Expert Systems With Applications 53 (2016) 57–74

Contents lists available at ScienceDirect

Expert Systems With Applications

journal homepage: www.elsevier.com/locate/eswa

Monochromatic and bichromatic reverse top-*k* group nearest neighbor queries

Bin Zhang[a], Tao Jiang[a,*], Zhifeng Bao[b], Raymond Chi-Wing Wong[c], Li Chen[a]

[a] *College of Mathematics Physics and Information Engineering, Jiaxing University, 56 Yuexiu Road (South), Jiaxing 314001, China*
[b] *School of Computer Science and Information Technology, RMIT University, GPO Box 2476, Melbourne 3001 Victoria, Australia*
[c] *Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong, China*

A R T I C L E   I N F O

Keywords:
Query processing
Group nearest neighbor
Top-*k* query
Spatial database

A B S T R A C T

The Group Nearest Neighbor (GNN) search is an important approach for expert and intelligent systems, i.e., Geographic Information System (GIS) and Decision Support System (DSS). However, traditional GNN search starts from users' perspective and selects the locations or objects that users like. Such applications fail to help the managers since they do not provide managerial insights. In this paper, we focus on solving the problem from the managers' perspective. In particular, we propose a novel GNN query, namely, the reverse top-*k* group nearest neighbor (R*k*GNN) query which returns *k* groups of data objects so that each group has the query object *q* as their group nearest neighbor (GNN). This query is an important tool for decision support, e.g., location-based service, product data analysis, trip planning, and disaster management because it provides data analysts an intuitive way for finding significant groups of data objects with respect to *q*. Despite their importance, this kind of queries has not received adequate attention from the research community and it is a challenging task to efficiently answer the R*k*GNN queries. To this end, we first formalize the reverse top-*k* group nearest neighbor query in both monochromatic and bichromatic cases, and then propose effective pruning methods, i.e., *sorting and threshold pruning, MBR property pruning*, and *window pruning*, to reduce the search space during the R*k*GNN query processing. Furthermore, we improve the performance by employing the reuse heap technique. As an extension to the R*k*GNN query, we also study an interesting variant of the R*k*GNN query, namely a *constrained reverse top-k group nearest neighbor* (CR*k*GN) query. Extensive experiments using synthetic and real datasets demonstrate the efficiency and effectiveness of our approaches.

1. Introduction

Recently, group nearest neighbor (GNN) queries (Papadias, Tao, Mouratidis, & Hui, 2005) have attracted more and more attentions due to its wide usage in many applications, such as geographic information systems (Gao, Zheng, Chen, Chen, & Li, 2011), location-based services (Wong, Ozsu, Yu, Fu, & Liu, 2009; Yu, 2016), navigation systems, mobile computing systems (Mouratidis, Yiu, Papadias, & Mamoulis, 2006), and data mining (e.g., clustering objects). Specifically, given a set of data objects $P = \{p_1, p_2, \ldots, p_n\}$ and a set of query objects $Q = \{q_1, q_2, \ldots, q_m\}$, a group nearest neighbor (GNN) query returns a data object p in P whereby p has the smallest sum of distances to all data objects in Q, that is, $\sum_{q_i \in Q} dist(q_i, p) \leq \sum_{q_i \in Q} dist(q_i, p')$, $p' \in P \backslash p$, where $dist(.)$ is the Euclidean distance function.

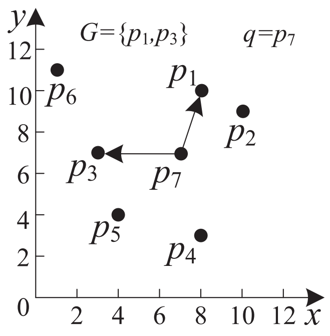
In this paper, we study the GNN queries from a reverse perspective. A *reverse group nearest neighbor (RGNN)* finds all sets of data objects (G) that have a query object q as their GNN, i.e., $\sum_{\forall G \subset P, p_i \in G} dist(p_i, q) \leq \sum_{\forall G \subset P, p_i \in G, p' \in P \backslash G} dist(p_i, p')$, $|G| = m$ $(m \geq 2)$[1]. Since there could be many such sets of G while not all of them are of same interest in applications which will be discussed later (in Section 2.2), we refine the definition of RGNN to R*k*GNN which returns only the k nearest sets to the query object q in terms of the aggregated distances between G and q (denoted by $adist(q, G)$).

Example 1. To have a better understanding of the R*k*GNN query, let us look at the example shown in Fig. 1 where the black points denote a set of objects in a 2-dimensional space, i.e., $P = \{p_1, p_2, \ldots, p_7\}$. Consider a R*k*GNN query with the query object

* Corresponding author. Tel.: +86 159 9031 6256; fax: +86 573 8364 0102.
E-mail addresses: jxtaojiang@gmail.com, tjiangguido@gmail.com (T. Jiang), zhifeng.bao@rmit.edu.au (Z. Bao), raywong@cse.ust.hk (R.C.-W. Wong), chenli20040415@hit.edu.cn (L. Chen).

[1] Note that when $m = 1$, RGNN becomes the well-known reverse nearest neighbor (RNN) query.

http://dx.doi.org/10.1016/j.eswa.2016.01.012
0957-4174/© 2016 Elsevier Ltd. All rights reserved.



RGNN	$adist(G, q)^2$
$G_1=(p_1,p_2)$	23
$G_2=(p_1,p_3)$	26
$G_3=(p_1,p_4)$	27
$G_4=(p_1,p_5)$	28
$G_5=(p_2,p_3)$	29
$G_6=(p_2,p_4)$	30
$G_7=(p_2,p_5)$	31
$G_8=(p_3,p_5)$	34
$G_9=(p_4,p_5)$	35

Fig. 1. Monochromatic reverse group nearest neighbor query.

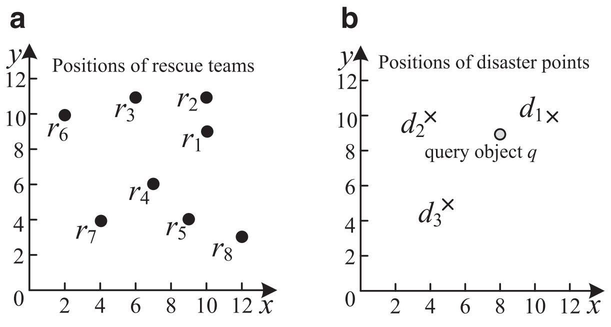


Fig. 2. Bichromatic reverse group nearest neighbor query. (a) Data set A (rescue teams) (b) Data set B (disaster points).

p_7, $k = 3$, and $m = 2$. It is easy to find that p_7 is the GNN of $G_1 = \{p_1, p_2\}$ as p_7 has the smallest aggregated distance to G_1 compared with any other objects in the dataset P, i.e., $adist(p_7,G_1) = \sum_{g_i\in G_1} dist(g_i, p_7) < \sum_{g_i\in G_1} dist(g_i, p') \ (p' \in P\backslash p_7)$. Similarly, we can find that p_7 is also the GNN of sets $G_2 = \{p_1, p_3\}$, $G_3 = \{p_1, p_4\}$, $G_4 = \{p_1, p_5\}$, $G_5 = \{p_2, p_3\}$, $G_6 = \{p_2, p_4\}$, $G_7 = \{p_2, p_5\}$, $G_8 = \{p_3, p_5\}$, and $G_9 = \{p_4, p_5\}$. Therefore, $G_1, G_2, \ldots,$ and G_9 are RGNNs of the object p_7 (or say that $G_1, G_2, \ldots,$ and G_9 have p_7 as their GNNs). According to the aggregated distances between p_7 and these sets as shown in the table in Fig. 1, G_1, G_2 and G_3 are the top three closest sets to p_7, and hence G_1, G_2 and G_3 are the answer of this R3GNN query.

Similar to GNN queries, our proposed RkGNN query is very useful for various applications such as business analysis applications and decision support systems. For example, GNN queries can be used in the following scenario: Given multiple supermarket branches, where would be a good location to set up a warehouse that has the shortest aggregated distance to all the branches? While RkGNN can be used to answer the query in a reverse manner: Given a warehouse, where would be the good locations to open new supermarket branches? More specifically, recall Example 1, object p_7 can be considered as a warehouse, and other objects can be considered as candidate locations for new supermarket branches. Note that the RkGNN query has another important feature that it can be used to address diversity concerns (Qin, Yu, & Chang, 2012) simultaneously. Specifically, the supermarket company wants the new branches to be close to the warehouse but may not want the new branches to be too close to each other which would otherwise cause unnecessary competition. The RkGNN query can return k possible answers to allow the supermarket company to choose the new branches that are far apart from each other. For example, $G_1(p_1, p_2)$ may not be selected compared with $G_2(p_1, p_3)$ because G_2 contains two locations that are far apart from each other.

Consider another example of the RkGNN queries in supporting decision making. A corporation plans to release a series of m products, and wants to select these m products from n candidate designs. The selection criterion is measured by the distance between the candidate products' properties and a prototype's properties. This problem can be modeled as an RkGNN query by treating the prototype as the query object and the candidate products as the data objects. The query results will contain the groups of candidate products which are most similar to the prototype. Furthermore, if the corporation also desires to have diversity among the released products, the corporation can filter out the query results which contain products with similar properties.

It is worth noting that RkGNN query is more suitable for decision makings in the aforementioned scenarios than the traditional *k nearest neighbor* (kNN) queries and range queries. This is because the RkGNN query not only considers minimizing the overall distance between the candidate objects and the query object, but also ensures the diversity of the answers. In contrast, kNN queries or range queries will only return answers that are concentrated in the query region. For example, a 2NN query with the query object q ($=p_7$) in Fig. 1 will return p_1 and p_2 which are too close to each other and hence may not be good choices of new supermarket branches.

It is a challenging task to perform an efficient RkGNN query over a large dataset. A naïve solution can enumerate up to $n!/((n-m)!m!)$ ($n \gg m$, $m \geq 2$) combinations of data objects and then check each combination to see whether the query object is its GNN. Such an approach is obviously time consuming due to the large number of combinations to be verified. Recently, there have been some algorithms that also consider combinations of objects to form a query answer (Chuang, Su, & Lee, 2013; Im & Park, 2012; Jiang, Zhang, Lin, Gao, & Li, 2015). However, they are essentially different technologies as they use different pruning conditions and their target queries are not exactly the same. Moreover, these approaches only handle skyline queries (Gao et al., 2014; Jiang, Gao, Zhang, Lin, & Li, 2014) that have totally different selection criteria compared with RkGNN. For example, traditional skyline queries returns all non-dominated objects whereas our proposed methods address nearest neighbor problem. Therefore, these algorithms cannot be used to solve the RkGNN problem.

Therefore, in this paper, we propose several efficient RkGNN query algorithms by integrating a suite of pruning and reuse techniques. Specifically, we develop a *sorting and threshold pruning* (STP) and a *lazy outputting* (LO) of combinations to reduce the search space and avoid unnecessary computation. We propose effective pruning heuristics, including *MBR property pruning* (MP) and *window pruning* (WP), which utilize the spatial relationship among data objects to prune the candidate combinations. We leverage the *reuse heap* (RH) technique to judiciously reuse available information and avoid repeated efforts.

Moreover, our algorithms are able to address not only the *monochoromatic reverse top-k group nearest neighbor* (MRkGN) as shown in aforementioned examples, but also the *bichromatic reverse top-k group nearest neighbor* (BRkGN) query. In the MRkGN query, both the query object and the data objects are in the same dataset shown in previous examples. In the BRkGN query, the query object and the data objects are from two different datasets.

Example 2. Fig. 2 illustrates an example BRkGN query, whereby Fig. 2(a) gives a set of rescue teams' positions (denoted as set A), and Fig. 2(b) shows the positions of disasters (denoted as set B). Given a disaster's position (i.e., a query object q from B), a BRkGN query can help identify a group of nearby rescue teams (from set A) with complementary skills to come for rescue. For example, we should select a group of rescue teams, that is, $\{r_1, r_3\}$ or $\{r_2, r_3\}$ for disaster point q if $m = 2$ because $\{r_1, r_3\}$ and $\{r_2, r_3\}$ take q

as their GNN. Note that although q has r_1 and r_2 as their nearest rescue teams, $\{r_1, r_2\}$ cannot be regarded as the RGNN of q since $\{r_1, r_2\}$ does not take q but d_1 as its GNN.

A preliminary version of this work appeared in Jiang, Gao, Zhang, Liu, and Chen (2013), where we present the basic concept of RkGNN. In this paper, we extend our work by including the following new contributions: (1) we define two new sub-types of RkGNN, i.e., MRkGN and BRkGN, which are useful in different scenarios; (2) we significantly improve the query performance by incrementally generate the candidate combinations of data objects, and introducing some novel pruning heuristics; (3) we also introduce a new concept, i.e., constrained RkGNN query, which considers the query in a given constrained region; (4) we have theoretically proved the correctness of our proposed approaches; (5) we have conducted a more thorough experimental study that verified the efficiency of our proposed new approaches.

The rest of the paper is organized as follows. Section 2 outlines the related work. Section 3 gives the formal definition of RkGNN query. Section 4 elaborates the pruning techniques of RkGNN query and our proposed processing algorithm. Section 5 further improves the RkGNN query by reuse heap method. The bichromatic RkGNN query is presented in Section 6. Section 7 extends the RkGNN query to the constrained setting. Section 8 reports the experimental results and our findings. Section 9 concludes the paper with directions for future work.

