings are reported in Section 6. Finally, Section 7 concludes the paper with some directions for future work.

2 Related Work

NN search is one of the oldest problems in computer science. Many algorithms based on R-trees[4] for processing NN queries have been proposed in the database literature. These algorithms follow either the depth-first (DF)[5-6] or the best-first (BF)[7] traversal paradigm. As demonstrated in [7] and [8], the DF-based algorithm accesses more nodes than necessary, while the BF-based algorithm achieves the optimal I/O performance by only visiting the nodes necessary. Furthermore, the BF-based algorithm is incremental, i.e., it returns the NNs in ascending order of their distances to the query point; and thus, k does not have to be known in advance. In addition, different variants of NN queries, such as continuous NN (CNN) search[9], range NN query[10], continuous obstructed NN search[11], reverse NN (RNN) retrieval[12], visible NN search[13], closest pair query[14], group NN search[15], ANN query[16], and so on, have been investigated as well.

However, the above NN query and its variants focus on the Euclidean space. In the context of road networks, they have also been well studied. In [2, 17-19], kNN queries in road networks are efficiently solved. In addition, some variations of NN queries in road networks (e.g., CNN[20], RNN[21]) and several interesting queries (e.g., distance join queries[22], multi-source skyline computation[23], optimal meeting point queries[24]) have been discussed in the literature.

As one of the most important queries in road networks, the ANN query was investigated by Yiu *et al.*[1] Three algorithms were proposed, namely Incremental Euclidean Restriction (IER), Threshold Algorithm (TA), and Concurrent Expansion (CE). IER uses the R-tree index to prune the search space by comparing the shortest aggregate distance with the Euclidean distance, while TA and CE incrementally expand the network around each query point and employ some top-k aggregate query processing techniques to guide and terminate the search. Our work is different from [1] as we utilize the NVD and effective pruning techniques to speed up the query processing, and we also explore the problem of approximate ANN search.

Another similar work is the Voronoi-based spatial skyline algorithm[25], which finds the skyline points to multiple query points by identifying the Voronoi polygons intersected with the convex hull of the query points. This differs from our work as the ANN query aims at finding the optimal aggregating point with respect to different aggregate functions, and the Voronoi polygons intersected with the convex hull may not be the result of an ANN query.

Qin *et al.*[26-27] addressed the problem of ANN query monitoring for moving objects in the Euclidean space. Li *et al.*[28] discussed the ANN queries whose query points are moving. Chen *et al.*[29] proposed the problem of ANN queries with keyword constraints for spatial-textual data. Hashem *et al.*[30] explored the ANN queries whose query points have location privacy concern. Lian *et al.*[31-32] addressed the ANN queries in uncertain databases and uncertain graphs.

3 Preliminaries

In this section, we introduce the definition of the ANN query on road networks, and then we review the network Voronoi diagram. Table 1 summarizes the symbols used frequently in the rest of this paper.

Table 1. Frequently Used Symbols

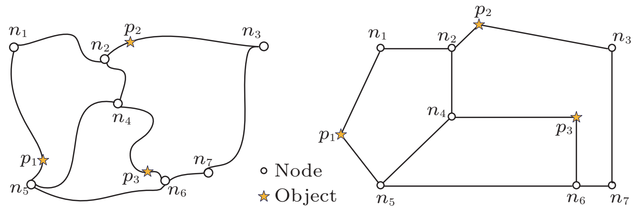
Notation	Description
n	Number of query points
P	Set of POIs
Q	Set of query points, $Q = \{q_1,\dots, q_n\}$
S	ANN candidates set, i.e., set of $p_i \in P$ which is possible to be ANN
S_i	Set of POIs which are expanded by q_i
H	Heap which is used to store all q_i with the *min* or *max* network distance from q_i to its latest kNN as its priority
NN_{q_i}	q_i's first NN, and $NN_{q_i} \in P$
kNN_{q_i}	q_i's k-th NN, and $kNN_{q_i} \in P$
FNN_{q_i}	q_i's furthest NN which is expanded currently
$NDist(p, q)$	Minimum network distance from p to q
$ADist(p_i, Q) =$ $f(NDist(p_i, q_1),$ $\dots, NDist(p_i, q_n))$	Aggregate distance between p_i and any query point in Q

3.1 Problem Definition

A network can be modeled as an undirected weighted graph with its nodes and POIs as the graph vertices and the links connecting two points as the graph edges. The locations of the POIs are constrained on the edges. The distance between two points is the length of the shortest path connecting them rather than the Euclidean distance.

Without loss of generality, suppose a road network $N = \{V, E, P\}$, in which V is the set of nodes, E is the set of edges, and P is the set of POIs. We first construct a new network $N' = \{V', E'\}$ from N as follows. Let $V' = V \cup P$. For any edge in E which does not contain a POI in P inside its interior, we add it to E' with the same weight. For any edge in E that contains one or more POIs, we subdivide them into multiple sub-edges according to the POIs on the edge, and the weights of the resulting sub-edges are proportional.

Fig.2(a) illustrates an example of a realistic network. Edges of the network are usually roadways and interesting objects (p_1, p_2, p_3) located on edges. The interesting object p_i represents a facility (e.g., restaurant, gas station). Fig.2(b) shows its topological graph corresponding to Fig.2(a). The weight on one edge is usually the distance or travelling time between endpoints.



(a) (b)

Fig.2. Example of a real network and its topological graph. (a) Real network. (b) Topological graph.

Definition 1. *Let f be an aggregate function, such as sum, max, and min. Then, given a set of query points Q, a set of POIs P, and an aggregate function f, an aggregate nearest neighbor query returns a POI $p \in P$ such that $ADist(p, Q) \leqslant ADist(p', Q)$, $\forall p' \in P - \{p\}$.*

3.2 Network Voronoi Diagram

In this subsection, we give a brief introduction to the network Voronoi diagram (NVD), as well as some related concepts.

Given a road network with a set of POIs $P = \{p_1, p_2, \ldots, p_m\}$, we divide the network into m regions, each of which contains a subset of edges and is surrounded by some dotted lines. Each region is associated with a POI p_i, which is the nearest neighbor of any query point located in this region. The associated POI p_i is called the *generator*, and the corresponding region is named as the network Voronoi polygon (NVP) of p_i, denoted as NVP(p_i). As shown in Fig.3, NVP(p_1) contains the edges: (n_1, p_1), (n_1, b_1), (n_5, p_1), (n_5, b_5), (n_5, b_2), and is surrounded by the lines: (b_1, b_2), (b_2, v), (v, b_5).

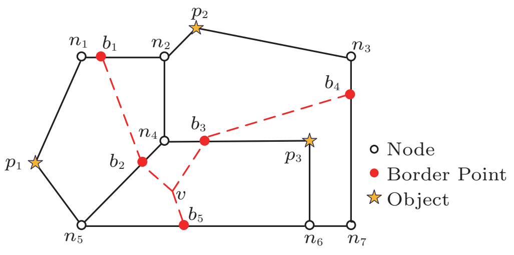
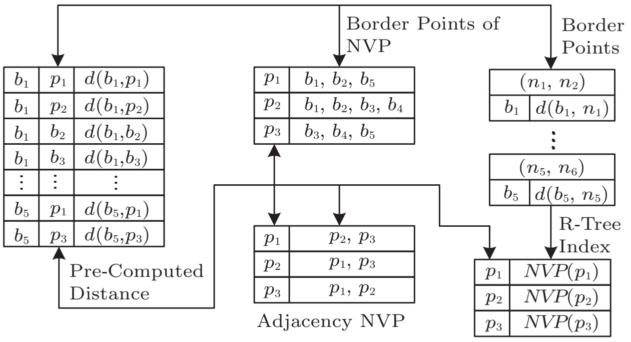


Fig.3. Example of a network Voronoi diagram.

Definition 2. *Given a road network with a set of POIs $P = \{p_1, p_2, \ldots, p_m\}$, the network Voronoi diagram is defined as:*

$$NVD(P) = \{NVP(p_1), NVP(p_2), \ldots, NVP(p_m)\}.$$

Fig.3 shows the network Voronoi diagram NVD(P) for the given road network, where $P = \{p_1, p_2, p_3\}$. Another important concept is the border points, which are the points lying on the edges connecting two adjacent NVPs. By the property of NVP, we can find that a border point is actually the intermediate point of a path between the generators of two adjacent NVPs. For example, in Fig.3, the location of b_1 is the intermediate position of the path (p_1-n_1-n_2-p_2) from p_1 to p_2. By computing the border points, the generation of NVD is facilitated. Intuitively, the generation of a Voronoi diagram in Euclidean spaces could be accomplished by making perpendicular bisector of two generator points. Nevertheless, how to generate an NVD based on a network graph? Hakimi *et al.*[33] proposed an algorithm to solve this problem. In this paper, we use a similar storage schema to store information of an NVD as in [2]. Fig.4 illustrates it for the NVD of Fig.3.



b_1	p_1	$d(b_1,p_1)$
b_1	p_2	$d(b_1,p_2)$
b_1	b_2	$d(b_1,b_2)$
b_1	b_3	$d(b_1,b_3)$
⋮	⋮	⋮
b_5	p_1	$d(b_5,p_1)$
b_5	p_3	$d(b_5,p_3)$

p_1	b_1, b_2, b_5
p_2	b_1, b_2, b_3, b_4
p_3	b_3, b_4, b_5

p_1	p_2, p_3
p_2	p_1, p_3
p_3	p_1, p_2

(n_1, n_2)	
b_1	$d(b_1, n_1)$
⋮	
(n_5, n_6)	
b_5	$d(b_5, n_5)$

p_1	$NVP(p_1)$
p_2	$NVP(p_2)$
p_3	$NVP(p_3)$

Fig.4. Example of NVD storage schema. $d(x, y)$: the distance between x and y.

4 ANN Query

A naive method to process an ANN query is to traverse the whole network and compute all aggregate distances for each POI, and then we can find the ANN with the minimum aggregate distance. However, it is obviously infeasible due to the huge cost of traversing the whole network for every query.

