Reverse top- k group nearest neighbor search

CONFERENCE PAPER · JUNE 2013

DOI: 10.1007/978-3-642-38562-9_44

5 AUTHORS, INCLUDING:

Tao Jiang

Jiaxing University

13 PUBLICATIONS **24** CITATIONS

SEE PROFILE

Reverse Top-*k* Group Nearest Neighbor Search

Tao Jiang[1], Yunjun Gao[2], Bin Zhang[1], Qing Liu[2], and Lu Chen[2]

[1] College of Mathematics Physics and Information Engineering, Jiaxing University, Jiaxing 314000, China
{jiangtao_guido,zhangbin_selina}@yahoo.com.cn
[2] College of Computer Science, Zhejiang University, Hangzhou 310027, China
{gaoyj,liuqing1988,chenl}@zju.edu.cn

Abstract. This paper identifies and solves a novel query, namely, *Reverse Top-k Group Nearest Neighbor* (R*k*GNN) query. Given a data set *P*, a query object *q*, and two (user specified) parameters *m* and *k*, an R*k*GNN query finds *k* subsets, which have the least *aggregate distances*, such that each subset contains *m* data objects from *P* and has *q* in its *group nearest neighbor*. We formalize the R*k*GNN query. Then, we propose several algorithms for efficiently processing R*k*GNN queries. Our methods employ some effective *pruning heuristics* to prune away unqualified candidate subsets, utilize the *sorting* and *threshold mechanisms* to shrink the search space, and make use of the advantages of *lazy* and *spatial pruning techniques*. Extensive experiments with both real and synthetic datasets demonstrate the performance of the proposed algorithms in terms of effectiveness and efficiency.

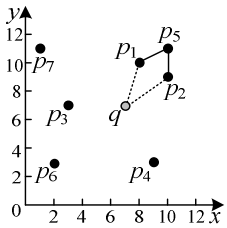
1 Introduction

Given a set of data objects $P = \{p_1, p_2, \ldots, p_n\}$ and a set of query objects $Q = \{q_1, q_2, \ldots, q_m\}$, an *group nearest neighbor* (GNN) query[1-8]returns a data object in dataset *P* with the smallest *sum* of distances to all data objects in *Q*. Figure 1(a) shows an example of group nearest neighbor over a dataset $P = \{p_1, p_2, \ldots, p_7, q\}$ in a 2D space, where the coordinates (*x*-axis and *y*-axis) of each object in *P* are shown in Figure 1(c) and $Q = \{p_1, p_2\}$. Since the object p_5 has the minimum *aggregate distance* to *Q*, that is, $adist(p_5, Q)$, than other objects $p' \in P \backslash Q$ (i.e., the object *q*), it constitutes the GNN of *Q* on dataset *P*.

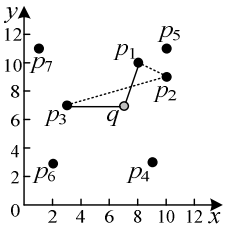
However, GNN query only finish the computation from the user's perspective. In this paper, we study *reverse group nearest neighbor* (RGNN) query which consider the GNN query from company's perspective. Given a dataset *P* and a query object *q*, an RGNN query aims to find multiple subsets from *P* such that each subset *G* has *q* as its GNN, and $|G|=m$ ($m \geq 2$), where *m* is a specified integer. When $m=1$, an RGNN query is reduced to an RNN query. In Figure 1(b), assume that $Q = \{p_1, p_3\}$ is the reference objects. Since $\{p_1, p_3\}$ has *q* as its GNN, it is a result of RGNN of *q*.

Unlike GNN query which only returns a data object as the answer, RGNN query might retrieve multiple subsets as the answers. To enhance user decision-making, we may set up a parameter *k* to limit the number of the subsets using the aggregate distance as the monotonic sorting function. This is a general RGNN query. We call it *reverse top-k group nearest neighbor* (R*k*GNN) query. Consider the example in Figure 1(b), R*k*GNN will return $\{p_1, p_3\}$ and $\{p_2, p_3\}$ as the answers when $k=2$.