2. Related work

2.1. Nearest neighbor (NN) queries and reverse neighbor (RNN) queries

A nearest neighbor (NN) query retrieves the object that is closest to a query object q in a multidimensional dataset. Many algorithms have been proposed for the NN queries. The original NN query algorithm (Roussopoulos, Kelly, & Vincent, 1995) for R-trees (Guttman, 1984) follows the policy of depth-first (DF) traversal. The DF algorithm starts from the root of the index of the data objects, and visits recursively the nodes with the smallest *mindist* from q. During backtracking to the upper levels, DF only visits the entries whose minimum distances are smaller than the distance of the NN already retrieved. Another representative NN algorithm (Hjaltason & Samet, 1999) traverses the index in a *best-first* (BF) manner and achieves better performance than the DF algorithm. The BF algorithm maintains a min-heap H with the entries visited so far, sorted by their *mindist*. Similar to DF, BF starts from the root, and inserts all its entries into H (together with their *mindist*). Then, at each step, BF visits the node in H with the smallest *mindist* until the first NN is found. BF is *incremental*, i.e., it returns the NNs in an ascending order of their distance to the query object.

Since our algorithms also use the BF retrieval, we provide more details of the BF algorithm via an illustrative example in Fig. 3, where Fig. 3(a) and (b) give the minimum bounding rectangles (MBRs) and the structure of an R-tree, respectively. At the beginning, BF inserts $\langle N_1, mindist(q, N_1)\rangle$ and $\langle N_2, mindist(q, N_2)\rangle$ into a min-heap H, where $mindist(q, N_i)$, $i \in [1, 2]$, is the minimal distance from query point q to any point on the perimeter of N_i (Roussopoulos et al., 1995). Then the algorithm accesses the top node N_1 in H, retrieves the content of N_1 and inserts all its children entries in H. Now, the content of H becomes $\{\langle N_2, mindist(q, N_2)\rangle, \langle N_4, mindist(q, N_4)\rangle, \langle N_3, mindist(q, N_3)\rangle\}$. Similarly, the next two nodes accessed are N_2 and N_4, in which p_1 is discovered as the current NN. After the node N_4 is accessed, the content of H becomes $\{\langle p_1, mindist(q, p_1)\rangle, \langle N_5, mindist(q, N_5)\rangle, \langle p_3, mindist(q, p_3)\rangle, \langle N_3, mindist(q, N_3)\rangle, \langle N_6, mindist(q, N_6)\rangle, \langle p_5, mindist(q, p_5)\rangle\}$. Continuing the procedure, the algorithm retrieves the content of

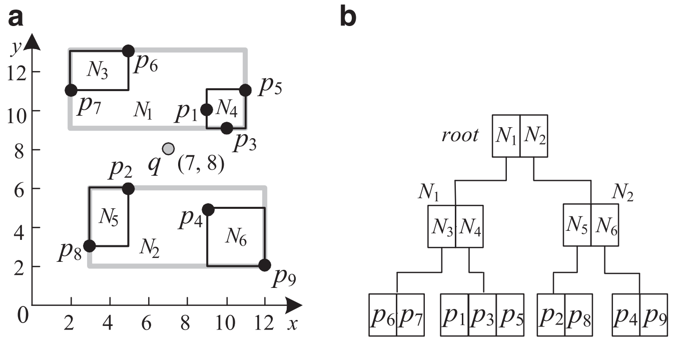


Fig. 3. An example of R-tree. (a) the MBRs (b) the structure.

N_5, and obtains the next NN p_2. BF will report other NNs in an ascending order of their distances between the rest of each data object and q, i.e., followed by p_3, p_4, p_5, p_6, p_7 and p_8.

Other NN algorithms include optimal multi-step algorithm (Seidl & Kriegel, 1998), Voronoi-based algorithm (Kolahdouzan & Shahabi, 2004), et al. In addition, there are many algorithms, which solve the NN queries for different application environments, such as continuous NN monitoring over moving objects (Yu, Pu, & Koudas, 2005), road network (Mouratidis et al., 2006), and obstacles (Gao et al., 2011).

In addition to the NN query, Korn and Muthukrishnan (2000) proposed a novel type of query, called *reverse nearest neighbor* (RNN) query, which identifies the influence of a query object q by retrieving those objects that have q as their nearest neighbor. The original RNN algorithm (Korn & Muthukrishnan, 2000) precomputes for each data object p its nearest neighbor to build an index based on the R-tree, called the RNN-tree, and then uses the RNN-tree to retrieve the RNNs given a query object q. Stanoi, Agrawal, and Abbadi (2000) eliminates the need for pre-computing all NNs by partitioning a 2D search space centered at the query object into six equal-sized sectors. Later, Tao, Papadias, and Lian (2004) proposed the TPL algorithm that exploits a half-plane property in space to locate RkNN candidates. However, TPL is restricted to the Euclidean space. More recently, Tao, Yiu, and Mamoulis (2006) presented a new algorithm for efficient RNN search in generic metric spaces. Moreover, there are several variants of RNN queries, which consider different application scenarios, such as object monitoring (Xia & Zhang, 2006), location-based service (Wong et al., 2009), ad hoc subspace (Yiu & Mamoulis, 2006), and graph network (Yiu, Papadias, Mamoulis, & Tao, 2006).

2.2. Group nearest neighbor (GNN) queries

The GNN query is first introduced by Papadias, Shen, Tao, and Mouratidis (2004). They proposed three efficient algorithms, namely, multiple query method (MQM), simple point method (SPM), and minimum bound method (MBM). MQM performs incremental nearest neighbor (NN) queries for each point in Q and combines their results using the *threshold algorithm* (Fagin, Lotem, & Naor, 2001). MQM will terminate the procedure of search if all the retrieved data objects so far is below the threshold *best_dist*. SPM calculates the centroid of the query set Q, which is a point in space with a small sum of distances to all query points. The centroid is used to prune the search space by exploring the triangular inequality. MBM is the most related work to our study, which is used in our experiments as a part of our basic RkGNN algorithm. MBM utilizes a *minimum bounding rectangle* (MBR) to bound all the query objects in Q and also explores the triangle inequality to reduce the search space. It recursively visits nodes beginning from the root of the R-tree and prunes the intermediate nodes that cannot contain any results by sequentially using two pruning conditions. Let M be the MBR of query objects in Q, N be the current node (and its MBR) from the R-tree, and *bestDist* be the distance of the best GNN found so far. The two pruning conditions are $n{*}mindist(N, M) \geq bestDist$, and $\sum_{q_i \in Q} mindist(N, q_i) \geq bestDist$, where $mindist(N, M)$ is the minimum distance between any two points on the perimeters of N and M, respectively, n is the cardinality of Q, and $mindist(N, q_i)$ is the minimum distance between N and query object $q_i \in Q$. Either condition can safely prune the node N (and its subtree). However, the second condition is applied only for nodes that pass the first one in order to save unnecessary computations. Later, the GNN query is generalized into *aggregate nearest neighbor* (ANN) (Papadias et al., 2005) query to support the *max* and *min* aggregated distance functions. Li, Yao, and Kumar (2010) explored new geometric insights, such as the minimum enclosing ball, the convex hull and the furthest Voronoi diagram to prune intermediate nodes.

Other GNN algorithms can be classified into two main categories. The first category tackles different user scenarios, such as *probabilistic group nearest neighbor* (PGNN) query (Lian & Chen, 2008), group nearest neighbor queries for ensuring user privacy (Hashem, Kulik, & Zhang, 2010), *metric aggregate similarity search* (MASS) (Razente, Barioni, Traina, Faloutsos, & C. T., 2008) based on metric index, ANN query in road networks (Yiu, Mamoulis, & Papadias, 2005). Recently, Xiao, Yao, and Li (2011) took the ANN query as the optimal location query and proposed a unified solution that addresses all variants of the optimal location query in road network with several efficient algorithms. The second category contains some other variations of ANN queries by loosening the restrictions on the definitions of the nearest neighbor, such as *flexible aggregate nearest neighbor* (FANN) query (Li, Li, Yi, Yao, & Wang, 2011) and *group nearest group* (GNG) query (Deng et al., 2012). Unfortunately, none of these algorithms can be directly applied to our proposed R*k*GNN query.

2.3. Combinatorial queries and diversified queries

Another relevant problem to our work is the combinatorial query (Chuang et al., 2013; Guo, Xiao, & Ishikawa, 2012; Im & Park, 2012; Jiang et al., 2015; Magnani & Assent, 2013), which have some combinations of objects as the query results. Su, Chuang, and Lee (2010) first proposed the *top-k combinatorial skyline query* (*k*CSQ), which is to find the top-*k* combinations of data objects rather than individual objects that form the skyline. Guo et al. (2012) also studied the combination skyline query and developed a pattern-based pruning algorithm to further reduce the search space. Later, Im, and Park (2012) introduced the *group skyline query* which generates all skyline groups rather than *k* groups in the *k*CSQ query (Su et al., 2010). Recently, Magnani and Assent (2013) proposed *aggregate skylines*, which merge the functionalities of two basic database operators, *skyline* and *group by*. Jiang et al. (2015) presented an efficient algorithm for evaluating top-*k* combinational metric skyline. However, as pointed out in Section 1, the above algorithms cannot be used to address our problem because the goal of our algorithm focuses on the computation of nearest neighbors rather than the skyline objects.

The last related work is the *diversified queries* (Drosou & Pitoura, 2010; Gollapudi & Sharma, 2009), which have applications in many domains, such as user personalized results and ambiguous keywords searches. Gollapudi and Sharma (2009) developed a set of natural axioms that a diversified system is expected to satisfy, and show that no diversified function can satisfy all the axioms simultaneously. Drosou and Pitoura (2010) surveyed various definitions, algorithms and metrics for result *diversification*. Vieira, Razente, and Barioni (2011) described a general framework to evaluate and optimize diversified query results. In these methods, an initial ranking candidate set produced by a query is used to construct a result set, where elements are ranked with respect to relevance and diversity features. Qin et al. (2012) defined a general diversified top-*k* search problem that only considers the similarity of the search results themselves. They then proposed a framework that can extend most existing solutions for top-*k* query processing to diversified top-*k* search. Tao, Ding, Lin, and Pei (2009) are the first one who introduce the concept of *diversity* into skyline queries, and propose a representative skyline. Our algorithm also integrates the concept of diversity. To the best of our knowledge, this is the first attempt on diversified query over the combinatorial queries.

Table 1
Frequently used symbols.

Notation	Description
$maxdist(e)$	The maximum distance between the lower left and the upper right of MBR of entry e
G, m	G is a combination from the dataset, and m is the cardinality of G
$adist(q, G)$	The sum of distances from q to the set G
G_{rlt}	A resultant set of the R*k*GNN query
G_{rfn}	A refined set of candidate combinations of the R*k*GNN query
L_n	An ordered list of n data objects sorted in ascending manner by their weights
$CL(n, m)$	An ordered list that contains all combinations by selecting m data objects from n data objects in L_n
$\triangle CL(n, m))$	An ordered list that contains all *incremental* combinations of $CL(n, m)$
$CL'(n, m)$	The updated list of $CL(n, m)$
$best\_kdist$	The k-th aggregated distance of combinations in G_{rlt} in ascending order
$delta\_mindist$	The minimum aggregated distance of currently output incremental combinations
$rfn\_mindist$	The minimum *aggregated distance* of all combinations in G_{rfn}

3. Problem definition

In this section, we first define the *monochoromatic reverse top-k group nearest neighbor* (MR*k*GN) query and the *bichoromatic reverse top-k group nearest neighbor* (BR*k*GN) query. Then we present the baseline algorithms. Table 1 summarizes the key notations used throughout the paper.

We assume that the data set P is indexed by an R-tree (Guttman, 1984) and consider the Euclidean distance between objects in the following discussion. Given a set of query objects $Q = \{q_1, q_2, \ldots, q_m\}$, where Q may or may not be a subset of P. Recall that a group nearest neighbor (GNN) query will return a data object p in P where p has the smallest sum of distances to all query objects in Q (Papadias et al., 2004). Formally, $\text{GNN}(Q, P) = argmin_{p \in P}\{adist(p, Q)\}$, where $adist(p, Q)$ represents the sum of distances of $p \in P$ from all data objects $q_i \in Q$, i.e., $adist(p, Q) = \sum_{q_i \in Q} dist(p, q_i)$and $dist(p, q_i)$ is the Euclidean distance between p and q_i. Based on the definition of the GNN query, we define the MR*k*GN and BR*k*GN as follows.

Definition 1 (**Monochromatic Reverse Group Nearest Neighbor, MRGN).** Given a dataset P and a query object q ($q \in P$), a monochromatic reverse group nearest neighbor (MRGN) query retrieves all combinations that have q as their GNN, where each combination contains m data objects. Formally, $\text{MRGN}(q, P) = \{G | q = \text{GNN}(G, P), G \subset P \wedge |G| = m, m \geq 2\}$.

Definition 2 (**MR*k*GN).** A monochoromatic reverse top-*k* group nearest neighbor (*MRkGN*) query returns the top k combinations obtained from the corresponding MRGN query. Formally, $\text{MRkGN}(q, k, P) = \{G^* | \, G^* \in argmin_{G_i \in MRGN(q,P)} (\sum_{j=1}^{k} adist(G_j \in G_i, q))\}$.

Definition 3 (**Bichromatic Reverse Group Nearest Neighbor, BRGN).** Let A and B denote two datasets respectively. Given a query

q ($q \in B$), a bichromatic reverse group nearest neighbor (BRGN) query retrieves all combinations in A that have q as their GNN in B, where each combination contains m data objects. Formally, BRGN(q, A, B) = $\{G|q = \text{GNN}(G, B), G \subset A \wedge q \in B \wedge |G| = m, m \geq 2\}$.

Definition 4 (**BR*k*GN**)**.** A bichromatic reverse top-k group nearest neighbor (*BRkGN*) query returns the top k combinations obtained from the corresponding BRGN query. Formally, *BRkGN(q, k, A, B)* = $\{G^*|\ G^* \in argmin_{G_i \in BRGN(q,A,B)}(\sum_{j=1}^{k} adist(G_j \in G_i, q))\}$.