In this section, we present our approach based on the NVD to tackle ANN queries in road networks. Our approach consists of two phases. 1) Searching phase. In this phase, we continuously compute the next NN from each query point to obtain a candidate set of POIs until we find a common POI for all query points. Here, the next NN of a query point q refers to the NN after the last found NN of q, i.e., supposed that the k-th NN is the furthest NN that has been found for q currently, then the next NN refers to the $(k+1)$-th NN of

q. In addition, the process of continuously computing the next NN for one query point is called expanding this query point in the rest of the paper. And the expanding order decides which query point we would like to expand. 2) Pruning phase. After we obtain a candidate set, we continuously expand the search space by computing the next NN for a certain query point, and prune the unqualified candidate POIs that cannot be ANNs.

Note that when the aggregate function is *min*, the processing of an ANN query is straightforward. We only need to compute the NN of each query point, and fetch the POI which has the smallest distance to its query point as the ANN result. Thus, we do not discuss the *min* function for space saving.

4.1 Searching Phase

The objective of searching phase is to obtain an ANN candidates set S, which includes the final query result. We assume that there are n query points in Q. The searching phase includes such steps as follows. To begin with, we search the first NN NN_{q_i} for each $q_i \in Q$ and add NN_{q_i} to the list S_i respectively. Then we check whether there is an intersection among S_1, $\ldots$, S_n. If not, we continue to expand the search space for one query point q_i.

When retrieving the kNN for each q_i, in this paper, we apply VN^3 algorithm[2] based on the NVD generated in advance. Based on the property of NVD, the next NN of a query point is located in one of the adjacent NVPs of the current NN. After kNN_{q_i} is retrieved, we insert it to the tail of the list S_i, which is actually an ordered list of q_i's NNs, i.e., $S_i = \{NN_{q_i}, 2NN_{q_i}, \ldots, kNN_{q_i}\}$. This phase stops until there is an intersection among $S_1, \ldots, S_n$, i.e., $S_1 \cap S_2 \cap \ldots \cap S_n \neq \emptyset$. This means that we have found out the first common $p \in P$ which is expanded by all q_i. Hence, p is the current best candidate of ANN, and we can compute aggregate distance $ADist(p, Q)$ denoted by $dist_{\rm agg}$.

Lemma 1. *When we have found out the first common $p \in P$ which is expanded by all q_i ($S_1 \cap S_2 \cap \ldots \cap S_n = \{p\}$), let $S = S_1 \cup S_2 \cup \ldots \cup S_n$, then the ANN must be in S.*

Proof. Suppose p' is the ANN and $p' \notin S$. As $p' \notin S$, $p' \notin S_i$. Since S_i is an ordered list of q_i's NNs, we can conclude that for each $q_i \in Q$, $NDist(p', q_i) \geqslant NDist(p, q_i)$. This conclusion is illustrated explicitly in Fig.5. Then we can derive that $ADist(p', Q) > ADist(p, Q)$. This indicates that p' is not the ANN, which contradicts with our assumption. Therefore the ANN must be in S. □

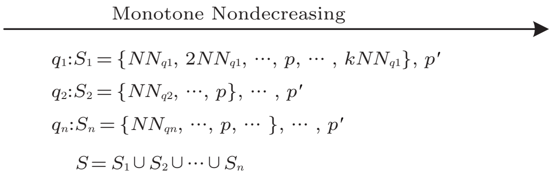

Monotone Nondecreasing
q_1:$S_1 = \{NN_{q1}, 2NN_{q1}, \cdots, p, \cdots, kNN_{q1}\}$, p'
q_2:$S_2 = \{NN_{q2}, \cdots, p\}, \cdots, p'$
q_n:$S_n = \{NN_{qn}, \cdots, p, \cdots\}, \cdots, p'$
$S = S_1 \cup S_2 \cup \cdots \cup S_n$

Fig.5. Illustration of searching phase.

Algorithm 1 details the process of the searching phase (lines 1∼11). For each query point q_i, the algorithm finds the first nearest neighbor and puts the NN into the query point's expanded list S_i. In addition, each query point is inserted into a *minheap* (H) with the weight $NDist(NN_{q_i}, q_i)$ (lines 3∼6). Then we compute the next nearest neighbor (e.g., p') of the top element in H and add it into the expanded set $S_{\rm current}$ of the corresponding query point $q_{\rm current}$. At the same time, we need to update the weight of $q_{\rm current}$ in H as $NDist(p', q_{\rm current})$ (lines 7∼11). The algorithm repeats the above procedures until there is a POI p which is expanded by all query points. After the common POI is found, the aggregate distance $ADist(p, Q)$ is computed (line 12).

```
Algorithm 1. Searching Phase (SP)
Input: $NVD$, $P$, $Q$, $f$
Output: $S$, $S_i$, $H$, $dist_{\rm agg}$
 1: $H$ = new heap; $S = \emptyset$; $S_i = \emptyset$; $dist_{\rm agg} = \infty$; $i = 1$;
 2: Create a new heap entry $E$;
 3: while $i \leqslant n$ do
 4:     Compute $q_i$'s first nearest neighbor $NN_{q_i}$;
 5:     $S_i = \{NN_{q_i}\}$; $E.qid = q_i$; $E.dist = NDist(NN_{q_i}, q_i)$;
 6:     $Insert(H, E)$; $i = i + 1$;
 7: while $\cap_{i=1}^{n} S_i = \emptyset$ do
 8:     $E = removeHead(H)$;
 9:     $q_{\rm current} = E.qid$; compute $q_{\rm current}$'s next NN $p'$;
10:     $S_{\rm current} = S_{\rm current} \cup \{p'\}$; $E.dist = NDist(p', q_{\rm current})$;
11:     $Insert(H, E)$;
12: return $S = \cup_{i=1}^{n} S_i$; $\{p\} = \cap_{i=1}^{n} S_i$; $dist_{\rm agg} = ADist(p,$
    $Q)$;
```

Example 2. Consider the network of Fig.6 and assume that we want to find the ANN for query points q_1, q_2, and q_3. Algorithm 1 finds the first nearest neighbor of query points q_1, q_2 and q_3. Their expanded sets and the heap are initialized to $S_1 = \{p_1\}$, $S_2 = \{p_2\}$, $S_3 = \{p_3\}$, and $H = \{(q_1, 1), (q_2, 1), (q_3, 1)\}$. Suppose the heap here is a *min* heap, then the query points continue to expand until there is a POI p which is accessed by all query points. At this time, $S_1 = \{p_1, p_9\}$, $S_2 = \{p_2, p_9, p_1\}$, $S_3 = \{p_3, p_7, p_9\}$, $S = \{p_1, p_2, p_3, p_7, p_9\}$, and $H = \{(q_1, 3), (q_3, 4), (q_2, 3)\}$. Here, p_9 is the common POI we want to find. When the aggregate function is *sum*, $dist_{\rm agg} = 9$. When the aggregate function is *max*, $dist_{\rm agg} = 4$.

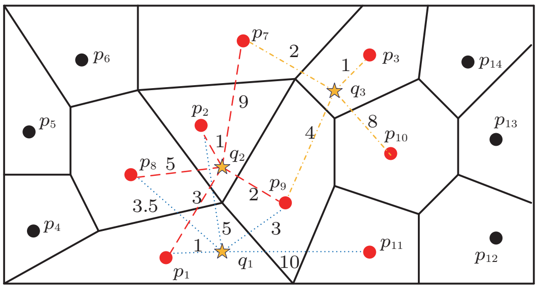


Fig.6. ANN query example.

Notice that Fig.6 only shows the outline of network Voronoi diagram and omits the nodes and edges inside NVPs. Thus, the paths between the query points and the POIs cannot be seen directly from the diagram. But we connect the query points with the POIs using different dashed lines and mark the network distance between them on each line.

4.2 Pruning Phase

The target of pruning phase is to prune unqualified POIs that cannot be the ANN from the candidate set S. If there is only one POI in S, then that POI is indeed the ANN we want. In the sequel, we discuss the pruning strategies for *sum* and *max* functions respectively.

4.2.1 Pruning Phase for sum Function

Before pruning objects from S, we select a query point according to a certain expanding order of query points to retrieve the next NN (p'), and then compute $dist = \sum_{i=1}^{n} NDist(x_i, q_i)$, where

$$x_i = \begin{cases} p', & \text{if } \ p' \in S_i, \\ NN_{q_i}, & \text{if } \ p' \notin S_i. \end{cases}$$

If $dist > dist_{\rm agg}$, we calculate the set S'_i for each q_i.

$$S'_i = \begin{cases} \{o | o \in S_i, NDist(o, q_i) < \\ \quad NDist(p', q_i)\}, & \text{if } \ p' \in S_i, \\ \emptyset, & \text{if } \ p' \notin S_i. \end{cases}$$

Then let $S' = S'_1 \cup \ldots \cup S'_n$. For $\forall p \in S \wedge p \notin S'$, object p can be pruned from S, as stated in Lemma 2.

Lemma 2. *Let p' be the next nearest neighbor of one query point, and $dist = \sum_{i=1}^{n} NDist(x_i, q_i)$. If $dist > dist_{\rm agg}$ and $\forall p \in S \wedge p \notin S'$, object p can be pruned from S.*

Proof. For each i where $p' \in S_i$, as $p \notin S'$, $p \notin S'_i$, and we can get $NDist(p, q_i) \geqslant NDist(p', q_i)$ according to the definition of S'_i. For each i where $p' \notin S_i$, we have $NDist(p, q_i) \geqslant NDist(NN_{q_i}, q_i)$. Since $dist = \sum_{i=1 \wedge p' \in S_i}^{n} NDist(p', q_i) + \sum_{i=1 \wedge p' \notin S_i}^{n} NDist(NN_{q_i}, q_i)$, we have $ADist(p, Q) = \sum_{i=1}^{n} NDist(p, q_i) \geqslant dist$. $ADist(p, Q) > dist_{\rm agg}$ due to $dist > dist_{\rm agg}$. Hence p cannot be the ANN, and can be discarded. □

If $dist < dist_{\rm agg}$, we only need to put q_i's next nearest neighbor p' into its expanded set S_i, and then determine whether p' is expanded by all query points or not. If so, we update $dist_{\rm agg}$ by the value of $ADist(p', Q)$ and change the current candidate of ANN to p'. Then we proceed in this way until there is only one POI in S.