J. Wang et al. (Eds.): WAIM 2013, LNCS 7923, pp. 429–439, 2013.



(a) $Q = \{p_1, p_2\}$



(b) $Q = \{p_1, p_3\}$

p_1	(8, 10)
p_2	(10, 9)
p_3	(3, 7)
p_4	(9, 3)
p_5	(10, 11)
p_6	(2, 3)
p_7	(1, 11)
q	(7, 7)

(c) the coordinates of points

Fig. 1. Illustration of group and reverse nearest neighbor

R*k*GNN query has many applications in business analytic and decision support systems. Consider the example: a big company plans to open m branches according to the location of headquarters (denotes as the query object q). The candidate locations of branches can be regarded as data objects in a database. Generally, the manager hopes that the branches have q as their GNN. In other words, these branches have less *sum* distances to q, and there is not another branch which has a smaller sum distances to the branches than q. Another example is: when designing a series of products (i.e., m products) from n alternative products picked up according to a specified prototype q, the corporation generally hopes that the sum distance from q is less for the group of products; meanwhile the new products should have a diversified design so that they can satisfy the requirement of different customer. The *diversity* can be measured by the sum distance from another product. Actually, the principle of design embodies the proximity with q and diversity from another object for a group of objects.

In this paper, we introduce two efficient algorithms, i.e., *Lazy RkGNN* (LR*k*GNN) and *Spatial pruning LRkGNN* (SLR*k*GNN) on datasets indexed by R-tree using Euclidean distance function, by several pruning heuristics, which reduce a large of subsets, the early stopping technologies which avoid searching the whole data space and the method of lazily outputting candidate subsets based sorting mechanism. Specifically, our main contributions are summarized as follows.

(i) We formalize the R*k*GNN query, an interesting variant of GNN query.

(ii) We develop two algorithms, viz. LR*k*GNN and SLR*k*GNN, for efficiently answering reverse top-k group nearest neighbor queries.

(iii) We demonstrate the effectiveness of our proposed heuristics and the performance of our proposed algorithms by extensive experiments.

2 Problem Formulation

In this section, we formally define the reverse top-k group nearest neighbor query. Table 1 contains the primary symbols used in our description.

Definition 1 (Reverse top-*k* Group Nearest Neighbor, R*k*GNN). Given a dataset *P*, a query object *q*, and two (user specified) parameters *m* and *k*, an R*k*GNN query finds *k* subsets, which have the least *aggregate distances*, such that each subset contains *m* data objects from *P* and has *q* as its *group nearest neighbor*.

Table 1. Frequently used symbols

Notation	Description
m	the number of data objects in a subset
maxdist(*e*)	the maximum distance between lower left and upper right corner of MBR(*e*)
G	a subset contained in dataset *P* which contains *m* entries
G_{rlt}	a set of the results of RGNN query
G_{rfn}	a set of candidate combinations of RGNN query
best_kdist	the *k*-th aggregate distance of subsets in G_{rlt} in ascending order
best_hdist	the minimum aggregate distance of orderly outputting subsets produced by the objects in auxiliary heap *H*
best_rfndist	the minimum *aggregate distance* of all subsets in G_{rfn}

A straightforward approach is to enumerate the combination (or say subset *G*) of the data objects, and compute the GNNs of each such combination. However, this approach is not efficient.

3 R*k*GNN Query Processing

3.1 Lazy R*k*GNN Algorithm

In this subsection, we describe the ***L**azy **R**everse top-**k** **G**roup **N**earest **N**eighbor algorithm* (LR*k*GNN), which makes use of the *sorting* and *threshold mechanisms* to shrink the search space based *lazy techniques*, and employs some effective *pruning heuristics* to discard candidate subsets.