Naïve Solution to MR*k*GN and BR*k*GN: Since the MR*k*GN and BR*k*GN queries have not been studied by any previous work, we introduce a naïve solution as the baseline approach for comparison. Specifically, the naïve algorithm sequentially enumerates all combinations using a list of data objects in P sorted by the distance between a data object and the query object q. For each combination G, the naïve solution will compute its GNN, and check if the GNN of G is the query object q. If so, this combination will be recorded. Whenever the algorithm has identified at least k such combinations, the naïve algorithm will sort them in an ascending order of their aggregated distances to q, and then use them to update the final results until the top k combinations are found. As discussed in Section 1, such an approach that uses *linear scan* (denoted as LS algorithm) is very time consuming due to the enormous amount of combinations to be evaluated. In what follows, we present our proposed efficient query algorithms.

4. Monochoromatic reverse top-*k* group nearest neighbor (MRkGN) query

Our proposed MR*k*GN algorithm leverages various techniques to achieve efficient performance. Its general framework consists of three phases, *indexing, pruning*, and *refinement phase*. Specifically, the *indexing phase* builds up m B$^+$-tree indexes during the procedure of R-tree traversal. The algorithm searches the R-tree in *best-first* (BF) manner (Hjaltason & Samet, 1999) until the auxiliary heap H becomes empty. Each time, whenever a data object p is removed from H, the algorithm enumerates all incremental combinations using p and other data objects (or nodes) in H. These combinations will be inserted into the corresponding B$^+$-trees and then processed one by one. If a combination contains at least an intermediate node, it will be inserted into a refined set, G_{rfn}. The *pruning phase* aims to prune the combinations that are not qualified as the query results. In this phase, we propose a so called *sorting and threshold pruning, MBR property pruning*, and several *window pruning* methods to significantly reduce the search space. Finally, for each remaining candidate combinations in G_{rfn}, the *refinement* phase extends the combinations by replacing the current entry e with its children e_i so that the resulting combinations will only contain data objects but not any intermediate nodes. Then, the obtained combinations will be processed to find the final query results. In the rest of this section, we will first introduce the underlying techniques and then present the overall query algorithm.

4.1. Incremental generation of combinations

In this subsection, we introduce a *combinatorial sorting algorithm* (CSA) based on the *dynamic programming*, which are designed to reduce the number of combinations that need to be generated and evaluated. CSA takes advantages of previously generated combinations to form new combinations in an incremental manner.

More specifically, let L_n be a list of n data objects, $\{p_1, p_2, \ldots, p_n\}$, where the weight of each data object, $w(p_i)$, satisfies the inequality, $w(p_1) < w(p_2) < \ldots, < w(p_n)$. We use $dist(q, p_i)$ as the weight of p_i ($i \in [1, n]$). As a result, L_n can be seen as an ordered list that contains n objects in an ascending order by $dist(p_i, q)$. Let an m-object *combination* be a combination of a cardinality of m (i.e., combined by m data objects). For a combination G, its weight is denoted as *adist*(q, G). Let $CL(n, m)$ be the ordered list that contains all m-object *combinations* of a list L_n with n data objects $\{p_1, p_2, \ldots, p_n\}$. For example, given $L_3 = \{p_1, p_2, p_3\}$, we can generate an ordered list of all 2-object combinations, i.e., $CL(3, 2) = \{\{p_1, p_2\}, \{p_1, p_3\}, \{p_2, p_3\}\}$.

Next, we introduce $\oplus$ as an adding operator which adds a data object to each combination in a sorted list. For instance, given p_3 and $CL(2, 1) = \{\{p_1\},\{p_2\}\}$, $CL(2, 1) \oplus p_3 = \{\{p_1, p_3\}, \{p_2, p_3\}\}$. Note that the sets, $CL(n, m)$, $CL(n{-}1, m)$, and $CL(n{-}1, m{-}1)$ satisfy the following equality:

$$CL(n, m) = CL(n-1, m) \cup (CL(n-1, m-1) \oplus p_n) \tag{1}$$

For example, $CL(n, m) = CL(3, 2) = \{\{p_1, p_2\}, \{p_1, p_3\}, \{p_2, p_3\}\}$, $CL(n{-}1, m) = CL(2, 2) = \{\{p_1, p_2\}\}$, $CL(n{-}1, m{-}1) \oplus p_n = CL(2, 1) \oplus p_n = \{\{p_1\}, \{p_2\}\} \oplus p_3 = \{\{p_1, p_3\}, \{p_2, p_3\}\}$ if $n = 3$ and $m = 2$. Thus, $CL(3, 2) = CL(2, 2) \cup (CL(2, 1) \oplus p_3)$. Eq. (1) tells us that if the sets of $CL(n{-}1, m)$ and $CL(n{-}1, m{-}1)$ are derived, then we can utilize them to construct $CL(n, m)$. This can, in turn, be applied to obtain $CL(n{-}1, m{-}1)$ if the sets of $CL(n{-}2, m{-}2)$ and $CL(n{-}2, m{-}1)$ are known. Following this, we can enumerate all combinations by a recursive method.

For ease of illustration, we denote the incremental set of $CL(n, m)$, namely, $CL(n{-}1, m{-}1) \oplus p_n$, as $\Delta CL(n, m)$. Thus, the equality (1) can be rewritten into the following equality (2):

$$CL(n, m) = CL(n-1, m) \cup \Delta CL(n, m) \tag{2}$$

For instance, $CL(n, m) = CL(3, 2) = CL(2, 2) \cup \Delta CL(3, 2) = \Delta CL(2, 2) \cup \Delta CL(3, 2)$, namely, $\{\{p_1, p_2\}, \{p_1, p_3\}, \{p_2, p_3\}\} = \{\{p_1, p_2\}\} \cup \{\{p_1\}, \{p_2\}\} \oplus p_3$. Note that $CL(2, 2) = \Delta CL(2, 2)$.

Based on Eqs. (1) and (2), we now introduce the *combination sorting algorithm* (CSA). The main idea is to repeatedly apply the two operations: *decomposing* and *re-sorting*. We illustrate this technique by an example shown in Fig. 4.

Example 3. From Fig. 4(a), we easily observe that $CL(2, 2)$ and $CL(2, 1)$ are first obtained. Since the combination $\{p_1, p_2\}$ in $CL(2, 2)$ has the least weight 23, it will be added to the output. Then, the algorithm generates the sorted combinations in $\Delta CL(3, 2)$ (i.e., $\{p_1, p_3\}$ and $\{p_2, p_3\}$) by adding p_3 to $CL(2, 1)$ at the second layer. Thereafter, two parts of the combinations, namely, $CL(2, 2)$ and $\Delta CL(3, 2)$, are re-sorted. Thus, the algorithm obtains three sorted combinations, $\{p_1, p_2\}$, $\{p_1, p_3\}$, and $\{p_2, p_3\}$, which have the weights, 23, 26, and 29, respectively, and are used to generate $CL(3, 2)$ at the third layer. Meanwhile, the combinations in $\Delta CL(3, 2) = \{\{p_1, p_3\}, \{p_2, p_3\}\}$ are outputted since they are in a global order. Similarly, the algorithm obtains $\Delta CL(4, 2) = \{\{p_1, p_4\}, \{p_2, p_4\}, \{p_3, p_4\}\}$. Although the combinations in $\Delta CL(4, 2)$ are sorted locally, their order in the global space (i.e., among all the combinations) might be different. In other words, there might exist another combination (i.e., $\{p_1, p_5\}$ in $\Delta CL(5, 2)$) that has a less weight than that of a combination in $\Delta CL(4, 2)$ (i.e., $\{p_3, p_4\}$). Therefore, we take a global amending step. For example, the algorithm updates the list of $\Delta CL(4, 2)$ since $w(\{p_1, p_5\}) = 35$ is less than $w(\{p_3, p_4\}) = 36$. Thus, the algorithm obtains the updated list of $\Delta CL(4, 2)$, namely, $\Delta CL'(4, 2) = \{\{p_1, p_4\}, \{p_2, p_4\}, \{p_1, p_5\}\}$, in which the combinations are regarded as the next outputted combinations. Following this, the algorithm will terminate until reaching the top layer. Fig. 4(b) shows how to generate the combinations which contain three data objects in a recursive graph. Note that the incremental combinations are in bold font and the updated combinations are marked with an underscore in Fig. 4.

We can draw two conclusions from the above example. (1) Sometimes all combinations in $CL(n, m)$ (or $\Delta CL(n, m)$) are not

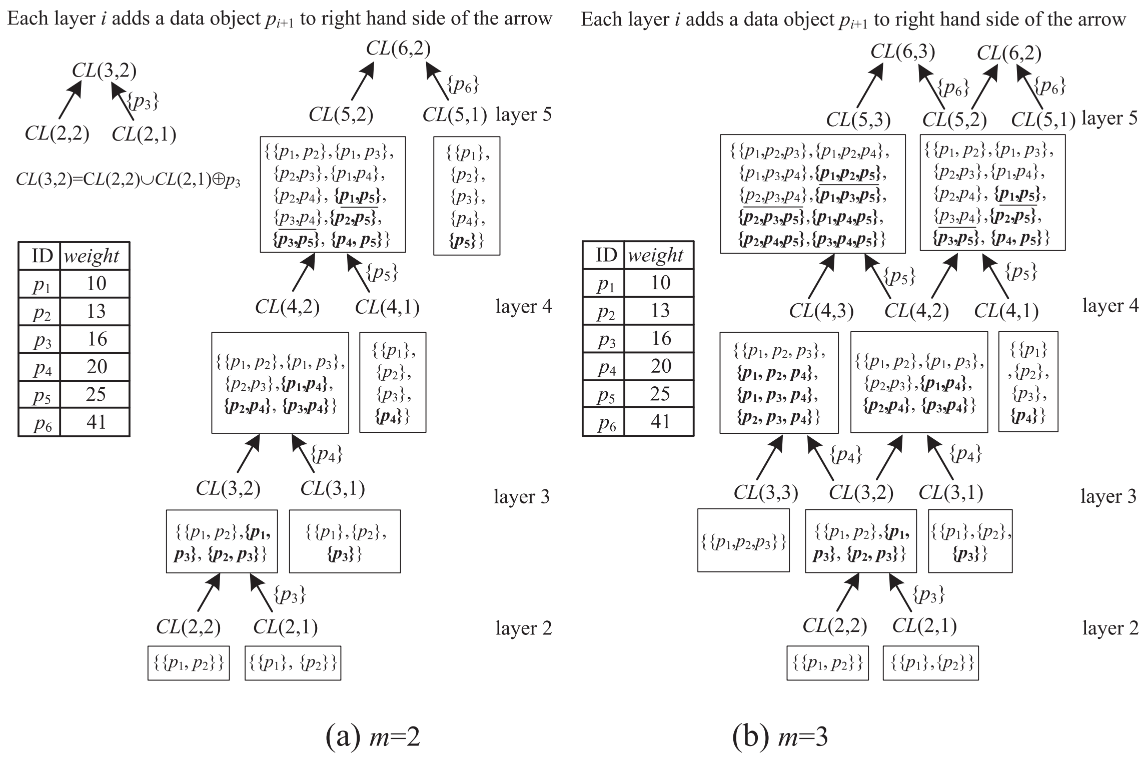

Each layer i adds a data object p_{i+1} to right hand side of the arrow
CL(3,2)
{p3}
CL(2,2) CL(2,1)
CL(3,2)=CL(2,2)∪CL(2,1)⊕p3
ID weight
p1 10
p2 13
p3 16
p4 20
p5 25
p6 41
CL(6,2)
{p6}
CL(5,2) CL(5,1) layer 5
{{p1, p2},{p1, p3}, {p2,p3},{p1,p4}, {p2,p4}, {p1,p5}, {p3,p4},{p2,p5}, {p3,p5}, {p4, p5}}
{{p1}, {p2}, {p3}, {p4}, {p5}}
{p5}
CL(4,2) CL(4,1) layer 4
{{p1, p2},{p1, p3}, {p2,p3},{p1,p4}, {p2,p4}, {p3,p4}}
{{p1}, {p2}, {p3}, {p4}}
{p4}
CL(3,2) CL(3,1) layer 3
{{p1, p2},{p1, p3}, {p2, p3}}
{{p1},{p2}, {p3}}
{p3}
CL(2,2) CL(2,1) layer 2
{{p1, p2}}
{{p1}, {p2}}
(a) m=2
Each layer i adds a data object p_{i+1} to right hand side of the arrow
CL(6,3) CL(6,2)
{p6} {p6}
CL(5,3) CL(5,2) CL(5,1) layer 5
{{p1,p2,p3},{p1,p2,p4}, {p1,p3,p4},{p1,p2,p5}, {p2,p3,p4},{p1,p3,p5}, {p2,p3,p5},{p1,p4,p5}, {p2,p4,p5},{p3,p4,p5}}
{{p1, p2},{p1, p3}, {p2,p3},{p1,p4}, {p2,p4}, {p1,p5}, {p3,p4},{p2,p5}, {p3,p5}, {p4, p5}}
{p5} {p5}
CL(4,3) CL(4,2) CL(4,1) layer 4
{{p1, p2, p3}, {p1, p2, p4}, {p1, p3, p4}, {p2, p3, p4}}
{{p1, p2},{p1, p3}, {p2,p3},{p1,p4}, {p2,p4}, {p3,p4}}
{{p1} ,{p2}, {p3}, {p4}}
{p4} {p4}
CL(3,3) CL(3,2) CL(3,1) layer 3
{{p1,p2,p3}}
{{p1, p2},{p1, p3}, {p2, p3}}
{{p1},{p2}, {p3}}
{p3}
CL(2,2) CL(2,1) layer 2
{{p1, p2}}
{{p1},{p2}}
(b) m=3

Fig. 4. The generated combinations in a recursive graph.

always in a global order. Therefore, we often need to carry up at least a forward recursive operation, i.e., producing incremental combinations of $CL(n+1, m)$; (2) The algorithm can incrementally output $|\Delta CL(m+i, m)|$ ($m \geq 2$) combinations at the ith step if all combinations in $\Delta CL(m+i, m)$ are in a global order. Let $CL'(n, m)$ and $\Delta CL'(n, m)$ be the *updated* lists of $CL(n, m)$ and $\Delta CL(n, m)$, respectively. Obviously, all combinations in $\Delta CL'(n, m)$ (or $CL'(n, m)$) are in a global order since they are updated. Let $LB_w(\Delta CL(n, m))$ and $UB_w(\Delta CL(n, m))$ be the minimum and maximum weights of combinations in $\Delta CL(n, m)$, respectively. For example, $LB_w(\Delta CL(3, 2)) = w(\{p_1, p_3\}) = 26$ and $UB_w(\Delta CL(4, 2)) = w(\{p_3, p_4\}) = 36$ in Fig. 4. Then, we can immediately obtain the following conclusion: the combinations among $\Delta CL(n, m)$ are in a global order if $LB_w(\Delta CL(n+1, m)) \geq UB_w(\Delta CL(n, m))$; otherwise the combinations in $\Delta CL'(n, m)$ is in a global order.