Next we analyze which POIs can be pruned from S in terms of the pruning strategy. Fig.7 shows an example of the pruning phase. As shown in Fig.7, we have searched q_1's next nearest neighbor p', and then we compute the value of *dist* according to its definition in the pruning strategy for *sum* function. Here suppose p is the current candidate of ANN, which has been expanded by all query points. In general, there are three cases for the relationship of expanding order between p and p'. First, p' is expanded prior to p (such as objects in S_2). Second, p' is expanded after p (such as objects in S_j). Third, some query points have not expanded to p' (such as q_n), i.e., p' does not belong to S_n. According to the definition of the set S'_i and the set S' mentioned earlier, POIs pruning can be triggered in the first or the third case. For the first case, the POIs between p' and p (such as p'') can be pruned when p'' does not belong to S'. For the third case, the POIs which are expanded prior to p can be pruned when they do not belong to S'.

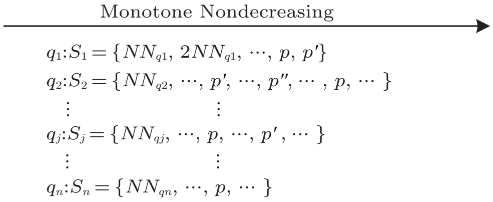

Monotone Nondecreasing
q_1:$S_1 = \{NN_{q1}, 2NN_{q1}, \cdots, p, p'\}$
q_2:$S_2 = \{NN_{q2}, \cdots, p', \cdots, p'', \cdots, p, \cdots\}$
$\vdots$ $\vdots$
q_j:$S_j = \{NN_{qj}, \cdots, p, \cdots, p', \cdots\}$
$\vdots$ $\vdots$
q_n:$S_n = \{NN_{qn}, \cdots, p, \cdots\}$

Fig.7. Illustration of the pruning phase.

Algorithm 2 shows the pseudo-code of our pruning algorithm for *sum* function. In Algorithm 2, we use a heap to store query points. In every round of the pruning phase, we first retrieve the head of the heap H and get its next nearest neighbor. Then we apply our pruning strategy to speed up the ANN query processing. The heap may be *min* heap or *max* heap, depending on the expanding order of query points, which is to be discussed in Subsection 4.4.

Algorithm 2. Pruning for *sum* Function (PSF)

```
Input: $NVD$, $P$, $Q$, $S$, $S_i$, $H$, $dist_{\rm agg}$, $f$
Output: $ANN$, $dist_{\rm agg}$
1:  while $|S| > 1$ do
2:      $E = removeHead(H)$;
3:      $q_{\rm current} = E.qid$; compute $q_{\rm current}$'s next NN $p'$;
4:      $dist = \sum_{i=1 \wedge p' \in S_i}^{n} NDist(p', q_i) +$
        $\sum_{i=1 \wedge p' \notin S_i}^{n} NDist(NN_{q_i}, q_i)$;
5:      if $dist > dist_{\rm agg}$ then
6:          Compute $S'_1, S'_2, \ldots, S'_n$;
7:          $S = S \cap (\cup_{i=1}^{n} S'_i)$; $S_{\rm current} = S_{\rm current} + \{p'\}$;
            //$S_{\rm current}$ is the expanding set of $q_{\rm current}$
8:      else
9:          $S_{\rm current} = S_{\rm current} + \{p'\}$;
10:         if $p' \in \cap_{i=1}^{n} S_i$ then
11:             $S = S - \{p\}$; $p = p'$; $dist_{\rm agg} = ADist(p, Q)$;
12:     $E.qid = q_{\rm current}$; $E.dist = NDist(p', q_{\rm current})$;
13:     $Insert(H, E)$;
14: return $ANN$ (the last POI in $S$) and $dist_{\rm agg}$;
```

As shown in Algorithm 2, lines 1∼13 are the pruning phase which is the core of the algorithm. Line 4 computes the value of *dist*. If the next nearest neighbor p' of top element in H belongs to some expanded set S_i, we use the actual distance value from p' to each query point q_i to compute the value of $NDist(p', q_i)$; otherwise we use the distance from query point q_i to its first nearest neighbor, i.e., $NDist(NN_{q_i}, q_i)$. The value of *dist* computed in this way is the lower bound of the aggregate distance from p' to all query points. If $dist > dist_{\rm agg}$, some POIs that cannot be the ANN can be pruned by intersecting the sets S and the union set of S'_i (line 7). If $dist \leqslant dist_{\rm agg}$ and p' is also a common POI, line 11 updates the current candidate ANN and $dist_{\rm agg}$. Lines 12∼13 update the weight of $q_{\rm current}$ in H.

Example 3. Take Fig.6 as an example, and suppose after searching phase, we have got a POI p_9 accessed by all query points. At this time, $dist_{\rm agg} = 9$, $S_1 = \{p_1, p_9\}$, $S_2 = \{p_2, p_9, p_1\}$, $S_3 = \{p_3, p_7, p_9\}$, $S = \{p_1, p_2, p_3, p_7, p_9\}$, and $H = \{(q_1, 3), (q_3, 4), (q_2, 3)\}$. Next the algorithm turns into the pruning phase. We compute q_1's next nearest neighbor p_8 and $dist = NDist(p_8, q_1) + NDist(p_2, q_2) + NDist(p_3, q_3) = 5.5$. As $dist < dist_{\rm agg}$, we put p_8 into S_1 and update q_1's weight in H. Hence, $S_1 = \{p_1, p_9, p_8\}$, and $H = \{(q_2, 3), (q_3, 4), (q_1, 3.5)\}$.

Then we compute q_2's next nearest neighbor p_8 and $dist = NDist(p_8, q_1) + NDist(p_8, q_2) + NDist(p_3, q_3) = 9.5$. Since $dist > dist_{\rm agg}$, we can get $S' = \{p_1, p_9\} \cup \{p_2, p_9, p_1\} = \{p_1, p_2, p_9\}$. $S = S \cap S' = \{p_1, p_2, p_9\}$, i.e., we have pruned the POIs p_3 and p_7. At this time, $S_2 = \{p_2, p_9, p_1, p_8\}$, and $H = \{(q_1, 3.5), (q_2, 5), (q_3, 4)\}$. Then we compute q_1's next nearest neighbor p_2 and $dist = NDist(p_2, q_1) + NDist(p_2, q_2) + NDist(p_3, q_3) = 7$. As $dist < dist_{\rm agg}$, we just put p_2 into S_1 and update q_1's weight in H. Thus, $S_1 = \{p_1, p_9, p_8, p_2\}$, and $H = \{(q_3, 4), (q_2, 5), (q_1, 5)\}$. Then we compute q_3's next nearest neighbor p_{10} and $dist = NDist(p_1, q_1) + NDist(p_2, q_2) + NDist(p_{10}, q_3) = 10$. As $dist > dist_{\rm agg}$, we can get $S' = \{p_3, p_7, p_9\}$, and $S = S \cap S' = \{p_9\}$. Hence, we have pruned the POIs p_1 and p_2, and the remaining p_9 in S is the final query result.

4.2.2 Pruning Phase for max Function

In Subsection 4.2.1, we have discussed the pruning phase when the aggregate function is *sum*. Now in this subsection, we present the pruning phase for the *max* function. Before discussing this, we give the pruning differences between the *sum* and the *max* functions.

If the aggregate function is *max*, its goal is to find a POI which minimizes the maximum distance from any query point to that POI. Hence, in the pruning phase, in order to compute the value of *dist*, we apply the maximum distance from any query point to the POI p' instead of the sum of these distances in the *sum* function. In addition, similar to the pruning phase for the *sum* function, if p' does not belong to some query point q_i's expanded set, we use the distance from query point q_i to its latest expanded neighbor FNN_{q_i} instead of the distance from query point q_i to its first nearest neighbor NN_{q_i}. This can guarantee the value of *dist* as large as possible, and then can improve the performance of pruning.

Pruning Strategy for max Function. Before pruning POIs from S, we select a query point according to a certain expanding order of query points to retrieve the next nearest neighbor (p'), and then compute $dist = \max\{NDist(x_1, q_1), \ldots, NDist(x_n, q_n)\}$, where

$$x_i = \begin{cases} p', & \text{if } p' \in S_i, \\ FNN_{q_i}, & \text{if } p' \notin S_i. \end{cases}$$

We suppose $dist = NDist(x_k,\ q_k)$. If $dist > dist_{\rm agg}$, we calculate the set S' as follows:

$$S' = \begin{cases} \{o\,|o \in S_k, NDist(o, q_k) < \\ \quad NDist(p', q_k)\}, & \text{if } x_k = p', \\ S_k, & \text{if } x_k = FNN_{q_k}. \end{cases}$$

For $\forall p \in S \wedge p \notin S'$, object p can be pruned from S, as stated in Lemma 3.

Lemma 3. *Let p' be the next nearest neighbor of one query point, and $dist = \max\{NDist(x_1,\ q_1),\ldots,\ NDist(x_n,\ q_n)\}$. If $dist > dist_{\rm agg}$ and $\forall p \in S \wedge p \notin S'$, object p can be pruned from S.*

Proof. We suppose $dist = NDist(x_k,\ q_k)$. For $x_k = p'$, as $p \notin S'$, $NDist(p,\ q_k) > NDist(p',\ q_k)$. This is to say $\forall p \notin S'$, $ADist(p,\ Q) \geqslant NDist(p,\ q_k) > NDist(p',\ q_k) = dist$. As $dist > dist_{\rm agg}$, $ADist(p,\ Q) > dist_{\rm agg}$. Hence p cannot be the ANN, and can be discarded. For $x_k = FNN_{q_k}$, the proof is similar. □

Same as the pruning strategy for function *sum*, if $dist < dist_{\rm agg}$, we only need to put q_i's next nearest neighbor p' into its expanded set S_i, then update $dist_{\rm agg}$ by the value of $ADist(p',\ Q)$ and change current candidate of ANN to p' when p' is expanded by all query points. In addition, in the pruning strategy for function *max*, we can compute S' by S_k directly rather than by S'_i of each query point.

Considering this pruning strategy, we could find it has higher pruning performance than the pruning strategy for function *sum*. This can be found from the way to compute the set S'. In Subsection 4.2.1, we have discussed the pruning strategy for the *sum* function which prunes the POIs in the two cases. These POIs are between p' and p for the first case, and are expanded prior to p for the second case. We can prune them if and only if they do not belong to S'. In fact, this condition cannot be satisfied easily because most POIs are very likely to belong to the set S'_i of a certain query point. Then when we let $S' = S'_1 \cup \ldots \cup S'_n$, they are still in the set S', and hence we cannot prune them. However, in the pruning strategy for function *max*, it will prune the POIs which are not in the set S_k or are not in the POIs which are prior to p' (according to the definition of S' in the pruning strategy for function *max*). Hence, the number of POIs in the set S' in the pruning strategy for the *max* function is far less than that in the pruning strategy for the *sum* function. In general, we can discard more POIs in one iteration in the pruning phase of the *max* function.