Lazy Output and Advanced Sorting Technique

Generally, basic R*k*GNN algorithm searches in a *best-first* manner [1], and uses the minimum distance between current entry *e* and the query object *q* as the sorting *key*. Moreover, it enumerates all subsets (or say combinations) using *e* and the entries among *H* when an entry *e* is removed from the auxiliary heap *H*. If the candidate subset *G* only contains data objects, it is processed by invoking MBM algorithm in [1] to judge whether *q* is the GNN of *G*; otherwise *G* is inserted into the refined set G_{rfn} so that it will be processed later until the intermediate entry in *G* is expanded.

Unlike basic R*k*GNN algorithm, LR*k*GNN algorithm does not immediately enumerate the combinations for each time popped object until there are multiple data objects. LR*k*GNN uses two min-heaps, H_t and H_a, to save the data objects popped from

H satisfying (1), and all entries among H before enumerating, respectively. Then, LRkGNN generates all combinations so far using the data objects or entries of H_a.

Owing to that LRkGNN only need to retrieve k final results, LRkGNN uses the following inequality in (1) as the triggered conditions of enumerated combinations:

$$C_{|H_t|}^{m} \le k \le C_{|H_t|+1}^{m}, |H_t| \ge m+1 \quad (1),$$

where $|H_t|$ represents the number of data objects in H_t. The inequality in (1) guarantees that the algorithm generates at least k combinations; meanwhile it avoids to expand too many intermediate entries.

In fact, there still are too many entries in auxiliary heap H because a leaf node often includes more than one hundred data objects. Thus, the number of combinations is still very large, i.e., the number will be 161700 when $|H|=100$ and $m = 3$. To reduce the number of evaluating candidate subsets, LRkGNN progressively outputs some subsets from total subsets and process them according to their aggregate distances.

How to progressively generate some ordered subsets? Clearly, this problem is not trivial when the size of H_t is large. Our main idea is repeatedly applying two operations: decomposing and re-sorting. First, we sort the n entries in ascending order according their *key*, i.e., *mindist*(e, q). Then, we decompose the set (i.e., C_n^m) into two parts (i.e., C_{n-1}^m and C_{n-1}^{m-1}) through a recursion formula in (2):

$$C_n^m = C_{n-1}^m + C_{n-1}^{m-1}, n>m. \quad (2).$$

Next, the two parts of subsets are merged and form a sorting list of subsets in ascending order according to *adist*(q, G). Repeating the procedure until the aggregate distance of current subset is larger than the k-th aggregate distance of query results, namely, *best_kdist*. Our experiments show that only about 0.1% subsets need to be processed for each enumerating.

Example 1. Choose 2 objects from 4 ordered objects, o_1, o_2, o_3, and o_4, and their distances from q are 1, 2, 3, and 3.5, respectively. LRkGNN decomposes set C_4^2 into two sets, C_3^2 and $o_4 \cup C_3^1$. The former can be further decomposed two sets, C_2^2 (namely, $\{o_2, o_1\}$) and $o_3 \cup C_2^1$(namely, $o_3 \cup o_2|o_1$). The latter is $o_4 \cup o_3|o_2|o_1$, namely, $\{o_4, o_1\}$, $\{o_4, o_2\}$, and $\{o_4, o_3\}$. The set $\{o_2, o_1\}$ is firstly output, and then $\{o_3, o_1\}$ and $\{o_4, o_1\}$ is output. Finally, $\{o_3, o_2\}$, $\{o_4, o_2\}$, and $\{o_4, o_3\}$ are outputted as results.

Sorting and Threshold Method

Obviously, two key factors ultimately determine the actual performance of LRkGNN: how to progressively sort for all subsets formed by the entries among H, which might severely influence the number of subsets to be evaluated, and the strategy of choosing the stop point, which greatly shrink the search space.

Now, let's consider the second factor now. By Example 1, we know that LRkGNN progressively processes three parts of ordered subsets. Therefore, we use the variable *best_hdist* to save the minimum aggregate distance of each part of subsets. Moreover, when a new part of subsets are produced, *best_hdist* is also updated. For instance,

best_hdist is 3 when LR*k*GNN deals with the first part of subsets in Example 1. However, *best_hdist* becomes 4 when LR*k*GNN processed the second part of subsets.