Recall Example 3, the combinations in $\Delta CL(3, 2)$ are in a global order because $LB_w(\Delta CL(4, 2)) = w(\{p_1, p_4\}) = 30$ is larger than $UB_w(\Delta CL(3, 2)) = w(\{p_2, p_3\}) = 29$ according to the above mentioned conclusion. However, the combinations in $\Delta CL(4, 3)$ are not in a global order because $LB_w(\Delta CL(5, 3)) = w(\{p_1, p_2, p_5\}) = 48$ is less than $UB_w(\Delta CL(4, 3)) = w(\{p_2, p_3, p_4\}) = 49$. If we take a global amending step for $\Delta CL(4, 3)$, then we will obtain its updated set, i.e., $\Delta CL'(4, 3) = \{\{p_1, p_2, p_4\}, \{p_1, p_3, p_4\}, \{p_1, p_2, p_5\}\}$. Thus, all incremental combinations in $\Delta CL'(4, 3)$ can be outputted since they are in a global order.

In order to implement the CSA algorithm, we employ m B$^+$-trees, each of which is referred to as the BC$_i$-tree to store the combinations of $CL(j, i)$ ($i \leq j$, $i \in [1, m]$, $j \in [1, n]$). For example, BC$_1$-tree is used to store n objects in $CL(n, 1) = \{\{p_1\}, \{p_2\}, \ldots, \{p_n\}\}$, BC$_2$-tree is used to store all incremental combinations in $\Delta CL(2, 2) = \{\{p_1, p_2\}\}$, $\Delta CL(3, 2) = \{\{p_1, p_3\}, \{p_2, p_3\}\}, \ldots$, and $\Delta CL(n, 2)$ if $m = 2$. The keys of the BC$_i$-tree are defined by the weights of the combinations in $CL(x, i)$ ($x \in [i, n]$, $i \in [1, m]$). Note that although there are an exponential number of combinations, all B$^+$-trees only need to index a small number of combinations since the combinations are generated in an incremental manner and the parameter k of RkGNN query limits the number of generated combinations.

CSA uses a *zig-zag* manner to produce the combinations by a recursive method and will terminate the procedure once it has obtained at least k combinations. CSA generates the combinations in an incremental manner. Each time it uses *pdelta_maxkey* and *cdelta_minkey* to record the maximum weight of previous incremental combinations in $\Delta CL(x-1, y)$ (or $\Delta CL'(x-1, y)$) and the minimum weight of current incremental combinations in $\Delta CL(x, y)$, respectively. It inserts all incremental combinations of $\Delta CL(x, y)$ into the corresponding B$^+$-tree (i.e., BC$_y$_tree). After that, the algorithm updates the incremental combinations in $\Delta CL(x, y)$, which are not in a global order. Specifically, in order to obtain the updated set $\Delta CL'(x, y)$, the algorithm forwards λ steps until the following inequality (3) no longer holds:

$$LB_w(\Delta CL(x+\lambda, y)) \leq pdelta\_maxkey \tag{3}$$

In each step, the algorithm only inserts the combinations $G_i \in \Delta CL(x+\lambda, y)$, which have a less weight than *pdelta_maxkey* and are not in the BC$_y$-tree. Whenever the incremental combinations of the previous step, i.e., $\Delta CL'(x, m)$, are produced, the algorithm will output them.

Although, the method proposed in Zhang, Chee, Mondal, Tung, & Kitsuregawa (2009) is similar to CSA for generating the combinations of objects, it cannot be used to solve our problem because of two differences: (1) CSA generates a set of combinations which contain y data objects, i.e., $\Delta CL(x, y)$, by the equality (1), whereas their method enumerates the candidate item sets of length y (i.e., φ_y) from item sets of length $y-1$ (i.e., ϕ_{y-1}) using priori algorithm where the synthetic formulae is $\varphi_y = \{a \cup \{b\} | a \in \phi_{y-1} \wedge b \in \cup\phi_{y-1} \wedge b \notin a\}$. (2) CSA products the combinations in an incremental manner; however their method is not the case although it also adopts a recursive fashion.

4.2. Sorting and threshold pruning

Since the MRkGN query is only interested in the top k combinations rather than all the combinations that have q as their GNN, we propose the *sorting and threshold pruning* (STP) method that can stop the search much earlier without examining all the combinations.

In the following discussion, we use *delta_mindist* to denote the minimum aggregated distance of currently output incremental combinations. Note that if $\Delta CL(x, y)$ has been updated to $\Delta CL'(x, y)$, the newly updated weight will be assigned to *delta_mindist*. For example, *delta_mindist* of $\Delta CL(3, 2)$ and $\Delta CL'(5, 2)$ in Fig. 4(a) are $w(\{p_1, p_3\})$ and $w(\{p_3, p_4\})$, respectively. Similarly, *delta_mindist* of $\Delta CL(5, 3)$ in Fig. 4(b) is $w(\{p_2, p_3, p_4\}) = 49$ rather than $w(\{p_1, p_2, p_5\}) = 48$ because $\{p_2, p_3, p_4\}$ in $\Delta CL(4, 3)$ is swapped with $\{p_1, p_2, p_5\}$ in $\Delta CL(5, 3)$. If a candidate combination contains at least an intermediate entry, then the combination will be inserted into a refined set, called G_{rfn}. These combinations among G_{rfn} will be expanded when their corresponding entries are visited. Let *rfn_mindist* be the minimum weight of the combinations among G_{rfn}. Then, we immediately derive Theorem 1 and Corollary 1 as follows.

Theorem 1. *The RkGNN query can terminate the search if delta_mindist and rfn_mindist are both larger than best_kdist.*

Proof. Since *rfn_mindist* is larger than *best_kdist*, there is not any other combination which has a smaller aggregate weight than *best_kdist*. Thus, any combination in G_{rfn} is not the result of the R*k*GNN query. Meanwhile, there is not any other new combination which has a smaller aggregate weight than *best_kdist* due to *delta_mindist* $\geq$ *best_kdist*. □

By Theorem 1, we can easily obtain the following Corollary 1.

Corollary 1. *The search of RkGNN stops if delta_mindist is larger than best_kdist when the set of* G_{rfn} *is empty.*

We omit the proof of Corollary 1 here because it is very straightforward. The R*k*GNN query processes each combination in an ascending order according to its weight since the algorithm enumerates all new combinations in an incremental manner. Therefore, the current enumeration can be terminated based on the following Theorem 2.

Theorem 2. *The RkGNN query does not have to process the rest of combinations for each enumeration if the current combination G has a weight larger than best_kdist,* i.e., *adist(q, G)* $\geq$ *best_kdist.*

Proof. According to CSA, since all new combinations are generated in an ascending order of their weights for each round of enumeration, the combinations after the current combination *G* must have a larger weight. Thus, there will not be any another combination which has a larger weight than *G* due to *adist*(*q*, *G*) $\geq$ *best_kdist*. □

According to Theorem 2, we can immediately derive the following Corollary 2.

Corollary 2. *For the current combination G, it can be pruned if adist(q, G)* $\geq$ *best_kdist.*

We omit the proof of Corollary 2 too since it is also straightforward according to Theorem 2.

Consider the example in Fig. 5 ($m = 2$, $k = 4$). The aggregated distance of $\{p_1, p_4\}$ is less than that of $\{p_2, p_5\}$ owing to $dist(q, p_1)+dist(q, p_4) = 3.16+4.47 = 7.63$ and $dist(q, p_2)+dist(q, p_5) = 3.61+5 = 8.61$. Thus, $\{p_2, p_5\}$ can be pruned because *best_kdist* is the aggregated distance of $\{p_1, p_4\}$.

Theorem 1–2 and Corollary 1–2 are very effective in reducing the search space and saving a large amount of unnecessary computations. For instance, our experiments show that only about 0.1% combinations need to be processed for each enumeration, which means the STP method helps reduce 99.9% computations. Note that the STP method can also be adopted in the BR*k*GN query.

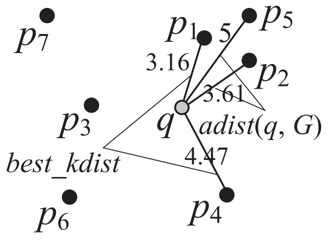


Fig. 5. Illustration of how to prune the current combination.

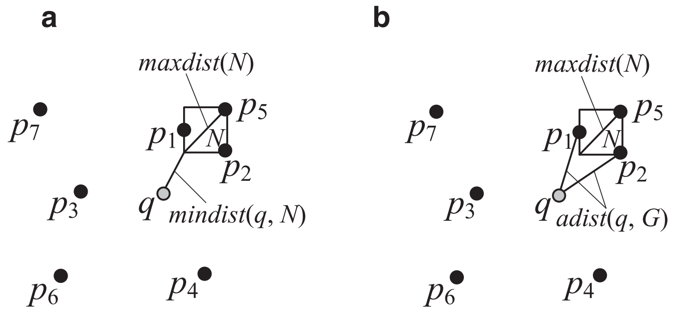


Fig. 6. Illustrate the pruning heuristics. (a) pruning a node; (b) pruning a combination.

4.3. MBR property pruning

Besides the STP method, we also take advantages of the property of the MBR (minimum bounding rectangle) in the R-tree to further improve the pruning power. We propose a novel and effective pruning method, namely *MBR Property Pruning* (MP). Specially, let *maxdist*(*N*) be the distance between the lower left and upper right of the MBR of the current node *N*, and let *mindist(q, N)* be the minimum distance between the node *N* and *q*. Then, we derive Lemma 1 as follows.

Lemma 1 (Monochoromatic MBR Property Pruning)**.** *Any combinations among a node N cannot be the result of MRGN query if the node N contains at least m+1 data objects, and maxdist(N)≤ mindist(q, N).*

Proof. If any combination *G* from current node *N* contains *m* data objects, we have *mindist*(*q*, *N*)**m* $\leq$ *adist*(*q*, *G*). Assume that there is another point p' in *N* which is not contained in *G*. Obviously, the sum of distances between p' and data objects in *G*, *adist*(p', *G*), is less than or equal to *maxdist*(*N*)**m*. Thus, *adist*(p', *G*) $\leq$ *maxdist*(*N*)**m* $\leq$ *mindist*(*q*, *N*)**m* $\leq$ *adist*(*q*, *G*). Then, we have *adist*(p', *G*) $\leq$ *adist*(*q*, *G*). This indicates that *q* is not the GNN of *G*. Since this is a general assumption, Lemma 1 is correct. □

Example 4. The basic idea of the *MP* is illustrated in Fig. 6. As shown in Fig. 6(a), the MBR of node *N* contains three data objects, p_1, p_2, and p_5. If $m = 2$, the node *N* will produce three combinations, $\{p_1, p_2\}$, $\{p_1, p_5\}$, and $\{p_2, p_5\}$. Since maxdist(*N*) $\leq$ mindist(*q*, *N*), the three combinations of *N* are not the results of the *MRkGN query.*

Lemma 1 works well when the MBR is not too large. In case that there are some large MBRs, we will utilize the following tighter condition (equality (4)) to check if the current combination *G* can be pruned:

$$\mathrm{maxdist}(N) * m \leq adist(q, G) \tag{4}$$

Consider the combination $G_1 = \{p_1, p_2\}$ of node *N* in Fig. 6(b), where the parameter *m* is 2. G_1 is not the result of MRGN query due to *maxdist*(*N*)*2 $\leq$ *adist*(*q*, G_1).

Clearly, Lemma 1 assumes that the data objects of all combinations are from the same node. Often this is not the case because

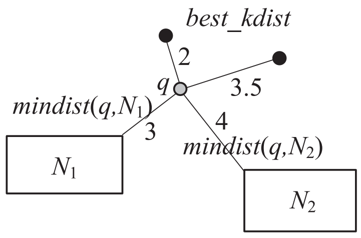


Fig. 7. Illustration of pruning the combinations from two nodes.

the data objects of a combination may come from multiple nodes of the R-tree. In such case, we may still prune these combinations according to Corollary 2. At this moment, we need to take advantage of the characteristics of minimum distances of nodes. Consider the following example in Fig. 7, where the distances of q from N_1 and N_2 are 3 and 4, respectively, and the parameter m is 2. The best distance, *best_kdist*, is 2+3.5 = 5.5. Obviously, $mindist(q,\ N_1)*1+mindist(q,\ N_2)*(2-1) = 3+4 = 7$ is less $adist(q,\ G)$ and meanwhile it is greater *best_kdist* = 5.5. So, $adist(q,\ G) \geq$ *best_kdist*. In other words, any combination from N_1 and N_2 cannot be the results of MRGN query.

The MP method improves the query performance significantly by avoiding a large number of unnecessary computations of GNNs.

4.4. Window pruning

The previous MP method relies on the existence of the MBRs to compute the pruning conditions. In this subsection, we propose a more generic pruning method which is called *window pruning* (WP). We illustrate the intuition of our window pruning method in Fig. 8.