The pruning phase of the *max* function is detailed in Algorithm 3. Line 4 computes the value of *dist* according to the pruning strategy for function *max*. If $dist \leqslant dist_{\rm agg}$, the processing way is the same as Algorithm 2. If $dist > dist_{\rm agg}$, different from Algorithm 2, we only need to compute the intersection set between S and S', and some POIs that cannot be the ANN can be pruned. Line 11 updates the current candidate ANN and $dist_{\rm agg}$. Lines 12∼13 update the weight of query point $q_{\rm current}$ in H.

```
Algorithm 3. Pruning for $max$ Function (PMF)
Input: $NVD$, $P$, $Q$, $S$, $S_i$, $H$, $dist_{\rm agg}$, $f$
Output: $ANN$, $dist_{\rm agg}$
 1: while $|S| > 1$ do
 2:     $E = removeHead(H)$;
 3:     $q_{\rm current} = E.qid$; compute $q_{\rm current}$'s next NN $p'$;
 4:     $dist = \max\{NDist(x_1,\ q_1),\ldots,\ NDist(x_n,\ q_n)\}$;
 5:     if $dist > dist_{\rm agg}$ then
 6:         Compute $S'$;
 7:         $S = S \cap S'$; $S_{\rm current} = S_{\rm current} + \{p'\}$;
 8:     else
 9:         $S_{\rm current} = S_{\rm current} + \{p'\}$;
10:         if $p' \in \cap_{i=1}^{n} S_i$ then
11:             $S = S - \{p\}$; $p = p'$; $dist_{\rm agg} = ADist(p,\ Q)$;
12:     $E.qid = q_{\rm current}$; $E.dist = NDist(p',\ q_{\rm current})$;
13:     $Insert(H,\ E)$;
14: return $ANN$ (the last POI in $S$) and $dist_{\rm agg}$;
```

Example 4. Also take Fig.6 as an example. Similar to the phase, we find the ANN for the *sum* function, and next we compute q_1's next nearest neighbor p_8 and $dist = \max\{NDist(p_8,\ q_1),\ NDist(p_1,\ q_2),\ NDist(p_9,\ q_3)\} = 4$. As $dist = dist_{\rm agg}$, we only need to put p_8 into S_1 and update q_1's weight in H. Hence, $S_1 = \{p_1,\ p_9,\ p_8\}$, and $H = \{(q_2,\ 3),\ (q_3,\ 4),\ (q_1,\ 3.5)\}$. Then we compute q_2's next nearest neighbor p_8 and $dist = \max\{NDist(p_8,\ q_1),\ NDist(p_8,\ q_2),\ NDist(p_9,\ q_3)\} = 5$. As $dist > dist_{\rm agg}$, we can get $S' = \{p_2,\ p_9,\ p_1\}$. $S = S \cap S' = \{p_1,\ p_2,\ p_9\}$, i.e., we have pruned the POIs p_3 and p_7. At this time, $S_2 = \{p_2,\ p_9,\ p_1,\ p_8\}$, and $H = \{(q_1,\ 3.5),\ (q_3,\ 4),\ (q_2,\ 5)\}$. Then we compute q_1's next nearest neighbor p_2 and $dist = \max\{NDist(p_2,\ q_1),\ NDist(p_8,\ q_2),\ NDist(p_9,\ q_3)\} = 5$. As $dist > dist_{\rm agg}$, we can get $S' = \{p_1,\ p_9,\ p_8\}$, and $S = S \cap S' = \{p_1,\ p_9\}$, i.e., we have pruned the POI p_2. At this time, $S_1 = \{p_1,\ p_9,\ p_8,\ p_2\}$, and $H = \{(q_3,\ 4),\ (q_1,\ 5),\ (q_2,\ 5)\}$. Then we compute q_3's next nearest neighbor p_{10} and $dist = \max\{NDist(p_2,\ q_1),\ NDist(p_8,\ q_2),\ NDist(p_{10},\ q_3)\} = 8$. As $dist > dist_{\rm agg}$, we can get $S' = \{p_3,\ p_7,\ p_9\}$. $S = S \cap S' = \{p_9\}$; hence, we have the POI p_1, and p_9 is the final query result for the *max* function.

4.3 Voronoi-Based ANN Algorithm

As mentioned earlier, our Voronoi-based ANN algorithm (VANN) contains the searching phase and the pruning phase. Algorithm 4 lists the pseudo-code. Line 1 invokes Algorithm 1. Lines 3 and 5 obtain the ANN result for the *sum* function and the *max* function respectively.

```
Algorithm 4. Voronoi-Based ANN Algorithm (VANN)
Input: $NVD$, $P$, $Q$, $f$
Output: $ANN$, $dist_{\rm agg}$
1: $S$, $S_i$, $H$, $dist_{\rm agg}$ = $SP(NVD, P, Q, f)$; // see Algorithm 1
2: if $f$ is sum function then
3:     $ANN$, $dist_{\rm agg}$ = $PSF(NVD, P, Q, S, S_i, H, dist_{\rm agg}, f)$;
4: else if $f$ is max function then
5:     $ANN$, $dist_{\rm agg}$ = $PMF(NVD, P, Q, S, S_i, H, dist_{\rm agg}, f)$;
6: return $ANN$ and $dist_{\rm agg}$;
```

Example 5. As the example depicted in Fig.6, after calling Algorithm 1, we can get $S = \{p_1, p_2, p_3, p_7, p_9\}$, $S_1 = \{p_1, p_9\}$, $S_2 = \{p_2, p_9, p_1\}$, $S_3 = \{p_3, p_7, p_9\}$, $H = \{(q_1, 3), (q_3, 4), (q_2, 3)\}$, and $dist_{\rm agg} = 9$ or 4. If the aggregate function is *sum*, we call Algorithm 2, and the ANN result is p_9 and $dist_{\rm agg} = 9$. When the aggregate function is *max*, we employ Algorithm 3, and the ANN result is p_9 and $dist_{\rm agg} = 4$. The pruning details can be seen in example 3 and example 4, respectively.

The following theorem proves the correctness of our algorithms.

Theorem 1. *For an ANN query, if we do the first step according to Algorithm* 1*, and then apply Algorithm* 2 *for function sum and Algorithm* 3 *for function max, the final ANN result we obtain is correct.*

Proof. According to the proof of Algorithm 1 in Lemma 1, we are sure that the accurate ANN result is in S. Then from the proof of Algorithm 2 in Lemma 2 and that of Algorithm 3 in Lemma 3, we can find it only prunes objects not being the ANN results. Thus according to our algorithms, we obtain a correct result. □

4.4 Analysis

In this subsection, we first discuss the effect of expanding order of query points. Then we analyze its time complexity.

When query points expand their search space by gradually computing their nearest neighbors, there are three classes of expanding orders as follows. The first is that we expand query points in their number order. That is to say, we compute each query point q_i's next nearest neighbor circularly. The next two kinds are corresponding to the type of the heap, i.e., *min* heap or *max* heap.

Number-Based Expanding Strategy. This strategy is to expand the search space in the order of all query points circularly. In fact, there must be a specific optimal expanding order for query points. However, we cannot know this optimal expanding order in advance. Thus this strategy might cause some useless search space expanding, resulting in ineffective pruning performance.

Minimum Distance Expanding First Strategy. This strategy is based on the intuition that the location of the ANN is more likely to be close to the centroid of the query points. Thus, we always expand the query point whose distance to its latest expanded neighbor is minimum. This leads to all query points expanding to the centroid fairly. Hence it can avoid some useless expanding.

Maximum Distance Expanding First Strategy. This strategy gives priority to the expanding of the query point whose distance to its latest expanded neighbor is maximum. In fact, this strategy almost cannot prune any POIs in S. The reason is that it expands one query point all the time until it visits all the POIs. Next it will expand another query point until it visits all the POIs. Thus, in our experimental evaluation, we do not apply it in our approach.

Optimization About Expanding. During the pruning phase, if a query point q_i's expanded set S_i has contained all of the elements in S, we are disinclined to expand this query point. The reason is that if we continue to expand the query point q_i, the pruning strategy for the *sum* function almost cannot prune any more POIs in S. Since even though it meets the pruning condition, and the expanded sets of all query points do not contain query point q_i's next nearest neighbor, at this time, the number of POIs in S' is minimum (according to the definition of S' in the *sum* function). Now S', which is equal to S'_i, is still the subset of S. Therefore it cannot prune any POI in S. For the pruning strategy for the *max* function, its pruning performance is not much better than that by expanding other query points whose expanded set does not include all the elements in S.

Next, we analyze the time complexity of our algorithms. As mentioned before, we employ the NVD-based kNN algorithm[2] to compute the kNN of the query points. This algorithm is similar to the Dijkstra algorithm (see [2] for more details).

Theorem 2. *Let $|B|$ be the number of border points in NVD, and $|D|$ be the record number of border points to border points and border points to its generators. The time complexity of our algorithm is $O(|Q| \times (|B| + |P| + |D|) \times \log(|B| + |P|))$.*

Proof. For a graph with $|V|$ vertices and $|E|$ edges, the complexity of the Dijkstra algorithm is $O(|V| + |E|) \times \log |V|$. Our algorithm uses the Dijkstra algorithm to compute kNNs based on NVD, where $|V| = |B| + |P|$ and $|E| = |D|$. Thus, the time complexity of kNN processing in our algorithm is $O(|B| + |P| + |D|) \times \log(|B| + |P|)$. In the worst case, we will compute all kNNs for each query point; hence, the time complexity of our algorithm is $O(|Q| \times (|B| + |P| + |D|) \times \log(|B| + |P|))$. □

5 Approximate ANN Query

For the scenarios where the number of query points is very large, the execution time of an ANN query will become unbearable. Sometimes it may be unnecessary to find the exact ANN result. Instead, an approximate result is preferred, for the purpose of reducing the processing time significantly.