In fact, LR*k*GNN can terminate the search using the following Theorem 1 and Corollary 1.

Theorem 1. R*k*GNN can stop the search if *best_hdist* and *best_rfndist* are both larger than *best_kdist*, namely, *best_hdist*≥ *best_kdist* and *best_rfndist*≥ *best_kdist*.

Proof. Since *best_rfndist* is larger than *best_kdist*, there is not any combination which has a lower aggregate distance than *best_kdist* so far. Therefore, any combination among G_{rfn} is not the result of R*k*GNN query. On the other hand, since *best_hdist*≥ *best_kdist*, there is not any new combination which has a less aggregate distance than *best_kdist.* So, the Theorem 1 is correct based on above two cases. □

Corollary 1. The search of R*k*GNN can be stopped if *best_hdist* is larger than *best_kdist*, namely, *best_hdist*≥ *best_kdist*, when the set of G_{rfn} is empty.

In practice, LR*k*GNN can also stop enumerating procedure by following Lemma 1.

Lemma 1. R*k*GNN can ignore the rest of processing subsets if the current subset G being processed has a larger aggregate distance than *best_kdist*, namely, *adist*(q, G)≥*best_kdist*.

Proof. Since all enumerated subsets are arranged in ascending order according to their aggregate distance, the subsets after the current subset G have a larger aggregate distance. Thus, there is not another subset which has a larger aggregate distance than that of G if *adist*(q, G)≥*best_kdist*. Therefore, the Lemma 1 is correct. □

Theorem 1, Corallary 1 and Lemma 1 is not trivial because they guarantee that the algorithm save much unnecessary cost of computation.

Pruning Heuristics

Next, we introduce some pruning heuristics to discard many candidate subsets by Lemma 2 and Lemma 3.

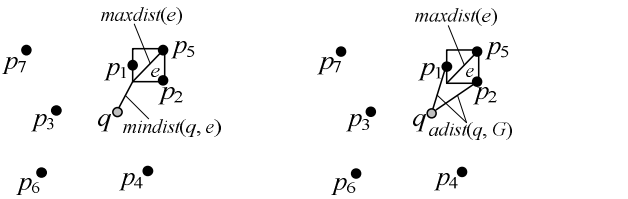
Lemma 2. Assume that a node e contains at least m+1 data objects, and *mindist*(q, e) denotes the minimum distance of node e from query object q. Any subsets among e cannot be the result of R*k*GNN query if *maxdist*(e) ≤*mindist*(q, e).

Proof. Since the minimum distance of e from q is *mindist*(e, q), we have *mindist*(q, e)**m* ≤*adist*(q,G) where all objects in G are from e. Assume that there is another point p' among e which is not contained in G. Obviously, the sum of distances between p' and data objects among G, *adist*(p', G) is less than or equal to *maxdist*(e)**m*. Thus, *adist*(p', G) ≤ *maxdist*(e)**m* ≤ *mindist*(q, e)**m*≤*adist*(q, G). Simplifying the inequality, we have *adist*(p', G)≤*adist*(q, G). This indicates that q is not the GNN of G. Since the subset G is a general assumption, therefore, we have that Lemma 2 is correct. □

Consider the example in Fig. 2(a) where the minimum bounding rectangle (MBR) of node e contains three data objects, p_1, p_2, and p_5. If the parameter m is 2, the node e will produce three combinations, $\{p_1, p_2\}$, $\{p_1, p_5\}$, and $\{p_2, p_5\}$. Because of $maxdist(e) \leq mindist(q, e)$, the three subsets of e, are not the result of RkGNN query. Although Lemma 2 can prune many combinations, it might be too restrictive to prune any combination when the MBR of the node is too large. At this point, we can still utilize the following Lemma 3 to prune the current subset G.

Lemma 3. If $maxdist(e)*m$ is less than or equal to $adist(q, G)$, then the current subset G cannot be the result of RkGNN query.