Specifically, let q be a query object and G be a candidate combination in the dataset P. We define the *window region* of an object p_i as a circle centered at point p_i with radius $dist(q,\ p_i)$, and denote it as $WR(q,\ p_i)$. For any G, its window region is defined as the union of the window regions of all the objects in G, i.e., $WR_\cup(q,\ G) = WR(q,\ p_1) \cup WR(q,\ p_2) \ldots \cup WR(q,\ p_m)$. Let $WR_\cap(q,\ G)$ be the intersection of the window regions of all objects in G, namely, $WR_\cap(q,\ G) = WR(q,\ p_1) \cap WR(q,\ p_2) \ldots \cap WR(q,\ p_m)$. Let $WR_\setminus(q,\ G)$ be the difference of two sets $WR_\cup(q,G)$ and $WR_\cap(q,\ G)$, namely, $WR_\setminus(q,\ G) = WR_\cup(q,G) - WR_\cap(q,\ G)$. Let $P \setminus G$ be the difference of P and G, namely, $P \setminus G = P - G$. Then, we derive the following Theorem 3 and Corollary 3, immediately.

Theorem 3 (Monochoromatic Window Pruning**).** *The candidate combination $G \subset P$ ($|G| = m$) is not the RGNN of the query object q if $\exists p' \in P \setminus G \wedge p' \in WR_\cap(q,\ G)$. G is the RGNN of q if $\neg \exists p' \in P \setminus G \wedge p' \notin WR_\cup(q,G)$.*

Proof. For any object $p' \in P \setminus G$, we prove Theorem 3 by analyzing the following two cases. □

Case 1. $p' \in WR_\cap(q,\ G)$. Consider the case in Fig. 8(a), since $dist(p_5,\ p_1) < dist(q,\ p_1)$ and $dist(p_5,\ p_2) < dist(q,\ p_2)$, we have $adist(p_5,\ \{p_1,\ p_2\}) < adist(q,\ \{p_1,\ p_2\})$. Therefore, G is not the RGNN of q.

Case 2: $p' \notin WR_\cup(q,G)$. Consider the case in Fig. 8(b), since there are not any object $p' \notin WR_\cup(q,G)$, we have $dist(p,\ p_3) > dist(q,\ p_3)$ and $dist(p,\ p_4) > dist(q,\ p_4)$. In other words, q is the GNN of the combination $G = \{p_3,\ p_4\}$. So, G is the RGNN of q.

Based on the above two cases, we have Theorem 3 derived. □

Corollary 3. *The MRGN query only needs to search the objects among $WR_\setminus(q,\ G)$ to judge whether or not G has q as its GNN if $\forall p' \in P \setminus G \wedge p' \notin WR_\cap(q,\ G)$ and $\exists p'' \in P \setminus G \wedge p'' \in WR_\setminus(q,\ G)$.*

Proof. Consider the case in Fig. 8(c), the whole search space consists of three regions, i.e., $R_1 = WR_\cap(q,\ G)$, $R_2 = WR_\setminus(q,\ G)$ and $R_3 = \neg WR_\cup(q,\ G)$. According to case 1 of Theorem 3, the algorithm still needs to search the rest of the space (i.e., R_2 and R_3) to judge whether G has q as its GNN if $\forall p' \in P \setminus G \wedge p' \notin R_1$. In addition, in terms of the case 2 of Theorem 3, $\forall p'''$ which satisfies $p''' \in P \setminus G \wedge p''' \in R_3$ cannot become the GNN of G. In other words, it is not necessary to search R_3 to judge whether G has q as its GNN. Thus, the algorithm only needs to search the rest of the data space, i.e., R_2. □

Based on Theorem 3 and Corollary 3, we give the *window pruning method* (WPM) as Algorithm 1, which can be seamlessly integrated into our RkGNN query. At the beginning, WPM initializes the distance of best GNN of current combination G found so far (i.e., *best_dist*) by $adist(q,\ G)$, and best GNN of G (i.e., *best_NN*) by q, respectively. Then, line 2 obtains $WR_\cup(q,G)$ and $WR_\cap(q,\ G)$ by carrying out a window query for each pair of $p_i \in G$ ($|G| = m$) and q; meanwhile saves the corresponding data objects into an union set *uSet* and an intersect set *iSet*, respectively. If the number of data objects in *uSet* is zero (namely, $|uSet| = 0$), G is marked as a RGNN of query object q (line 3). If the set of *iSet* is not empty (i.e., $|iSet|>0$), WPM sets *best_NN* to empty to represent G is not a RGNN of q (line 4). If $\exists p \in WR_\cup(q,G) \wedge p \notin G \wedge p \notin WR_\cap(q,\ G)$ (lines 5–10), WPM computes the aggregated distance for each pair of $p \in WR_\cup(q,G)$ and G, namely, $pG\_dist = adist(p,\ G)$ to obtain the *best_NN*. At last, WPM can judge whether G is a result of RkGNN query by the following method. If *best_NN* is q, G is a RGNN of q; otherwise, G is not a RGNN of q.

Theorem 3 and Corollary 3 enable the MRkGN algorithm to reduce the search space when it cannot quickly judge whether G has q as its GNN. On the other hand, they guarantee that our algorithm does not have to invoke MBM in Papadias et al. (2004) for the GNN query. It is worth noting that the saved cost come from the WP method cannot be dominated by the cost for performing WPM although WPM needs to perform multiple window queries (i.e., m window queries) for each combination. This is because, in order to judge whether the current combination G has q as its GNN, (i) a window query generally has a smaller search space than

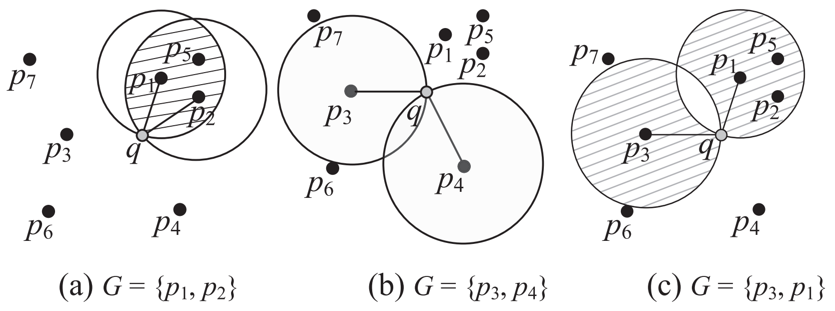


Fig. 8. Illustration of window pruning method in MRkGN query ($m = 2$).

MBM since MBM prunes data objects (or nodes) in the whole data space, whereas WPM only need to search a smaller limited space, i.e., $WR(q, G)$; (ii) the use of CSA and BF retrieval makes that a window query in WPM has a smaller radius. In other words, $WR(q, p_i \in G)$ generally contains a few of data objects; (iii) a substantial proportion of the window queries fall in Theorem 3 as shown in our experiments; (iv) the *inclusive* property of CSA (i.e., $CL(3, 2) \oplus p_4 \subset CL(4, 3)$) makes that a *father* combination (i.e., $\{p_1, p_3, p_4\}$ in $CL(4, 3)$) might usually be a result of the R*k*GNN query if a *child* combination (i.e., $\{p_1, p_3\}$ in $CL(3, 2)$) is a answer. In other words, *m* window queries are almost no additional cost relative to a single window query.

Algorithm 1 Window pruning method (*root, q, G, m*)

```
Input: The root of R-tree index, a query object q, the current combination G, a user specified integer m.
Output: the GNN of G, namely, best_NN
1.  best_dist = adist(q, G), best_NN = q;
2.  uSet← $WR_\cup(q,G)$, iSet← $WR_\cap(q, G)$; //by m window queries
3.  if (|uSet| = 0) return best_NN; // G is the RGNN of q if $\neg\exists p' \in P\backslash G \wedge p' \notin WR_\cup(q,G)$
4.  else if (|iSet|>0) best_NN←∅, return best_NN; // G is not a RGNN of q if $\exists p \in WR_\cap(q, G)$
5.  else // $\exists p \in WR_\cup(q,G) \wedge p \notin G \wedge p \notin WR_\cap(q, G)$
6.      for each object $p \in uSet$ do
7.          pG_dist ← adist(p, G); //compute the aggregated distance between p and G
8.          if (pG_dist < best_dist)
9.              best_dist ← pG_dist, best_NN ← p;
10.     return best_NN;
```

4.5. Lazy MR*k*GN query processing

We now introduce the technique, the *lazy outputting* of combinations, called LO, and then present a lazy MR*k*GN query algorithm that integrates all the proposed techniques.

Unlike the simple solution that immediately enumerate the combinations for each popped object, LO method starts to enumerate combinations until there are enough data objects so as to generate at least *k* orderly combinations in a global space. At the other hand, LO approach also avoids accumulating too many data objects to give free play to the pruning capacity of other pruning heuristics. Next, we will give the lazy MR*k*GN query algorithm.

In particular, we search the R-tree index in a *best-first* manner (Hjaltason & Samet, 1999). We maintain a minimum heap *H* to store the entries in the form of (*e, key*), where *e* contains the MBR of the node in the R-tree, and the *key* is the minimum Euclidean distance between *e* and *q*. By enumerating the combinations and the aforementioned pruning heuristics, the algorithm can quickly finish the MR*k*GN computation. We use a temporary min-heap H_t which is used to generate the first *k* incremental combinations, to store the data objects popped from *H*. In addition, we make use of another min-heap H_a to store all the data objects in H_t and the rest of data objects or entries in *H*. In order to produce at least *k* combinations and meanwhile avoid generating too many combinations, we take advantage of the following inequality (5) to limit the number of entries in H_t:

$$C_{|H_t|}^{m} \le k \le C_{|H_t|+1}^{m}, \; |H_t| \ge m+1 \tag{5}$$

where $|H_t|$ represents the size of H_t, i.e., the number of data objects in H_t, and $C_{|H_t|}^{m}$ ($C_{|H_t|+1}^{m}$) is the number of combinations selecting *m* objects from $|H_t|$ ($|H_t|$+1) objects, respectively.

The detailed description of lazy MR*k*GN query is given in Algorithm 2. At the beginning, we insert the root (*root*(*I*)) of the R-tree into heap *H* (line 2). Then, each time we remove an entry (*e, key*) from heap *H* (line 4) and process *e* until *H* becomes empty (lines 5∼27). Lines 5–7 check whether $delta\_mindist \ge best\_kdist$ and the refined set G_{rfn} is empty, or $delta\_mindist \ge best\_kdist$ and $rfn\_mindist \ge best\_kdist$. If so, the MR*k*GN query will be terminated by Theorem 1 and Corollary 1. Otherwise, the algorithm will process the current entry *e* as follows.

If the current entry *e* is a data object, we compute whether the number of current group of incremental combinations exceeds *k*. If not, the algorithm continues to pop the top entry of *H* (lines 9–10). Otherwise, line 11 takes advantage of CSA to generate the current group of incremental combinations, using the entries in H_a in an ascending order. This guarantees that the algorithm can quickly obtain the first *k* query results. For the current combination *G*, if it is marked as a *false positive*, the algorithm will discard it (line 12). If $adist(q, G) \ge best\_kdist$, our method will exit the current round and fetch next entry of *H* by taking advantage of Theorem 2 (line 13); otherwise, we process it as follows. If it contains at least an intermediate entry e', it is inserted into the refined set G_{rfn} so that it can be processed afterwards (line 16); If *G* only has data objects (line 14), a GNN query using MBM (Papadias et al., 2005) or window pruning method (WPM) is invoked to judge whether *q* is its GNN by Theorem 3 and Corollary 3. When the answer is true, the result set G_{rlt} is updated by the procedure of *UpdateRlt* (line 15).

Algorithm 2 Lazy_MR*k*GN(*root, P, q, m, k*)

```
Input: The root of R-tree index I constructed over P, a query object q, a user specified integer m, and a parameter k
Output: the combinations which have q as their GNNs
1.  $G_{rlt}$ = , $G_{rfn}$ = , H = , best_kdist = +;
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 (delta_mindistbest_kdist) //Theorem 1 and Corollary 1
6.          if ($G_{rfn}$ is empty) return;
7.          if ($G_{rfn}$ isn't empty and rfn_mindistbest_kdist) return;
8.      if (e is a data object)
9.          if (the number of incremental combinations does not exceeds k)
10.             continue; //LO method
11.         for each incremental combination G of current group do
            //in ascending order of adist(q, G), by CSA
12.             if (G is marked as false positive) continue;
13.             if (adist(q, G) best_kdist) break; //Theorem 2
14.             if (G consists of data objects) //by Theorem 3 and Corollary 3
15.                 if (q is the GNN of G) UpdateRlt(G, $G_{rlt}$, k, best_kdist)
16.             else insert G into $G_{rfn}$;
17.     else //leaf node or intermediate node
18.         for each data object or entry $e_i$ e do
19.             insert $e_i$ of e into heap H;
20.         mark combination G of e pruned by Lemma 1 as false positive;
21.         for combination G of $G_{rfn}$ including e do
22.             remove G from $G_{rfn}$;
23.             for each expanded combination G of G do //replace e of G using $e_i$
24.                 if (adist(q, G)best_kdist) continue; //Corollary 2
25.                 if (G consists of data objects) //by Theorem 3, Corollary 3
26.                     if (q is the GNN of G) UpdateRlt(G, $G_{rlt}$, k, best_kdist);
27.                 else insert G into $G_{rfn}$;
```

If the current entry *e* is an intermediate entry, we insert each child e_i of *e* into *H*, and maintain the content of H_a (lines 18–19). Then, the algorithm uses Lemma 1 to prune the combinations of *e*, and marks those being pruned as *false positives* (line 20). At last, the algorithm updates the corresponding combinations of G_{rfn} that contains the current entry *e* by replacing *e* in each combination with each child e_i. Thus, each combination *G* will produce |*e*| updated combinations, each of which is denoted as G', where |*e*| is the number of children of *e*. Line 22 removes *G* from G_{rfn}. Subsequently, we process each combination G' at lines 25–27, in a similar way as lines 14–16. However, note that we cannot apply Theorem 2 for line 24 because the algorithm does not sort all combinations in G_{rfn}. If we sort these combinations, it will introduce too much overhead. Next, the procedure *UpdateRlt* is used to update the results set G_{rlt} and the variable *best_kdist* when the MR*k*GN query obtains a new query result.