In fact, for either solving a meeting problem or selecting a location for residents in real life, people are inclined to gather together. Thus, we can consider a group of adjacent persons as one query point. In this way, we can greatly reduce the number of query points in the ANN query, and thus we can reduce the CPU time greatly.

5.1 Partition Algorithm

In this paper, we propose an algorithm to partition the query points. The main idea is that we regard some adjacent query points in the Euclidean space as a Union-Find set and compute the geometric center of these query points, and then we fetch the closest point to the geometric center on road network as a query point. This is because the proximity in the Euclidean space reflects the proximity in the road network to some extent, and the cost of finding adjacent query points in road network is too expensive. Nevertheless, how to partition the query points?

We define $|UFS|$ as the size (i.e., the upper bound number of query points) of a Union-Find set and $N_{\rm UFS}$ as the number of Union-Find sets. We propose some partition strategies by $|UFS|$ and $N_{\rm UFS}$. When $|UFS|$ is the limited parameter, we only need to combine the adjacent query points or their Union-Find sets as long as the number of query points in the Union-Find set is not beyond $|UFS|$ after combination. Thus, the number of Union-Find sets by this partition strategy is not sure. It depends on the distribution of the query points.

When $N_{\rm UFS}$ is the limited parameter, this means the number of Union-Find sets after the partition is sure and equal to $N_{\rm UFS}$. In addition, we can classify it into two kinds of strategies according to partitioning the query points for Union-Find sets averagely or arbitrarily. Averagely, we mean the number of query points in a Union-Find set is fixed as $(|Q|/N_{\rm UFS})$.

For example, suppose there are 100 query points in an ANN query, they must be partitioned to two Union-Find sets ($N_{\rm UFS} = 2$). The averagely partitioning result must be (50, 50), while arbitrarily partitioning result is $\{(x, y)|x + y = 100, 1 \leqslant x, y \leqslant 99\}$.

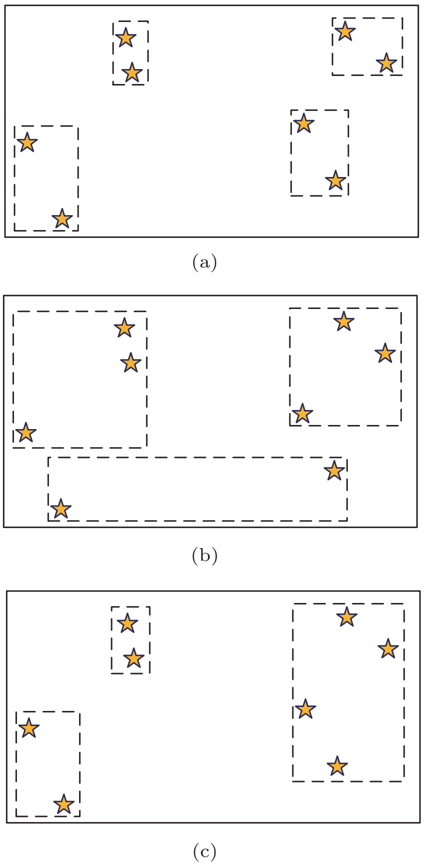
Algorithm 5 details the process for these three kinds of partition strategies. Lines 3∼5 compute the Euclidean distance between any two query points.

Algorithm 5. Partition Algorithm (PA)
Input: NVD, P, Q, $|UFS|$, $N_{\rm UFS}$
Output: Q'

```
1:  $H$ = new heap; create a new heap entry $E$;
2:  Create a Union-Find set $UFS$;
3:  foreach $q_i \in Q$ and $q_j \in Q$ $(i \neq j)$ do
4:      Compute $dist$ Euclidean distance between $q_i$ and $q_j$;
5:      $H \longleftarrow (q_i, q_j, dist)$;
6:  switch the partition strategy do
7:      case partition by $|UFS|$
8:          while there is a query point which is not added to a
            Union-Find set do
9:              $E = removeHead(H)$;
10:             if $|UFS_{E.q_i}| + |UFS_{E.q_j}| \leqslant |UFS|$ then
11:                 $Union(E.q_i, E.q_j)$;
12:     case partition by $N_{\rm UFS}$ averagely
13:         while there is a query point which is not added to a
            Union-Find set do
14:             Fetch a query point $q_i$ which is not added to a
                Union-Find set;
15:             for $i \leftarrow 1$ to $|Q|/N_{\rm UFS}$ do
16:                 Compute $q_i$'s adjacent query point $q_j$ which is not
                    added to a Union-Find set;
17:                 $Union(E.q_i, E.q_j)$;
18:     case partition by $N_{\rm UFS}$ arbitrarily
19:         while the number of Union-Finds is not equal to
            $N_{\rm UFS}$ do
20:             $E = removeHead(H)$;
21:             if $E.q_i$ and $E.q_j$ are not in the same Union-Find
                set then
22:                 $Union(E.q_i, E.q_j)$;
23: foreach equivalence partitioning in $UFS$ do
24:     Compute its average coordinate $\overline{x}$, $\overline{y}$;
25:     Fetch the closest point to $(\overline{x}, \overline{y})$ on road network as the
        query point $q'$;
26:     $Q' \longleftarrow q'$;
27: return $Q'$;
```

Lines 6∼11 partition the query points when $|UFS|$ is the limited parameter. Lines 12∼17 detail the process of averagely partitioning with parameter $N_{\rm UFS}$. Similarly, lines 18∼22 detail it with arbitrarily partitioning. Lines 23∼27 fetch a closest point on the road network as the new query point.

Fig.8 shows the partition results for three partition strategies. There are eight query points. If $|UFS| = 3$, the partition result may be like that shown in Fig.8(a). After the partition, it has four Union-Find sets. Fig.8(b) and Fig.8(c) show the partitioning results for $N_{\rm UFS} = 3$ with the average and the arbitrary partitioning respectively. Clearly, $N_{\rm UFS}$ with average partitioning may group two remote query points into a set, resulting in a set with poor proximity.


(a)

(b)

(c)

Fig.8. Three partition strategies. (a) $|UFS| = 3$. (b) $N_{\rm UFS} = 3$, averagely. (c) $N_{\rm UFS} = 3$, arbitrarily.

5.2 Partition Strategy and Parameter Selection

The selection of the three partition strategies depends on the real application scenarios. The $|UFS|$ strategy fixes the number of Union-Find sets, i.e., the upper bound of the number of query points is provided. Thus this strategy is preferred for the scenario where the number of query points is very large, and the system/application is sensitive to the response time. Compared with the $|UFS|$ strategy, the $N_{\rm UFS}$ average/arbitrary strategies constrain the size of each Union-Find set. With small $N_{\rm UFS}$, the Euclidean center of each set is closer to the points in the set, and thus the approximate ANN is closer to the exact ANN. In other words, the error bound of the approximate algorithm is guaranteed. Therefore, these two strategies are practical for the applications which require higher accuracy.

5.3 Complexity Analysis

The complexity of Algorithm 5 is $O(|Q|^2)$ and the complexity of exact ANN algorithm is $O(|Q| \times (|B| + |P| + |D|) \times \log(|B| + |P|))$. Thus, the complexity of approximate algorithm is $O(|Q|^2) + O(|Q'| \times (|B| + |P| + |D|) \times \log(|B| + |P|))$. In general, $|Q'| < |Q| \ll (|B|+|D|)$. Thus, the approximate ANN algorithm will greatly improve the performance.

6 Experimental Evaluation

In this section, we experimentally evaluate the efficiency of our proposed algorithms. We conduct extensive experiments for different kinds of query point expanding order and compare the performance with IER and TA[1]. Note that, we do not compare against CE algorithm[1] because, as shown in [1], it has the worst performance.

6.1 Experimental Setup

In our experiment, we use the real road network (San Francisco road map[34]) as datasets. In the original road map, since the weight of each edge is the Euclidean distance between its end-points, we set the weight of each edge to the Euclidean distance multiplied by a random number chosen from the range [1, F]. In this way, parameter F reflects the factor by which the actual weight may deviate from the Euclidean distance.

We uniformly generate interesting data objects on the network edges. The parameter P represents the density of data objects (i.e., the number of data objects over the number of edges in the network). The query points in Q are generated randomly on edges that are connected sub-network covering A percent of the network edges. For example, by default, each query has

10 query points randomly generated in $A = 4$ percent of the network. In addition, by default, the density of data objects P is equal to 0.04 and the parameter F is equal to 1. For each experiment setting, we average the results of the algorithm over 10 queries. The experiments are run on a PC with a Pentium® CPU of 2.0 GHz.

6.2 VANN Algorithm Performance

In this subsection, we evaluate the performance of the VANN algorithm based on the following three criteria: 1) query processing time; 2) page accesses during query processing; 3) the number of nodes visited when expanding on the road network (i.e., A* or Dijkstra algorithm). Notice that IER and TA apply A* algorithm to expand on the road network, and hence we compute the number of nodes visited according to the nodes accessed on the road network during the algorithms. However, when computing kNN based on Voronoi on road network, we only need to expand the cell of Voronoi, and thus we compute the number of the border points and objects as the number of nodes visited for our algorithms. In addition, we use the average value of iterations for query points when computing kNN to evaluate our algorithms. We compute it by $(Total\ of\ Iterations)/|Q|$. The average value of iterations reflects the performance of our pruning strategy. In our experiments, we show the number of nodes visited by dividing 1 000, and use Min.*sum* to represent our algorithm with minimum distance expanding first strategy (Min) for *sum* function (the similar meaning for Min.*max*, Num.*sum*, Num.*max*, IER.*sum*, IER.*max*, TA.*sum* and TA.*max*).

6.2.1 *Effect of A*

As shown in Fig.9, the costs of four algorithms increase with A since our algorithms must expand more interesting data objects, while IER and TA must expand a larger range of the network, incurring that the CPU time, page accesses and the number of nodes visited grow as the parameter A becomes larger.

However, as depicted in Fig.9(a), our algorithm with the Min strategy is better than IER in CPU time. The reason is that although IER uses R-tree index, it still needs to explore the network and compute the aggregate distance, while our algorithm does not need to expand the network and only searches the look-up tables. However, the CPU time of our algorithm with the Num strategy is larger because there are many invalid computations that do not prune data objects in S.