Proof. Assume that there is another point p' among e which is not contained in G. Since $maxdist(e)$ represents the maximum distance between any two data objects among e, the sum of distances between p' and data objects among G, $adist(p', G)$ is less than or equal to $maxdist(e)*m$. Moreover, $maxdist(e)*m$ is less than or equal to $adist(q, G)$, so $adist(p', G) \leq adist(q, G)$. In other words, q is not the GNN of G. Therefore, we have Lemma 3 is correct. □



(a) pruning all subsets of a node (b) pruning a subset of a node

Fig. 2. Illustrate the pruning heuristics

Query Processing

In this subsection, we describe the query procedure of LRkGNN algorithm. LRkGNN first initialize some variables at line 1, and then inserts all entries of *root* node of R-tree into an auxiliary heap H (line 2). The *key* is the minimum distance between current entry e and q. LRkGNN utilizes Theorem 1 and Corollary 1 to stop the search (lines 5~7). The lazy technique reflects on lines 9~18. Line 12 progressively generates candidate subsets in ascending order of $adist(q, G)$. Line 13 discards candidate subsets by Lemma 2 and Lemma 3. Line 14 guarantees that LRkGNN only need to evaluate a few of subsets when progressively output subsets. LRkGNN processes current subset G, where the procedure *UpdateRlt* is used to update the query results and current best aggregate distance *best_kdist* (Lines 15~17). The intermediate entry is expanded by lines 20~21. Line 22 employs Lemma 2 and Lemma 3 to mark the *false alarms*. LRkGNN makes use of lines 23~29 to expand the subsets which contains the current entry e using a similar procedure of lines 15~17.

Algorithm 1. LR*k*GNN($root$, P, q, m, k)

Input: The R-tree index over P, a query object q, two user specified parameters m and k

Output: the combinations which have q as their GNN

/* G_{rlt}: the set of query results; G_{rfn}: the set of refined subsets; H: auxiliary min-heap; H_t, H_a: min-heaps, sorted in ascending order of their distance from q. */

1. $G_{rlt} = \emptyset$, $G_{rfn} = \emptyset$, $H_t = \emptyset$, $H_a = \emptyset$, $best\_kdist = +\infty$, initialize min-heap H accepting entries (e, key)
2. insert ($root(I)$, $mindist(e,q)$) into heap H
3. **while** (heap H is not empty) **do**
4. remove top entry e from H
5. **if** ($best\_hdist \geq best\_kdist$)
6. **if** (G_{rfn} is empty) **return** // Theorem 1
7. **if** (G_{rfn} isn't empty **and** $best\_rfndist \geq best\_kdist$) **return** // Corollary 1
8. **if** (e is a data object)
9. $H_t = H_t \cup e$
10. **if** ($C_{|Ht|}^{m} \leq k \leq C_{|Ht|+1}^{m}$) $H_a = H_t \cup H$ //applying (1)
11. **else continue**
12. **for** $\forall G$ formed by the entries in H_a **do** //in ascending order of $adist(q, G)$
13. **if** (G is marked as *false alarm*) **continue //**Lemma 2 and Lemma 3
14. **if** ($adist(q, G) \geq best\_kdist$) **break**; //Lemma 1
15. **if** (G consists of data objects)
16. **if** (q is the GNN of G) *UpdateRlt*(G, G_{rlt}, k, $best\_kdist$)
17. **else** insert G into G_{rfn}
18. $H_t = \emptyset$, $H_a = \emptyset$
19. **else** //leaf node or intermediate node
20. **for** each data object or entry $e_i \in e$ **do**
21. insert e_i of e into heap H
22. mark $\forall$combination G of e pruned by Lemma 2~3 as *false alarm*
23. **for** $\forall$ combination G of G_{rfn} including e do
24. remove G from G_{rfn}
25. **for** each updated combination G' of G **do** //replace e of G using e_i
26. **if** ($adist(q, G') \geq best\_kdist$) **continue**
27. **if** (G' consists of data objects)
28. **if** (q is the GNN of G') *UpdateRlt*(G', G_{rlt}, k, $best\_kdist$)
29. **else** insert G' into G_{rfn}

3.2 Spatial Pruning LR*k*GNN Algorithm

Since LR*k*GNN need to invoke the MBM algorithm for each candidate subset, we present, in this subsection, an enhanced algorithm, called *Spatial pruning Reverse top-k Group Nearest Neighbor algorithm* (SLR*k*GNN), which takes advantage of *spatial pruning* method to identify or discard current candidate subset and does not has to invoke MBM algorithm to compute GNN.