In what follows, we prove the correctness of the algorithm.

Lemma 2. *The lazy MRkGN query algorithm will generate all combinations.*

Proof. Since the algorithm will pop each entry in H and expands the entry if it is an intermediate entry, the algorithm will process each object eventually. Specifically, each time the algorithm produces all incremental combinations of the currently popped data object using the data objects/entries in H_a. The combinations that contain intermediate entry/entries will be inserted into the refined set G_{rfn}. If the corresponding entry is popped, the combination(s) will be expanded. The updated combinations will again be expanded if they contain intermediate entry/entries. Therefore, Lemma 2 is correct. □

By integrating Lemma 1–2, Theorem 1–3 and Corollary 1–2, we have the following Theorem 4.

Theorem 4. *The lazy MRkGN query algorithm correctly returns all query results and does not produce any false positives or false negatives.*

Proof. By Lemma 2, we know that the query algorithm will enumerate all combinations and check whether q is their GNN. Clearly, this procedure does not produce false negatives. In addition, Lemma 1, Theorem 1–3 and Corollary 1–2 guarantee the correctness of pruning the combinations and do not produce false positives. □

4.6. Discussion

Although we have developed many pruning heuristics for the RkGNN query, there may be room for further improvement. At present, there are many research effects on parallel computation over massive data or cloud data (Fu et al., 2015; Xia, Wang, Sun, & Wang, 2015). In fact, we can also take the parallelization of the proposed techniques into consideration, i.e., using MapReduce (Park, Min, & Shim, 2013; Zhou, Wu, Chen, & Shou, 2014) and extending to multi-cores. Since it is our future work, we only provide some basic ideas of parallelization for the RkGNN query as follows.

(1) For the multi-core CPU, we can develop a multi-thread procedure to compute the query results of RkGNN by using multiple threads. For example, a main thread incrementally produces combinations and allocates these combinations to other sub-threads. Each sub-thread receives a part of combinations and then determines whether q is the GNN of each combination G by MBM or WPM in parallel independently. At last, each sub-thread sends its results to the main thread. Whenever the main thread has collected all results from the sub-threads, it will continue to enumerate the rest combinations until it has obtained the top-k results of RkGNN query.

(2) For the processing based on MapReduce, a master function incrementally produces the combinations by invoking the CSA, where the key is the distance between the point p and the query point q. In addition, the master function also takes charge of allocating the combinations with some necessary data points to some map functions and receiving the query results from a reduce function. When there are at least k combinations, it processes these combinations by MapReduce. When it has obtained the top-k query results of RkGNN, it will notify all map functions and the reduce function so that they can stop the computation. Each map function judges if q is the GNN of each combination G by WPM method. If it is yes, the form of output is $\langle G, 1\rangle$; otherwise, $\langle G, 0\rangle$. At last, a reduce function obtains all combinations which have the form of $\langle G, 1\rangle$ from the map functions and sends them the master function.

Table 2
The content of heap H and reuse heap H_r during the procedure of MRkGN query.

Action	Heap content(H)	Reuse heap content (H_r)
Access *root*	N_1, N_2	N_1, N_2
Expand N_1	N_2, N_4, N_3	N_2, N_4, N_3
Expand N_2	N_4, N_5, N_3, N_6	N_4, N_5, N_3, N_6
Expand N_4	p_1, N_5, p_3, N_3, N_6, p_5	p_1, N_5, p_3, N_3, N_6, p_5
Access p_1	N_5, p_3, N_3, N_6, p_5	p_1, N_5, p_3, N_3, N_6, p_5
Expand N_5	p_2, p_3, N_3, N_6, p_5, p_8	p_1, p_2, p_3, N_3, N_6, p_5, p_8
Access p_2	p_3, N_3, N_6, p_5, p_8	p_1, p_2, p_3, N_3, N_6, p_5, p_8
Access p_3	N_3, N_6, p_5	p_1, p_2, p_3, N_3, N_6, p_5

5. Improving query performance by reducing redundant I/O accesses

We observe that the lazy MRkGN algorithm needs to traverse R-tree multiple times to generate combinations and find GNNs for the candidate combinations. There are many redundant I/O accesses during the query processing. For example, consider the query in Fig. 3, where $m = 2$ and $k = 2$. According to Eq. (5), we have $|H_t| = 3$, and hence the algorithm sequentially outputs three combinations, $\{p_1, p_2\}$, $\{p_1, p_3\}$ and $\{p_2, p_3\}$. By utilizing the Corollary 3, the algorithm obtains the first result of MR2GN query, i.e., $\{p_1, p_2\}$. The combination $\{p_1, p_3\}$ is pruned by Theorem 3 since $p_5 \in WR_{\cap}(q,\{p_1, p_3\})$. Assume that each tree node is stored on one disk page, the algorithm requires 4 I/O accesses to visit the nodes, N_2, N_5, N_1, and N_4 when generating the combination $\{p_2, p_3\}$. Similarly, the algorithm still needs to visit the node N_6 two times for the window queries of $WR(q, p_2)$ and $WR(q, p_3)$ based on Theorem 3. Then, the algorithm obtains the second result of MR2GN query, namely, $\{p_2, p_3\}$.

In order to further improve the query performance by avoiding repeated visits to the same tree nodes, we take advantage of the reuse heap method (RH) in Jiang et al. (2014). Specifically, we use a reuse heap H_r to store the entries that have been accessed. Algorithm 2 can be revised by inserting a line code before line 20 to maintain the entries in H_r. The code inserts all children of current entry e into H_r and deletes e from H_r. As a result, the algorithm will no longer incur extra I/O cost if the entry being accessed can be found in H_r. Reconsider the example in Fig. 3, our approach will decrease 10 I/O accesses by using the information stored in the reuse heap H_r. Table 2 gives the content of H and H_r during the procedure of the query.

In our reuse heap method, we do not expand the corresponding entry to the leaf level that stores data objects because otherwise, the heap may grow too large resulting in too much maintenance cost. Moreover, we leverage the binary search to speed up the information update in the heap. In addition, it is worth noting that our proposed reusing heap technique is different from caching techniques adopted in most DBMS. The caching typically keeps the recent entries, whereas our reusing heap technique preserves the specific entries which are not changed during the search.

Based on the above analysis, we can easily obtain the following Theorem 5.

Theorem 5. *Given a MRkGN query, the reuse heap method ensures that each intermediate index node will be accessed at most once.*

Proof. Since each visited node will be stored in the reuse heap, the query algorithm based on the reuse heap will not produce any I/O access when processing the GNN query. □

6. Bichoromatic reverse top-*k* group nearest neighbor (BRkGN) query

In this section, we first discuss how to adapt the MR*k*GN's pruning technique for the BR*k*GN query, and then present the BR*k*GN query algorithm.

6.1. Differences between MRkGN and BRkGN

Although the BR*k*GN query involves two data sets and hence more complex, some MR*k*GN's pruning techniques can still be modified to be adopted by BR*k*GN query.

Firstly, the pruning methods in Section 4.2 can be directly adopted by BR*k*GN. In particular, we check whether or not a query object q is the GNN of a combination G. If so, G will be inserted into the result set G_{rlt}; otherwise, G can be safely pruned.

Secondly, we can also develop the bichromatic MBR property and window pruning methods for BR*k*GN as follows.

Lemma 3 (Bichromatic MBR Property Pruning)**.** *Any combinations among N cannot be the result of BRGN query if* $maxdist(N) \le mindist(q, N) \wedge p' \in B \wedge p' \in BR(N)$ *and the node N contains at least m data objects, where BR(N) denotes the boundary region of MBR of node N.*

Theorem 6 (Bichromatic Window Pruning)**.** *The candidate combination* $G \subset A$ ($|G| = m$) *is not the RGNN of query object q if* $\exists p' \in B \wedge p' \in WR_{\cap}(q, G)$. *G is the RGNN of q if* $\neg \exists p' \in B \wedge p' \notin WR_{\cup}(q,G)$.

Corollary 4. *The BRGN query only needs to search the objects among* $WR_{\setminus}(q, G)$ *over dataset B to judge whether or not G has q as its GNN if* $\forall p' \in B \wedge p' \notin WR_{\cap}(q, G)$ *and* $\exists p'' \in B \wedge p'' \in WR_{\setminus}(q, G)$.

Note that the MBR property pruning technique used by the BR*k*GN query is a little different from that used by the MR*k*GN query. The MR*k*GN query may use the information of data objects from the same index as the combination G to prune itself. However, the BR*k*GN query must utilize the information of data objects from another index over dataset B to prune G. In addition, the BR*k*GN query needs to traverse the index of dataset B (not dataset A) for the bichromatic window pruning method.

6.2. BRkGN query processing

In this subsection, we integrate the pruning heuristics and the reuse technology into the procedure of the BR*k*GN query. Consider a query object q and two data sets, A and B. We assume that the data sets A and B are indexed by two R-trees, T_A and T_B, respectively. The BR*k*GN query algorithm conducts two main tasks: (1) incrementally producing combinations Gs in T_A to obtain the candidate sets of possible query results, and (2) verifying each set G in the candidate sets whether or not it has q as its GNN in data set $B' = (B \cup \{q\})$. The overall query processing for BR*k*GN is similar to MR*k*GN. Therefore, we only present mainly the differences between the two query algorithms in the following:

(1) BR*k*GN traverses T_A in a BF manner according to the minimum distance between the current node of T_A and q, and produces the combinations Gs incrementally by the method in Section 4.1. Meanwhile, BR*k*GN prunes unqualified combinations by sorting and threshold pruning techniques mentioned in Section 4.2.

(2) For each current combination G, BR*k*GN needs to traverse T_B. BR*k*GN first uses the bichromatic MBR property pruning method (i.e., Lemma 3) to prune G. If G cannot be pruned, the algorithm will further take advantage of bichromatic window pruning method (e.g., Theorem 6 and Corollary 4) to prune G.

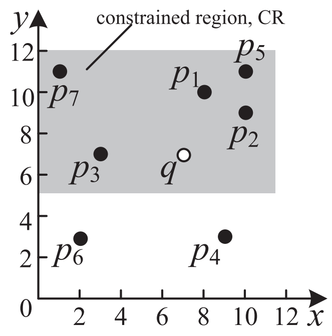


Fig. 9. Illustration of the constrained reverse group nearest neighbor.

(3) Similary, BR*k*GN also utilizes the reuse heap method to reduce the redundant I/O accesses and saves all entries in T_B into the reuse heap H_r.

7. Constrained reverse top-*k* group nearest neighbor query

In some scenarios, users may have additional constraints (e.g., distance, spatial region, etc.) on R*k*GNN queries. For example, a supermarket chain company may want to specify a residence area and select new branches within this area. To handle such cases, we define a variant of the R*k*GNN query, namely, *constrained reverse top-k group nearest neighbor* (CR*k*GN) query, which computes the reverse top-*k* group nearest neighbor in a specified region.

If only the top k combinations are returned to the user according to the sum distance of q and each object $p \in G$, we denote this query over a single dataset as *monochoromatic constrained reverse top-k group nearest neighbor* query and over two datasets as *bichromatic constrained reverse top-k group nearest neighbor* query.

Consider the example in Fig. 9. When $m = 2$ and $k = 2$, the combinations of $\{p_1, p_3\}$ and $\{p_2, p_3\}$ constitute the constrained top-*k* group nearest neighbor. Note that $\{p_1, p_4\}$ and $\{p_2, p_4\}$ are not the constrained reverse top-*k* group nearest neighbors of q although they are the reverse top-*k* group nearest neighbors of q. This is because the point p_4 is not located inside the constrained region.

To efficiently answer this new type of query, we propose to integrate the checking of additional conditions (i.e., constrained region, CR) into the execution of the regular R*k*GNN query. The main ideas include (i) discarding all combinations which contain the data objects or entries outside the constrained region; (2) only using the data objects inside the constrained region to compute the GNN of the current combination G. Therefore, for monochromatic and bichromatic cases, we only need to process the entries in index (or indexes) which intersecting with the constrained region. Since our proposed algorithm for answering CR*k*GN is similar to R*k*GNN query processing algorithms, we omit the details here.

8. Experimental evaluation

In this section, we evaluate the performance of our proposed MR*k*GN query algorithm and BR*k*GN query algorithm. We conduct some additional experiments for the previous approaches in Jiang et al. (2013), and provide a more comprehensive analysis and discussion.

8.1. Experimental setup

In our experiments, we use both real and synthetic datasets. The two real datasets, namely, *PP* and *NE*, are from *www.rtreepotral.org*. Specifically, *PP* consists of the populated places of the North America with 24,493 points. *NE* represents

Table 3
Parameter ranges and default values.

Parameter	Range	Default
k for MRkGN	10, 20, 30, 40, 50	30
k for BRkGN	5, 10, 15, 20, 25	15
m	2, 3, 4, 5	3
N (cardinality)	20 K, 40 K, 60 K, 80 K, 100 K or 3 K, 6 K, 9 K, 12 K, 15 K for *PP*	60 K, or 9 K for *PP*
CR (% of the space)	4, 8, 16, 32, 64	16

three metropolitan areas (New York, Philadelphia, and Boston) containing 123,593 postal addresses (points). We also generate *Independent* (*IN*) and *Correlated* (*CO*) datasets with dimensionality $dim = 2$ and cardinality N in the range [20 K, 100 K] (*PP* is in the range [3 K, 15 K]). Specifically, *IN* consists of random points from the unit square. *CO* follows correlated distributions. All datasets is normalized to range [0, 1]. Each dataset is indexed by an R-tree (Guttman, 1984) with a page size of 4096 bytes.