Fig.9(b) illustrates the performance of page accesses. Our algorithm outperforms IER. The reason is that IER needs to explore the network and retrieve only a small number of edges at each step, whereas our algorithms retrieve the pre-computed values in only one step. Thus the number of page accesses is smaller than that of IER. The performance of nodes visited shown in Fig.9(c) is similar to that of page accesses in Fig.9(b).

Fig.9(d) plots the average value of iterations for our algorithms. We can observe that the performance of the Min strategy is much better than that of the Num strategy. This is because the Num strategy has computed many invalid kNNs. As the parameter A becomes larger, the expanding range of each query point is larger, and hence more kNNs must be computed for each query point to obtain ANN result.

Fig.10 shows the performance of ANN queries for the same criteria when the function is *max*. The performance of our algorithm for function *max* is better than that of the algorithm for function *sum* with both Min strategy and Num strategy. The reason is that the efficiency of the pruning strategy for function *max* is higher than that of the pruning strategy for function *sum*. This can be seen from Fig.10(d). The reasons

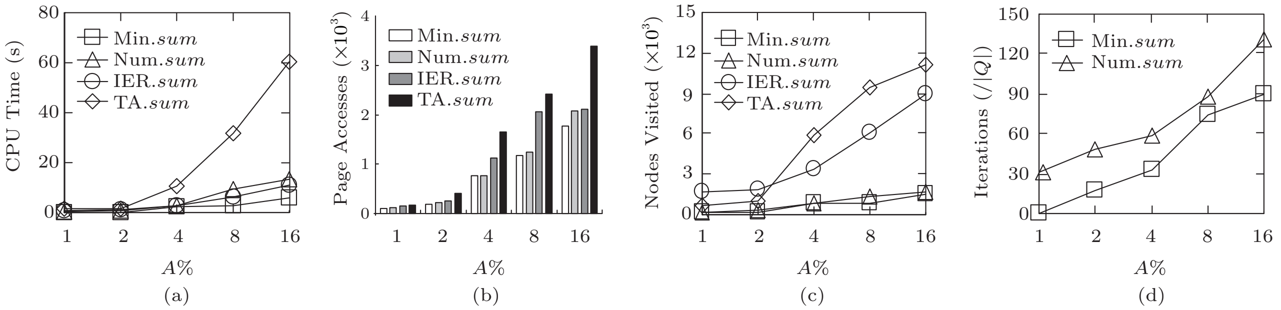


Fig.9. Cost as a function of query area A for the *sum* function. (a) CPU time. (b) Page accesses. (c) Nodes visited. (d) Average iterations.

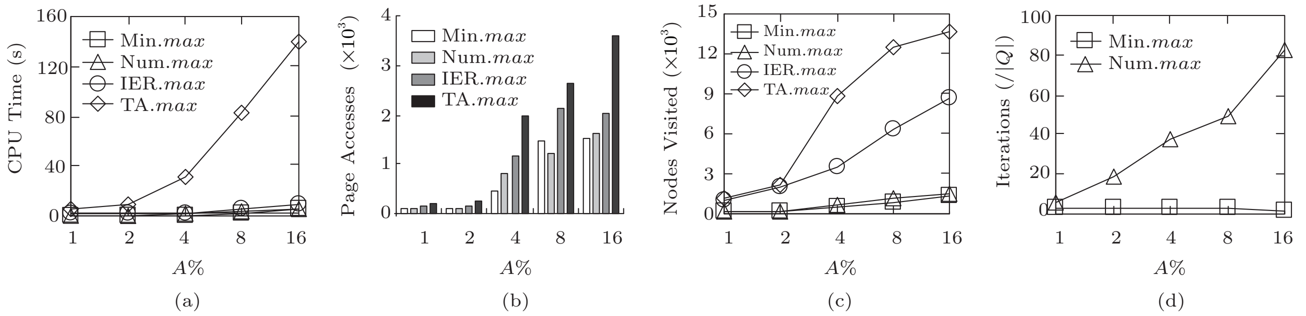


Fig.10. Cost as a function of query area A for the *max* function. (a) CPU time. (b) Page accesses. (c) Nodes visited. (d) Average iterations.

why our algorithm with Min strategy outperforms IER and TA are similar to the explanation of algorithm for function *sum*. Note that our algorithm with Num strategy is also better than IER and TA. In addition, the performance of IER for function *max* is better than that for function *sum*, while the performance of TA for function *max* is worse than that for function *sum*. This is because the termination condition of IER for function *max* is more easily reached than that for function *sum*, while the termination condition of TA for function *max* is much harder than that for function *sum*.

6.2.2 *Effect of P*

In the next experiment, we compare the four algorithms on the density of objects. From Fig.11, we can observe that the costs of our algorithms increase with P since there are more NVPs, and thus our algorithms must expand more interesting data objects, while that has little effect on IER and TA. When P is equal to 0.08, our algorithms become worse than IER and TA when $F = 1$. In fact, the density of objects is mostly less than 0.08 in real life, for example, in [2], the maximal density of objects is 0.058 (Restaurant).

In Fig.12, we compare the performance of our algorithms with that of IER and TA for the *max* function. Different from the *sum* function, the performance of our algorithms for the *max* function is better than that of IER and TA. The reason is that our algorithms for the *max* function are much more efficient than that for the *sum* function as stated above. In Fig.11(d) and

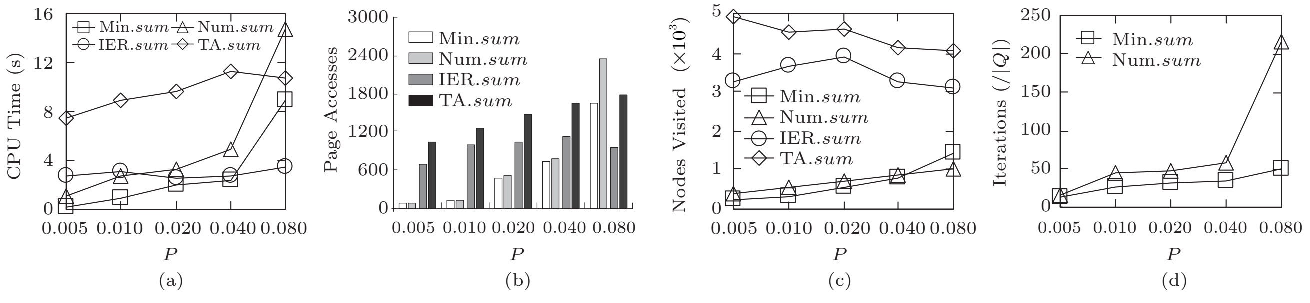


Fig.11. Cost as density of objects for the *sum* function. (a) CPU time. (b) Page accesses. (c) Nodes visited. (d) Average iterations.

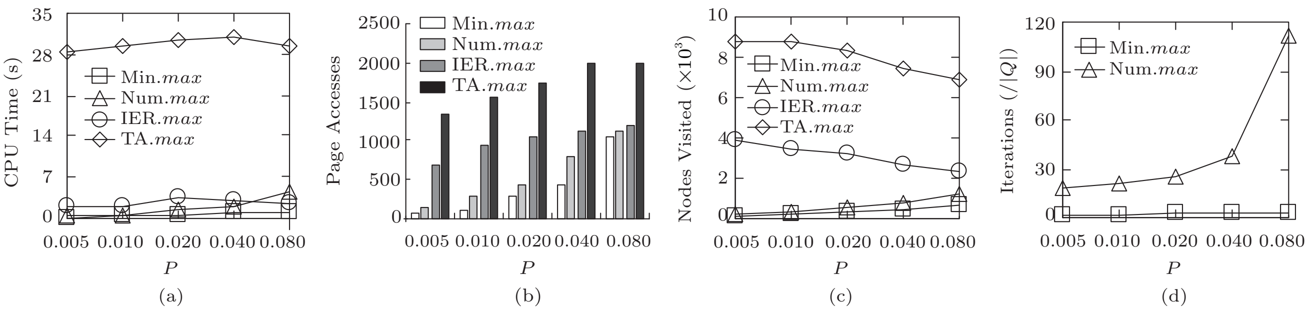


Fig.12. Cost as density of objects for the *max* function. (a) CPU time. (b) Page accesses. (c) Nodes visited. (d) Average iterations.

Fig.12(d), we can observe that the Num strategy is very sensible to the density of objects, and its average value of iterations is larger as the value of P becomes larger. This is because more kNNs must be expanded for each query point when we can prune the objects in set S, while this is not clear for the Min strategy.

6.2.3 *Effect of* $|Q|$

The next experimental factor is the number of query points. In Fig.13, we can find that the CPU time increases with $|Q|$ because more query points should be considered for ANN queries. From Fig.14, we can notice that TA is not very appropriate to evaluate ANN queries for the *max* function. In Fig.13(d) and Fig.14(d), we can observe that the average value of iterations is smaller as the number of query points becomes larger. This is because we compute it by $(Total\ of\ Iterations)/|Q|$.

6.2.4 *Effect of* F

In the next experiment, we compare the algorithms after distorting the edge weights by different factor F. As shown in Fig.15 and Fig.16, we can see that our algorithms and TA are not affected by this parameter. On the other hand, the performance of IER degrades with F. The reason is that IER uses Euclidean distance as a lower bound of the network distance. When the weight of each edge deviates from the Euclidean distance largely, it may do many unnecessary computations.

6.3 Performance of Approximate ANN Algorithm

In this subsection, we evaluate the performance of approximate ANN algorithm. In addition to the query processing time and page accesses, the value of deviation is also important. We compute it by $(dist_{\rm agg\_approximate} - dist_{\rm agg})/dist_{\rm agg} \times 100\%$. This criterion reflects the extent of deviation between approximate result and exact result. The factors for approximate experiment are $|UFS|$ and $N_{\rm UFS}$.