In the sequel, we illustrate the intuition of our spatial pruning method by Figure 3. Specifically, let q be a query object and $G = \{p_1, p_2, \ldots, p_m\}$ be a candidate subset which contains in the dataset P. In particular, we denotes the *pruning region* (circle) of object p_i, centered at point p_i and with radius $dist(q, p_i)$, as $PR(q, p_i)$. For any combination G, its pruning region is defined as the union of pruning region of all objects in G. Moreover, we define the intersection of pruning region for G as $\cap PR(q, G) = PR(q, p_1) \cap PR(q, p_2) \ldots \cap PR(q, p_m)$.

By analyzing the cases in Fig. 3, we immediately obtain the following Lemma 4.

Lemma 4. (*RGNN Spatial Pruning*) Given a candidate subset G and a query object q. If there exists at least an object $p' \in P \backslash G$ which is contained in $\cap PR(q, G)$, G is not the result of RGNN query of q; if there does not exist any objects contained in $\cup PR(q,G)$, G is the result of RGNN query of q; otherwise, there exists at least an object contained in the region $R = \cup PR(q,G) \backslash (\cap PR(q, G))$, RGNN query only needs to search the objects among R to judge whether or not G has q as its GNN.

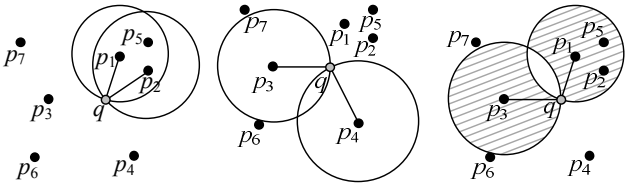


Fig. 3. Illustration of spatial pruning method in RkGNN query (m=2)

The importance of Lemma 4 is to enable RkGNN algorithm to shrink the search range when it cannot quickly judge whether current candidate subset G has q as its GNN. In practice, Lemma 4 saves much cost in distance computation and reduces many I/O accesses.

4 Experimental Evaluation

In this section, we experimentally evaluate the effectiveness of our proposed pruning heuristics and the performance of our developed algorithms for RkGNN query.

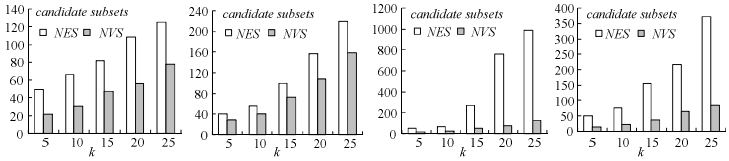
We use two real datasets, namely, *PP* and *NE*, from www.rtreepotral.org. We also create *Independent* (*IN*) and *Correlated* (*CO*) datasets with dimensionality $dim = 2$ and cardinality N in the range [20K, 100K] (*PP* is in the range [3K, 15K]). All datasets is normalized to range [0, 1]. Each dataset is indexed by an R-tree, with a page size of 4096 bytes.

We evaluate several factors, involving parameters k and m, and cardinality N. In each experiment, only one factor varies, whereas the others are fixed to their default values (k=15, m=3, N=60K). The *wall clock time* (i.e., the sum of I/O cost and CPU time, where the I/O cost is computed by charging 10ms for each page access, as with [1]), *the number of enumerating candidate subsets* (*NES*) which reflects the

performance of early stopping R*k*GNN computation, *the number of valid candidate subsets for GNN query* (*NVS*) which reports the capability of pruning heuristics (i.e., Lemma 1, Lemma 2, and Lemma 3, etc.), are used as the major performance metrics. Each reported value in the following diagrams in the average of 50 queries.