We evaluate the effects of several parameters including parameters k and m, and the cardinality N for the following four algorithms. (1) The baseline algorithm (naive solution) is the brute force approach using the *linear scan* (denoted as LS in figures). According to the type of the GNN query, other algorithms include (2) lazy RkGNN with MBM query method in Papadias et al. (2004) (denoted as basic RkGNN or M in figures), (3) lazy RkGNN with window pruning (denoted as RkGNN+W or W in figures), and (4) lazy RkGNN with window pruning and reuse technology (denoted as RkGNN+WR or WR in figures). Each type of algorithms can also be divided into *monochromatic* RkGNN query, *bichromatic* RkGNN query, and *constrained* RkGNN query. Note that the basic RkGNN and RkGNN+W correspond to the previous algorithms, LRkGNN and SLRkGNN in Jiang et al. (2013), respectively. Since other related approaches (i.e., GNN algorithms) cannot be adopted to address the RkGNN queries, we do not compare them with our algorithms in the following experiments. In each experiment, only one parameter is varied, whereas the others are fixed to their default values. The settings of the parameters and their default values are listed in Table 3.

The *wall clock time* (i.e., the sum of I/O time and CPU time), where the I/O time is computed by charging 10 ms for each page access, as with Papadias et al. (2005), *the number of node/page accesses* (*NA*), the number of enumerating combinations (*EC*) which reflects the performance of early stopping, *the* maximum number of entries in the reuse heap *(MH)*, are used as the major performance metrics. Each reported value in the diagrams is the average of 50 queries, where the query object q of each query is randomly selected from the corresponding dataset. In order to adequately show the efficiency of RkGNN+WR, we also measure the *speed-up ratio* of our approaches as the wall clock time of basic RkGNN divided by that of RkGNN+WR. Note that the columns in figures indicate the wall clock time, whereas the curves represent *NA*. All algorithms were implemented in C++ programming language, and all experiments were conducted on an Intel 2.0 GHz single-CPU PC with 4GB RAM.

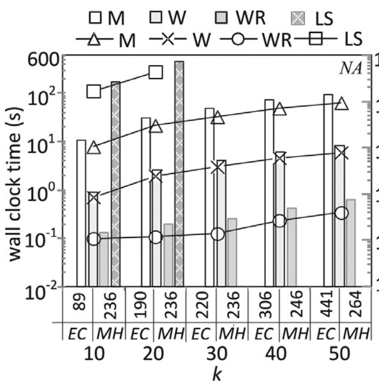
*8.2. Performance of MR*k*GN queries*

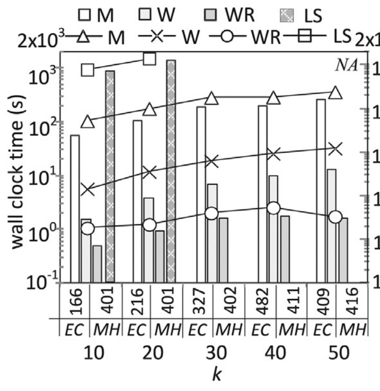
Effect of *k* on MR*k*GN. In the first set of experiments, we test the effect of k on the performance of the MRkGN queries. Fig. 10 illustrates the experimental results of four algorithms by setting $m = 3$ and varying the parameter k from 10 to 50. The data size is $N = 9$ K for the data set of *PP*, and $N = 60$ K for the data sets of *NE, CO* and *IN*. From figures, we can see that RkGNN+WR achieves the best performance and outperforms the baseline approach LS by about 3–4 orders of magnitude. Since MBM prunes data objects (or nodes) in the whole data space, whereas WP method only needs to search a smaller limited space, i.e., *WR*(*q, G*), this makes basic RkGNN with MBM slower than RkGNN+W. As expected, the wall clock time and *NA* of the four algorithms increase when k increases from 10 to 50. This is because that the number of candidate combinations increases with the growth of k accordingly. However, the rate of increase for RkGNN+WR is slower which can be attributed to the use of window pruning heuristic (WP) and reuse heap technology (RH). WP has a less number of node accesses than MBM on checking whether the current combination is a result of RkGNN query since WP only requires searching a small limited search range by Theorem 3 and Corollary 3. RH further cuts down the number of node accesses because it traverses the R-tree *only once*. In addition, we also observe that *EC* increases very fast with the growth of k, whereas *MH* increases much slower. This is because a big k causes an increase of visiting index nodes, and thus produces more combinations. Meanwhile, it also enlarges the size of the reuse heap. Since the baseline approach LS is thousands of times slower than our best algorithm due to the need to generate an exponential number of combinations, we will not report the results of LS in the subsequent experiments.

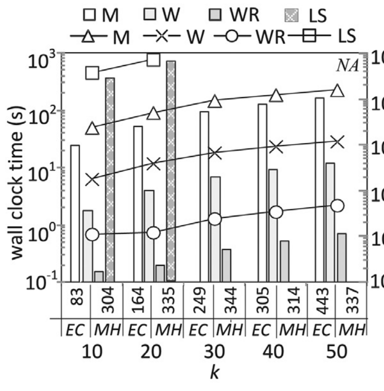
Effect of *m* on MR*k*GN. Fig. 11 illustrates the performance of the MRkGN queries by varying the number of data objects in a combination, where k is set to 30 and the data size $N = 100$ K ($N = 9$ K for *PP*). As shown in the figure, the wall clock time and *NA* decrease across all datasets when the parameter m increases. Moreover, *EC* and *MH* also decrease in most cases for a bigger parameter m. For example, *EC* is 623 on *PP* for $m = 2$, whereas *EC* becomes 256 on *PP* for $m = 4$. The *speed-up ratio* follows the same trend on all datasets. In fact, a bigger parameter m causes a larger search space. Why does this phenomenon take place? This is mainly because the MRkGN candidates are more likely to become the query results when the number of data objects in a combination is large. Consequently, MRkGN needs to visit fewer index nodes. In particular, RkGNN+WR obtains the highest speed-up ratio on the dataset *CO*, i.e., 533.32. On the contrary, the basic RkGNN has the worst performance among three algorithms owing to the adoption of MBM. The reason lies in the fact that the goal of MBM is to find the GNN of a combination G whereas the target of WP method is to determine whether the query object q is the GNN of G. Thus, WP method can take advantage of some useful information of q so as to accelerate the search.

Effect of *N* on MR*k*GN. In this set of experiments, we demonstrate the scalability of the MRkGN query algorithm by varying the data size N (the cardinality of dataset). The cardinality of *PP* is from 3 K to 15 K, and the cardinality of *NE* is from 20 K to 100 K. Since the real dataset with 100 K objects is too small to show that the proposed method is scalable, we also generate two bigger synthetic datasets, *CO* and *IN*, where their cardinality are from 128 K to 2048 K. By default, $m = 3$ and $k = 30$. As shown in Fig. 12, the wall clock time, *NA, EC and MH* are relatively constant with the increase of N. The main reason is that the larger data set often brings more valid combinations than a smaller data set. Thus, the MRkGN algorithm is still able to quickly obtain the top k query results although there is a larger search space for a bigger dataset. In most cases, basic MRkGN algorithm has a slower decrease of performance on wall clock time and *NA* with the increase of N. However RkGNN+W and RkGNN+WR show a better performance with the increase of N in most cases. This indicates that window pruning method wins a better efficiency than MBM for judging whether the current combination G is a RGNN of query object q owing to the same reason mentioned above the experiments of MRkGN about the parameters of k and m. This confirms the nice scalability of our approaches on data size.

The memory usages of R*k*GNN+WR. In this set of experiments, we report the memory usages of RkGNN+WR by varying the


(a) *PP*(*m*=3, *N*=9K)


(b) *NE*(*m*=3, *N*=60K)


(c) *CO*(*m*=3, *N*=60K)

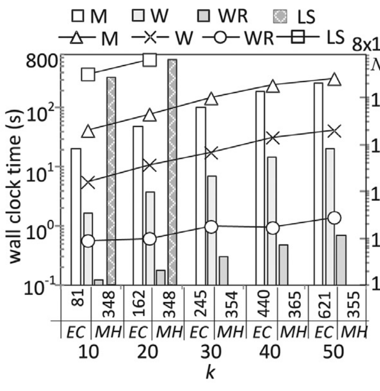
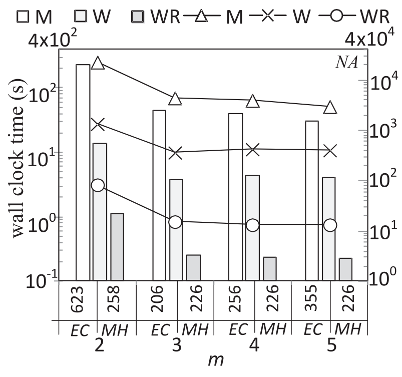
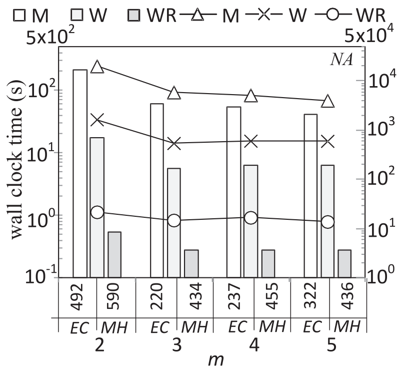
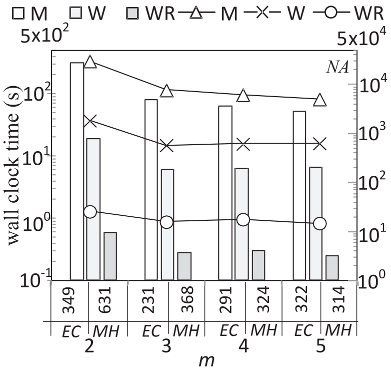

(d) *IN*(*m*=3, *N*=60K)

Fig. 10. The performance of MR*k*GN vs. *k*.


(a) *PP*(*k*=30, *N*=9K)


(b) *NE*(*k*=30, *N*=60K)


(c) *CO*(*k*=30, *N*=60K)

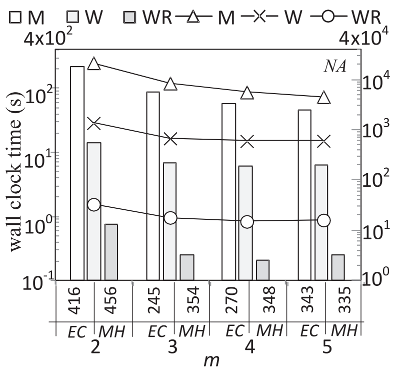

(d) *IN*(*k*=30, *N*=60K)

Fig. 11. The performance of MR*k*GN vs. *m*.

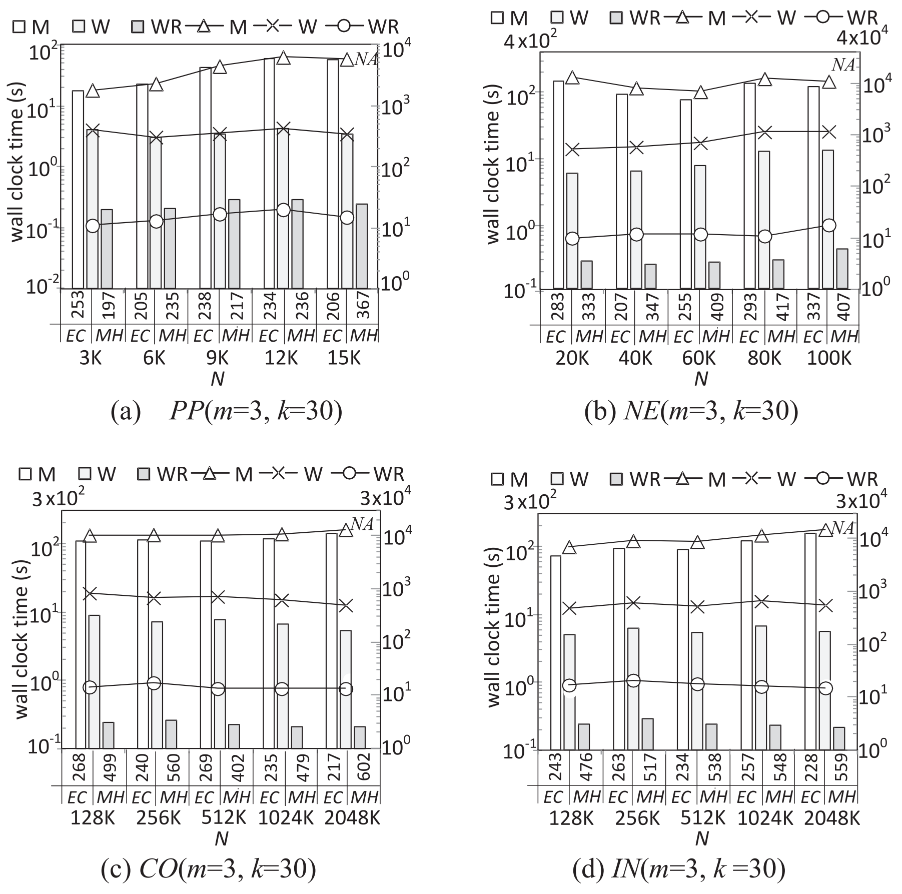


Fig. 12. The performance of MR*k*GN vs. *N*.

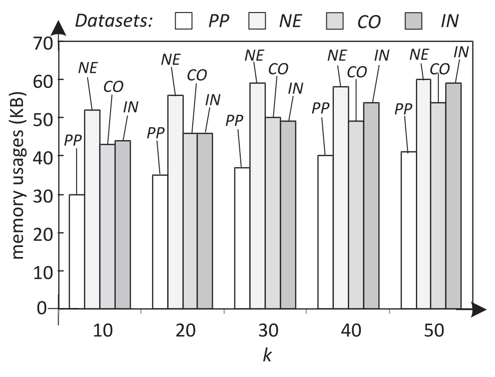


Fig. 13. The memory usages of MR*k*GN w.r.t. *k*.