6.3.1 *Effect of* $|UFS|$

Fig.17 shows the effect of the parameter $|UFS|$ on the performance of the algorithm for the *sum* function and the *max* function with the Min strategy. In this experiment, the number of query points is 200. As shown in Fig.17(a), we can observe that there is hardly any deviation after optimization for function *sum* when the upper bound number of query points is below 8. The

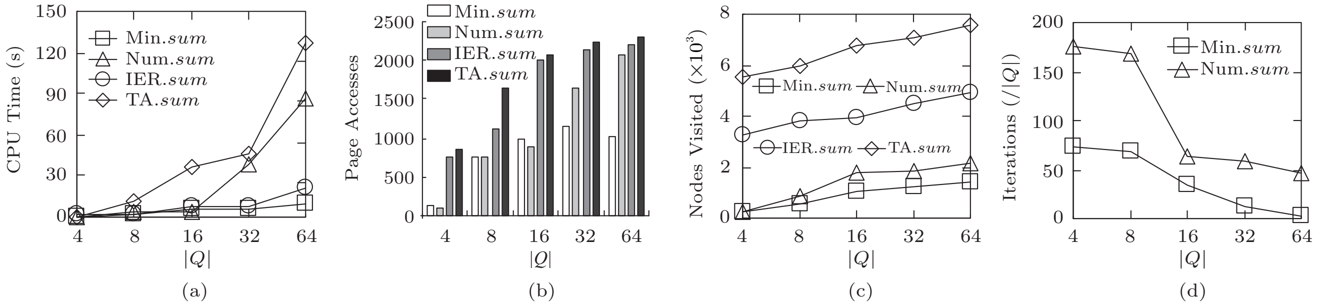


Fig.13. Cost as the number of query points for the *sum* function. (a) CPU time. (b) Page accesses. (c) Nodes visited. (d) Average iterations.

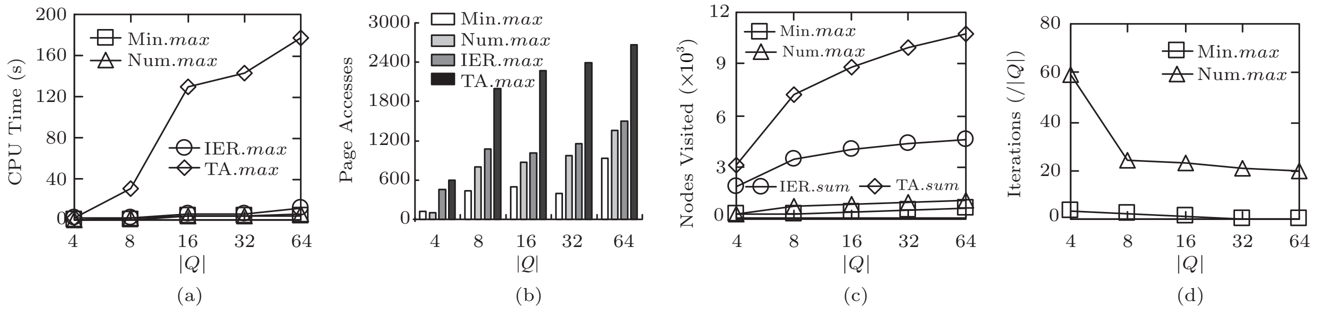


Fig.14. Cost as the number of query points for the *max* function. (a) CPU time. (b) Page accesses. (c) Nodes visited. (d) Average iterations.

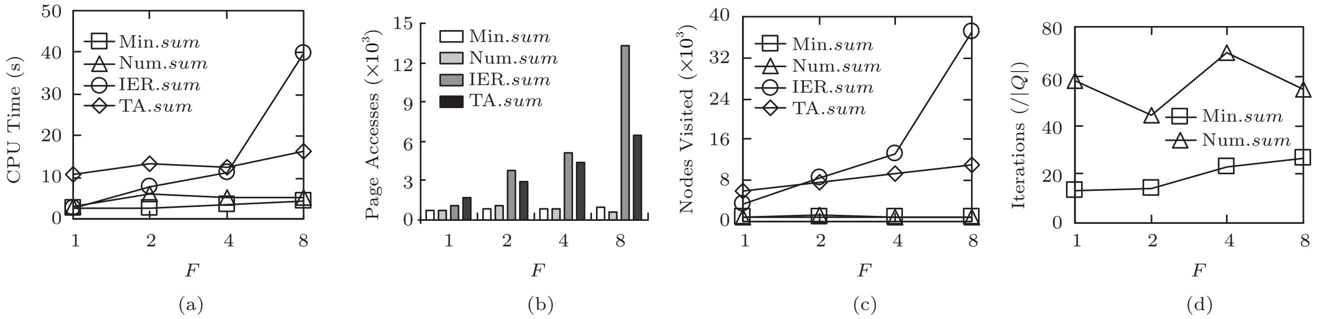


Fig.15. Effect of F for the *sum* function. (a) CPU time. (b) Page accesses. (c) Nodes visited. (d) Average iterations.

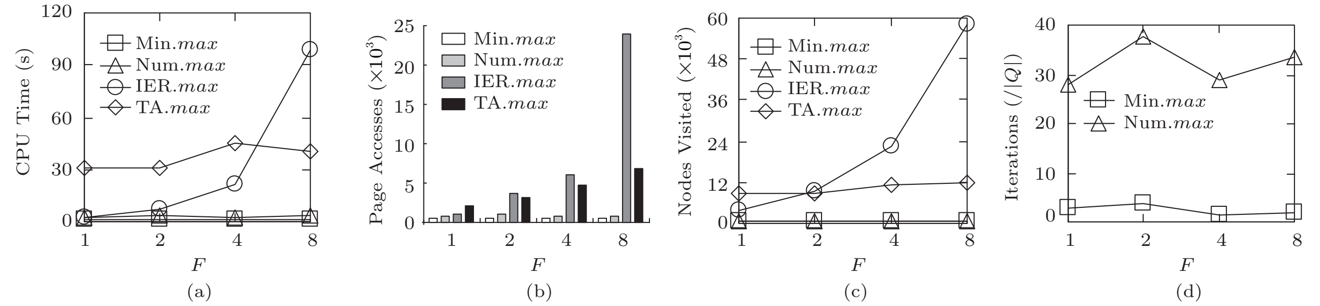


Fig.16. Effect of F for the *max* function. (a) CPU time. (b) Page accesses. (c) Nodes visited. (d) Average iterations.

deviation for function *max* is always higher than that for function *sum*. When $|UFS| = 0$, it means that we apply origin query points to evaluate ANN queries, and thus the value of deviation is 0. In Fig.17(b), the CPU time largely decreases as $|UFS|$ grows. Similar to the experimental result mentioned above, the CPU time of our algorithm for function *sum* is highly larger than that for function *min*. The performance comparison of the page accesses in Fig.17(c) is similar to that of CPU time.

6.3.2 Effect of $N_{\rm UFS}$

Fig.18 shows the performance of our algorithms by the effect of the parameter $N_{\rm UFS}$. In this experiment, we averagely partition query points for function *sum* and arbitrarily partition query points for function *max*. We can easily find that the experimental results by $N_{\rm UFS}$ are opposite to them by $|UFS|$. As shown in Fig.18(a), the deviation is very large when the aggregate function is *sum*. This is because it may damage the local proximity when we averagely partition the query points to a fixed number of Union-Find sets. When $N_{\rm UFS} = 200$, it is equivalent to $|UFS| = 0$, and hence the value of deviation is 0. In Fig.18(b) and Fig.18(c), the CPU time and page accesses increase as $|UFS|$ grows.

According to what discussed above, the performance of the approximate algorithm for ANN queries depends on the partition strategy. The deviation is acceptable if we do not limit the number of query points in a Union-Find set; otherwise the deviation is very large. This can be reflected by comparing the experimental results in Fig.17(a) and Fig.18(a).

To sum up, the performance of our approach with the Min strategy is better than that with the Num strategy in most cases. The efficiency of pruning strategy for function *max* is much higher than that for function *sum*. Our approach with the Min strategy outperforms IER and TA significantly in most cases. In addition, our algorithms are not sensible to the deviation between the edge weight and its Euclidean length.

7 Conclusions

In this paper, we proposed a novel approach to solve the ANN query problem in road networks. Our approach contains two phases: searching phase and pruning phase. The task of searching phase is to obtain a candidate set S by computing the NNs of the query points using NVD. In the pruning phase, effective pruning strategy is applied to prune interesting data objects from S. In addition, we proposed an approximate algorithm for an ANN query with a large number of query

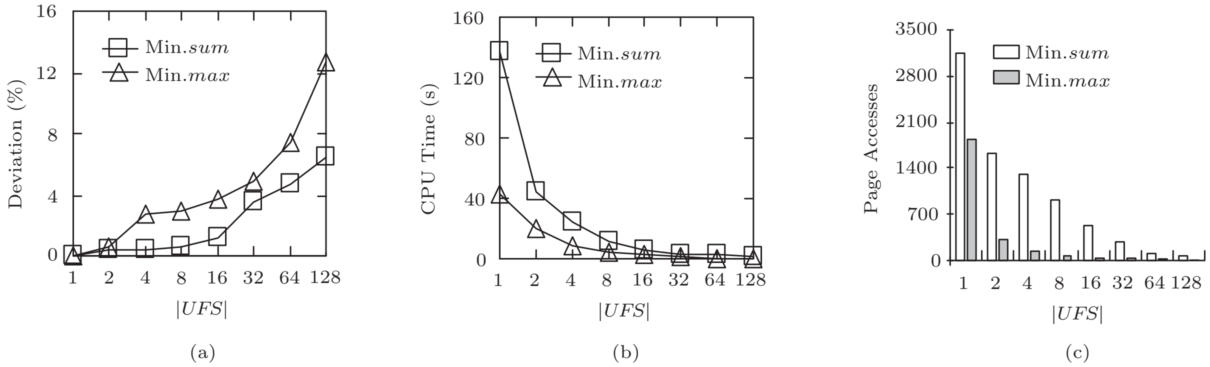


Fig.17. Effect of $|UFS|$. (a) Deviation. (b) CPU time. (c) Page accesses.