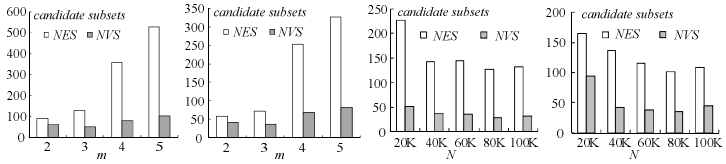
(1) The effectiveness of pruning techniques

This set of experiments aims at verifying the effectiveness of our proposed pruning heuristics. We vary *k* from 5 to 25, with *m* fixed at 3 in Figure 4, using four datasets, *PP*, *NE*, *CO* and *IN*. Evidently, the pruning techniques prunes a large number of candidate subsets (or say combinations), which validates their usefulness because *NES* and *NVS* both have a small value. In fact, a small value of *NES* verifies the efficiency of Theorem 1 and Corollary 1; meanwhile a small value of *NVS* confirms the efficiency of our proposed Lemma 1~Lemma 3. Although computing R*k*GNN is time consuming, our algorithm still obtains a higher performance owing to the adoption of early stopping technology which dramatically shrinks the search space. Similarly, Figure 5 and Figure 6 illustrate the prune efficiency of pruning techniques w.r.t. *m* and *N*, respectively, using real datasets and synthetic datasets, respectively. The diagrams confirm the observations and corresponding explanations of Figure 4.



(a) *PP* (*m*=3,15K) (b) *NE* (*m*=3,100K) (c) *CO* (*m*=3,100K) (d) *IN* (*m*=3,100K)

Fig. 4. Heuristic efficiency vs. *k*

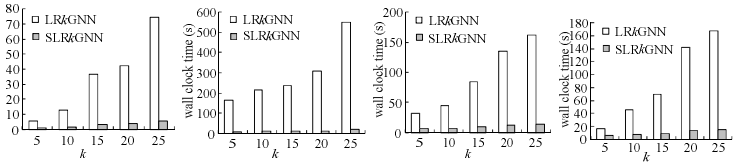


(a) *PP* (*k*=15,15K) (b) *NE* (*k*=15,100K) (a) *CO* (*k*=15, *m*=3) (b) *IN* (*k*=15, *m*=3)

Fig. 5. Heuristic efficiency vs. *m* **Fig. 6.** Heuristic efficiency vs. *N*

(2) The results on R*k*GNN queries

The second set of experiments studies the performance of our proposed algorithms in answering R*k*GNN queries. First, we explore the impact of *k* on algorithms, and the results are shown in Figure 7. Clearly, SLR*k*GNN outperforms LR*k*GNN in all cases because it integrates spatial pruning heuristics to prune unqualified candidate points. Furthermore, as *k* grows, the cost of LR*k*GNN and SLR*k*GNN slightly increase.

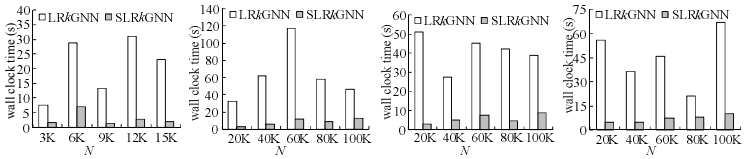


(a) *PP* (*m*=3,9K) (b) *NE* (*m*=3,60K) (c) *CO* (*m*=3,60K) (d) *IN* (*m*=3,60K)

Fig. 7. RkGNN cost vs. *k*

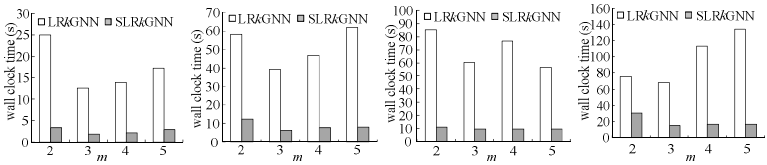
Then, we evaluate the effect of *N* on the algorithms. Figure 8 depicts the cost as a function of *N*. SLR*k*GNN clearly gains a better performance than LR*k*GNN. When *N* increases, the cost of the algorithms does not always increase. This is because, on the one hand, the query objects is randomly choose on each test, and on the other hand, there is a big chance of finding some subsets which have *q* as their GNN in a big dataset than in a small dataset.