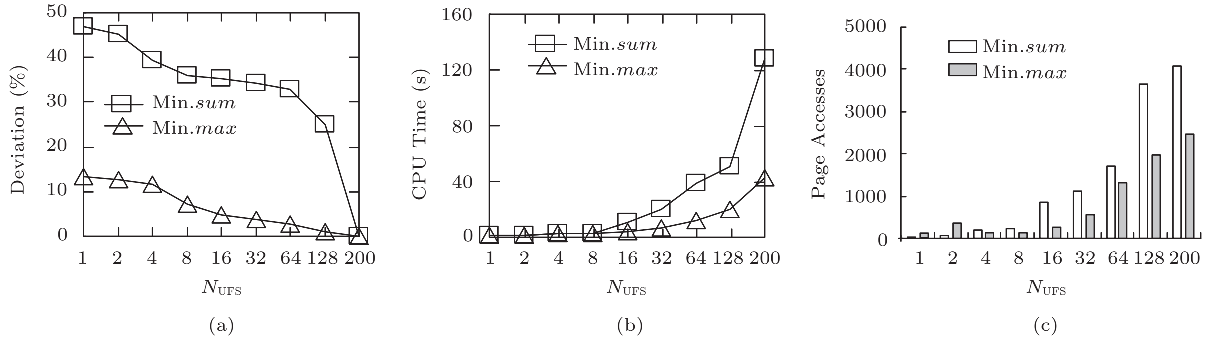


Fig.18. Effect of N_{UFS}. (a) Deviation. (b) CPU time. (c) Page accesses.

points and the accuracy of result is not crucial. Extensive experimental results show that the performance of our proposed algorithms outperforms that of their competitors in most cases. In the future, we plan to investigate some more interesting and challenging problems of ANN queries, such as how to provide a theoretical analysis of the approximate ANN algorithm and give a theoretical bound of the error, and how to process ANN queries efficiently for moving objects.

References

[1] Yiu M L, Mamoulis N, Papadias D. Aggregate nearest neighbor queries in road networks. *IEEE Transactions on Knowledge and Data Engineering*, 2005, 17(6): 820-833.

[2] Kolahdouzan M R, Shahabi C. Voronoi-based k nearest neighbor search for spatial network databases. In *Proc. the 30th VLDB*, Aug.31-Sept.3, 2004, pp.840-851.

[3] Zhu L, Jing Y, Sun W, Mao D, Liu P. Voronoi-based aggregate nearest neighbor query processing in road networks. In *Proc. the 18th ACM SIGSPATIAL GIS*, Nov. 2010, pp.518-521.

[4] Guttman A. R-trees: A dynamic index structure for spatial searching. In *Proc. ACM SIGMOD*, June 1984, pp.47-57.

[5] Roussopoulos N, Kelley S, Vincent F. Nearest neighbor queries. In *Proc. ACM SIGMOD*, May 1995, pp.71-79.

[6] Cheung K L, Fu A W. Enhanced nearest neighbour search on the R-tree. *ACM SIGMOD Record*, 1998, 27(3): 16-21.

[7] Hjaltason G, Samet H. Distance browsing in spatial databases. *ACM Transactions on Database Systems*, 1999, 24(2): 265-318.

[8] Papadopoulos A, Manolopoulos Y. Performance of nearest neighbor queries in R-trees. In *Proc. the 6th ICDT*, Jan. 1997, pp.394-408.

[9] Tao Y, Papadias D, Shen Q. Continuous nearest neighbor search. In *Proc. the 28th VLDB*, Aug. 2002, pp.287-298.

[10] Hu H, Lee D L. Range nearest-neighbor query. *IEEE Transactions on Knowledge and Data Engineering*, 2006, 18(1): 78-91.

[11] Gao Y, Zheng B. Continuous obstructed nearest neighbor queries in spatial databases. In *Proc. ACM SIGMOD*, June 29-July 2, 2009, pp.577-590.

[12] Tao Y, Yiu M L, Mamoulis N. Reverse nearest neighbor search in metric spaces. *IEEE Transactions on Knowledge and Data Engineering*, 2006, 18(9): 1239-1252.

[13] Nutanong S, Tanin E, Zhang R. Visible nearest neighbor querying. In *Proc. the 12th DASFAA*, Apr. 2007, pp.876-883.

[14] Corral A, Manolopoulos Y, Theodoridis Y, Vassilakopoulos M. Closest pair queries in spatial databases. In *Proc. ACM SIGMOD*, May 2000, pp.189-200.

[15] Papadias D, Shen Q, Tao Y, Mouratidis K. Group nearest neighbor queries. In *Proc. the 20th ICDE*, Mar. 30-Apr. 2, 2004, pp.301-312.

[16] Papadias D, Tao Y, Mouratidis K, Hui C K. Aggregate nearest neighbor queries in spatial databases. *ACM Transactions on Database Systems*, 2005, 30(2): 529-576.

[17] Jensen C S, Kolárvr J, Pedersen T B, Timko I. Nearest neighbor queries in road networks. In *Proc. the 11th GIS*, Nov. 2003, pp.1-8.

[18] Papadias D, Zhang J, Mamoulis N, Tao Y. Query processing in spatial network databases. In *Proc. the 29th VLDB*, Sept. 2003, pp.802-813.

[19] Huang X, Jensen C S, Šaltenis S. The islands approach to nearest neighbor querying in spatial networks. In *Proc. the 9th SSTD*, Aug. 2005, pp.73-90.

[20] Kolahdouzan M R, Shahabi C. Continuous *k*-nearest neighbor queries in spatial network databases. In *Proc. the 2nd STDBM*, Aug. 2004, pp.33-40.

[21] Yiu M L, Papadias D, Mamoulis N, Tao Y. Reverse nearest neighbors in large graphs. *IEEE Transactions on Knowledge and Data Engineering*, 2006, 18(4): 540-553.

[22] Sankaranarayanan J, Alborzi H, Samet H. Distance join queries on spatial networks. In *Proc. the 14th ACM GIS*, Nov. 2006, pp.211-218.

[23] Deng K, Zhou X, Shen H T. Multi-source skyline query processing in road networks. In *Proc. the 23rd ICDE*, Apr. 2007, pp.796-805.

[24] Yan D, Zhao Z, Ng W. Efficient algorithms for finding optimal meeting point on road networks. In *Proc. the 37th VLDB*, Aug. 29-Sept. 3, 2011, pp.968–979.

[25] Sharifzadeh M, Shahabi C, Kazemi L. Processing spatial skyline queries in both vector spaces and spatial network databases. *ACM Transactions on Database Systems*, 2009, 34(3): Article No. 14.

[26] Qin L, Yu J X, Ding B, Ishikawa Y. Monitoring aggregate *k*-NN objects in road networks. In *Proc. the 20th SSDBM*, July 2008, pp.168-186.

[27] Elmongui H G, Mokbel M F, Aref W G. Continuous aggregate nearest neighbor queries. *GeoInformatica*, 2013, 17(1): 63-95.

[28] Li J, Yiu M L, Mamoulis N. Efficient notification of meeting points for moving groups via independent safe regions. In *Proc. the 29th ICDE*, Apr. 2013, pp.422-433.

[29] Chen K, Sun W, Tu C, Chen C, Huang Y. Aggregate keyword routing in spatial database. In *Proc. the 20th ACM SIGSPATIAL GIS*, Nov. 2012, pp.430-433.

[30] Hashem T, Kulik L, Zhang R. Privacy preserving group nearest neighbor queries. In *Proc. the 13th EDBT*, Mar. 2010, pp.489-500.

[31] Lian X, Chen L. Probabilistic group nearest neighbor queries in uncertain databases. *IEEE Transactions on Knowledge and Data Engineering*, 2008, 20(6): 809-824.

[32] Liu Z, Wang C, Wang J. Aggregate nearest neighbor queries in uncertain graphs. *World Wide Web*, 2014, 17(1): 161-188.

[33] Hakimi S L, Labbé M, Schmeichel E F. The Voronoi partition of a network and its implications in location theory. *INFORMS Journal on Computing*, 1992, 4(4): 412-417.

[34] Li F, Cheng D, Hadjieleftheriou M, Kollios G, Teng S H. On trip planning queries in spatial databases. In *Proc. the 9th SSTD*, Aug. 2005, pp.273-290.

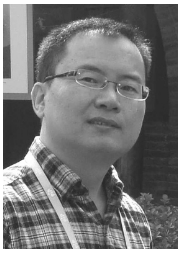
Wei-Wei Sun received his Bachelor's degree in computer science and technology in 1992, and M.S. and Ph.D. degrees in computer software and theory in 1998 and 2002 respectively, all from Fudan University, Shanghai. He is currently an associate professor in the School of Computer Science and the director of Mobile Data Management Laboratory, Fudan University, Shanghai. His research interests include spatial database, wireless data broadcast and geo-social.

Chu-Nan Chen received his Bachelor's degree in information security in 2010 and M.S. degree in computer software and theory in 2013, both from Fudan University, Shanghai. His research interests include wireless data broadcast and spatial data management.

Liang Zhu received his M.S. degree in computer science from Fudan University, Shanghai, in 2012. His research interests include spatial and spatio-temporal databases, *k*-NN neighbor queries on road networks, and uncertain and incomplete databases. He has published papers in journals and conferences including SIGSPATIAL, NDBC.

Yun-Jun Gao received his Ph.D. degree in computer science from Zhejiang University, Hangzhou, in 2008. He is currently an associate professor in the College of Computer Science, Zhejiang University. Prior to joining the faculty, he was a postdoctoral researcher at the Singapore Management University during 2008∼2010, and a visiting scholar or research assistant at the Nanyang Technological University, Simon Fraser University, and City University of Hong Kong respectively. His research interests include spatial and spatio-temporal databases, uncertain and incomplete databases, and spatio-textual data management. He has published papers in journals and conferences including TODS, VLDBJ, TKDE, SIGMOD, VLDB, ICDE, and SIGIR. He is a senior member of CCF, and a member of ACM and IEEE.

Yi-Nan Jing received his Ph.D. degree in computer science from Fudan University, Shanghai, in 2007. He is an assistant professor with the School of Computer Science at Fudan University. He was also a visiting researcher with the Department of Computer Science at the University of Southern California. His current research interests include spatial and temporal data management, geographic information systems, mobile computing, and security and privacy. He is a member of ACM and IEEE.



Qing Li is currently a professor in the Department of Computer Science, City University of Hong Kong where he joined as a faculty member in September 1998. Prior to that, he taught at the Hong Kong Polytechnic University, the Hong Kong University of Science and Technology, and the Australian National University (Canberra, Australia). He is also a guest professor or visiting professor of several universities such as Zhejiang University, Hangzhou, and Chinese Academy of Science, Beijing. His research interests include object modeling, multimedia databases, web services, social media, and big data analytics. He is a fellow of IET, a senior member of IEEE, a member of ACM SIGMOD and IEEE TCDE. He is the chairperson of the Hong Kong Web Society and a steering committee member of ER, DASFAA, ICWL, UMEDIA, and the international WISE Society.