Finally, we inspect the effect of *m* on the algorithms, by fixing *k* = 15, *N*=60K (except for *N*= 9K on dataset *PP*). Figure 9 shows the performance of algorithms as a function of *m*. Again, SLR*k*GNN outperforms LR*k*GNN in all cases. Moreover, although the number of candidate subsets dramatically increase as *m* increases, the cost of algorithms does not always grow. This is because on the one hand, Theorem 1, Corollary 1 and Lemma 1 prune most of the candidate subsets, and on the other hand, a big *m* brings more candidate subsets which have *q* as their GNN object.



(a) *PP* (*k*=15, *m*=3) (b) *NE* (*k*=15, *m*=3) (c) *CO* (*k*=15, *m*=3) (d) *IN* (*k*=15, *m*=3)

Fig. 8. RkGNN cost vs. *N*



(a) *PP* (*k*=15,9K) (b) *NE* (*k*=15,60K) (c) *CO* (*k*=15, 60K) (d) *IN* (*k*=15, 60K)

Fig. 9. RkGNN cost vs. *m*

5 Conclusions

This paper firstly introduces and solves a new form of nearest neighbor queries, namely, *reverse top-k group nearest neighbor* (R*k*GNN) retrieval. We develop two

efficient algorithms, LR*k*GNN and S LR*k*GNN, using sorting and threshold technique, and some smart pruning heuristics to prune away most of unqualified candidate subsets. In the future, we intend to devise more efficient algorithm(s) for answering R*k*GNN queries by using the reuse technique. Another interesting direction for future work is to extend our approaches to tackle other variants of R*k*GNN queries, i.e., constrained R*k*GNN, the R*k*GNN in metric spaces, etc.

Acknowledgements. This work was supported in part by NSFC 61003049, ZJNSF LY12F02047 and LY12F02019, the Science and Technology Plan Fund of Jiaxing Municipal Bureau under Grant 2011AY1005, the Fundamental Research Funds for the Central Universities under Grant 2012QNA5018, the Key Project of Zhejiang University Excellent Young Teacher Fund (Zijin Plan).

References

1. Papadias, D., Tao, Y., Mouratidis, K., Hui, C.K.: Aggregate Nearest Neighbor Queries in Spatial Databases. ACM Trans. Database Syst. 30(2), 529–576 (2005)
2. Li, F., Yao, B., Kumar, P.: Group enclosing queries. IEEE Trans. Knowl. Data Eng. 23(10), 1526–1540 (2010)
3. Deng, K., Sadiq, S., Zhou, X., Xu, H., et al.: On Group Nearest Group Query Processing. IEEE Trans. on Knowl. and Data Engineering 24(2), 295–308 (2012)
4. Lian, X., Chen, L.: Probabilistic Group Nearest Neighbor Queries in Uncertain Databases. IEEE Trans. on Knowl. and Data Engineering 20(6), 809–824 (2008)
5. Yiu, M.L., Mamoulis, N., Papadias, D.: Aggregate Nearest Neighbor Queries in Road Networks. IEEE Trans. Knowl. Data Eng. 17(6), 820–833 (2005)
6. Zhu, L., Jing, Y., Sun, W., Mao, D., Liu, P.: Voronoi-based Aggregate Nearest Neighbor Query Processing in Road Networks. In: GIS, pp. 518–521 (2010)
7. Xiao, X., Yao, B., Li, F.: Optimal Location Queries in Road Network Databases. In: IEEE ICDE, pp. 804–815 (2011)
8. Li, Y., Li, F., Yi, K., Yao, B., Wang, M.: Flexible Aggregate Similarity Search. In: SIGMOD, pp. 1009–1020 (2011)