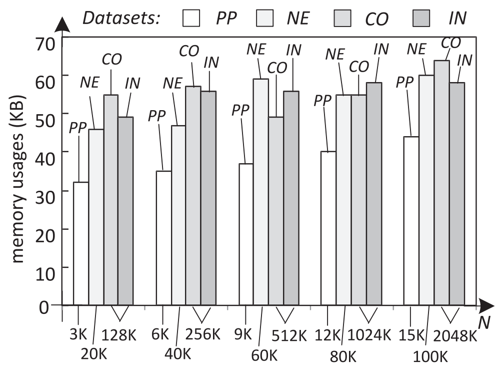


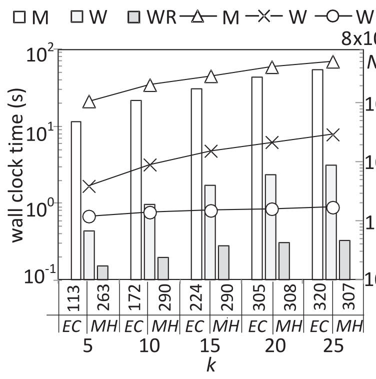
Fig. 14. The memory usages of MR*k*GN w.r.t. *N*.

parameter *k* and the data size *N*, where the other parameter settings follow the settings of above experiments in Fig. 10 and Fig. 12, respectively, since R*k*GNN+WR maintains the expanded points in memory instead of discarding them. Figs. 13 and 14 show the experimental results. From the figures, we can observe that the two parameters exert a little impact on memory usages of R*k*GNN+WR. In fact, the maximum memory usages are less than 70 KB over all experiments. The main reason is that R*k*GNN+WR with CSA can quickly obtain top-*k* results and then it only need to traverse a few of intermediate index nodes. On the other hand, the window queries in WP approach only require a little of memory since the pruning region is smaller at the beginning stage of incremental evaluation. Below, we will not present the experimental results because all the subsequent experimental results show similar trends for other parameters.

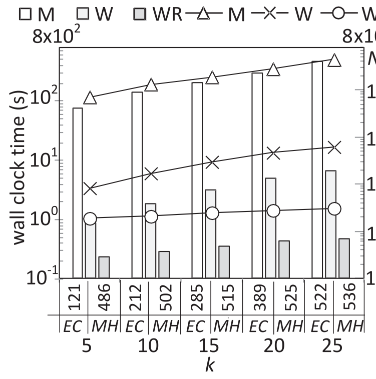
8.3. Performance of BR*k*GN queries

In this subsection, we study the query performance of the BR*k*GN query processing. Since BR*k*GN involves two datasets, we randomly extract 6 K data objects from *PP* as dataset *A*, and the rest of *PP* as dataset *B*. For the datasets of *NE, CO* and *IN*, we split them into the datasets *A* and *B* in a similar way, which contain 40 K and 60 K data objects, respectively.

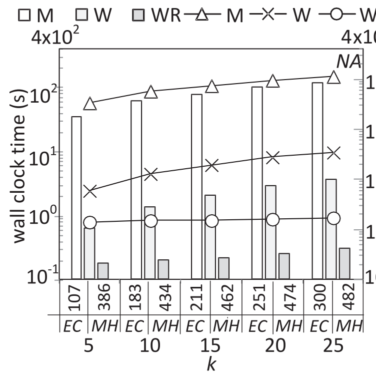
Effect of *k* on BR*k*GN. Fig. 15 illustrates the query performance of BR*k*GN by letting $m = 3$ and varying the parameter *k* from 5 to 25. From Fig. 15, we can see that when *k* increases, although all algorithms take more time to process the query, BR*k*GN+W and BR*k*GN+WR are much faster than the basic BR*k*GN. This indicates the effectiveness of our bichromatic window pruning method and reuse heap technology. In addition, we also find that BR*k*GN



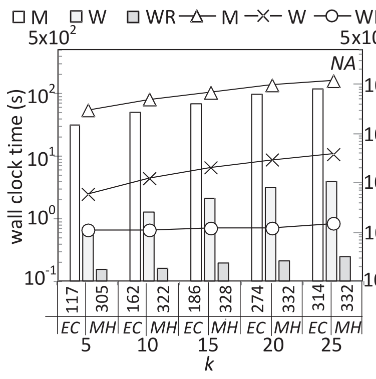
(a) *PP*6K-*PP*9K (*m*=3)



(b) *NE*40K-*NE*60K (*m*=3)

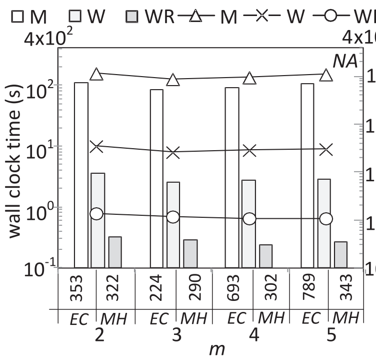


(c) *CO*40K-*CO*60K (*m*=3)

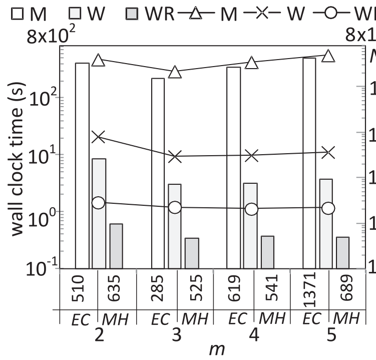


(d) *IN*40K-*IN*60K(*m*= 3)

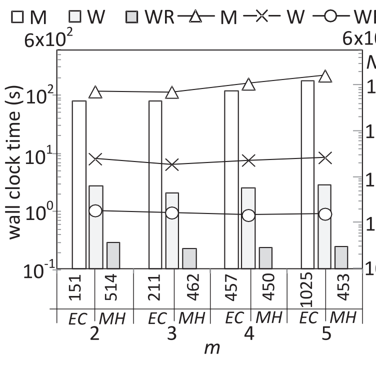
Fig. 15. The performance of BR*k*GN vs. *k*.



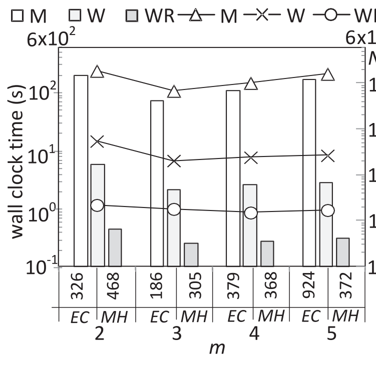
(a) *PP*6K-*PP*9K (*k*=15)



(b) *NE*40K-*NE*60K (*k*=15)

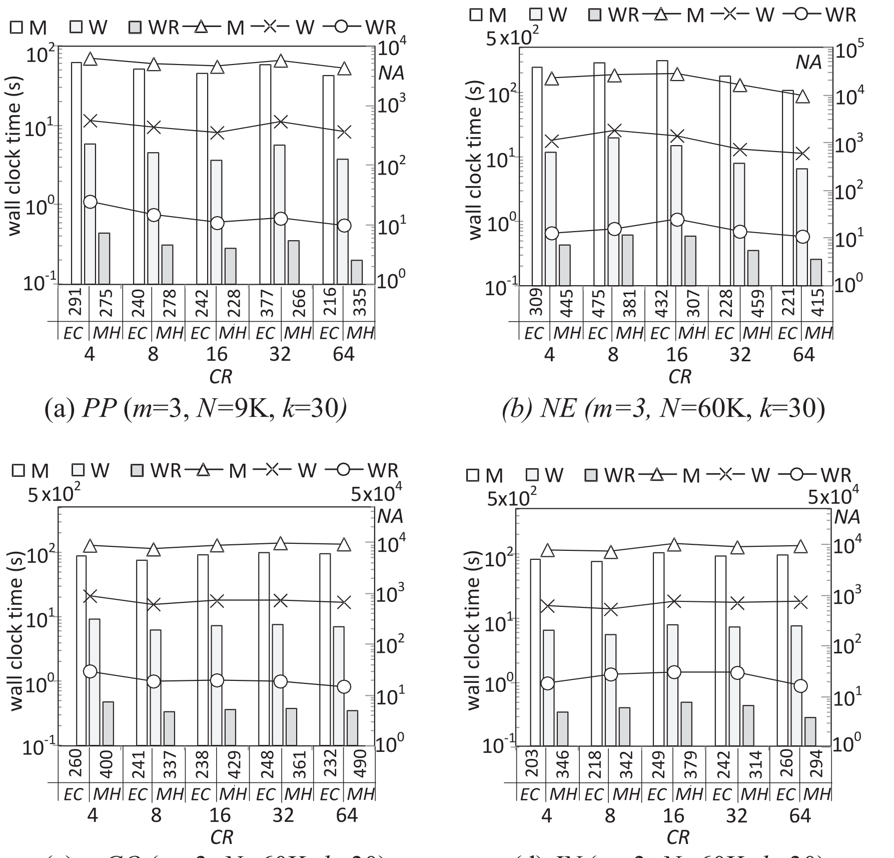


(c) *CO*40K-*CO*60K (*k*=15)



(d) *IN*40K-*IN*60K (*k*=15)

Fig. 16. The performance of BR*k*GN vs. *m*.



(a) *PP* (*m*=3, *N*=9K, *k*=30)   (b) *NE* (*m*=3, *N*=60K, *k*=30)

(c) *CO* (*m*=3, *N*=60K, *k*=30)   (d) *IN* (*m*=3, *N*=60K, *k*=30)

Fig. 17. The performance of MR*k*GN vs. CR (constrained region).

is slower than MR*k*GN. The reason is straightforward, i.e., BR*k*GN needs to traverse two R-trees whereas MR*k*GN only needs to traverse one.

Effect of *m* on BR*k*GN. Furthermore, we study the effect of *m* on our BR*k*GN query processing in Fig. 16, where $k = 15$. The experimental results show that, when the number of data objects in a combination increases, the required number of node accesses increases slightly in most of cases. This is because the bigger *m* is, the more candidate combinations will be. However, the wall clock time is nearly unchanged for $m = 3$, 4 and 5. The reason is that a bigger *m* brings more candidate combinations which have *q* as their GNN objects. Overall, BR*k*GN+WR has a better performance than basic BR*k*GN and BR*k*GN+W for all cases. Moreover, when *m* increases, the wall clock time also goes up.

8.4. Performance of constrained queries

In this subsection, we evaluate the performance of the constrained R*k*GNN. Due to space limitations, we only give the experimental results of MR*k*GN w.r.t. *CR* and *k*.

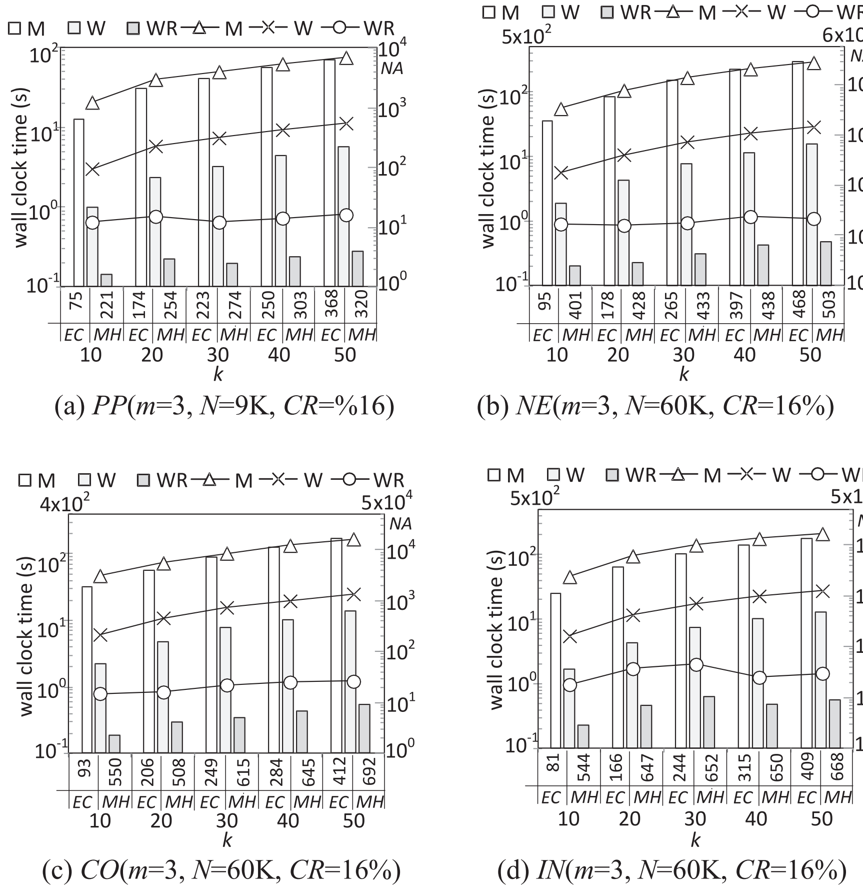
Effect of *CR* on MR*k*GN. First, we examine the influence of *CR* on the performance of the MR*k*GN query by varying *CR* from 4% to 64% (of the data space), where $m = 3$, $k = 30$, and the data size $N = 60$ K ($N = 9$ K for *PP*). The results are shown in Fig.17. It is obvious that, all the algorithms are I/O bounded, and both the wall clock time and *NA* smoothly decrease with *CR* in most cases. The reason is that, we generate the constrained region according to the center point of *root* node's MBR, and a larger constrained region usually contains more qualified combinations. Thus, the algorithm will more easily obtain the top *k* query results. The experimental results confirm the nice scalability on *CR*.

Effect of *k* on constrained MR*k*GN. Then, we evaluate the effect of *k* on the efficiency of the MR*k*GN algorithms by varying *k* from 10 to 50 with $m = 3$, the data size $N = 60$ K ($N = 9$ K for *PP*). Fig. 18 reports the results. As expected, the larger the parameter *k*, the higher the cost since the number of combinations increases with *k*. CSA greatly reduces the number of combinations to be evaluated. In all cases, the constrained MR*k*GN+WR always achieves the best performance. The experimental results indicate that, the constrained MR*k*GN algorithm has the same characteristics as the regular MR*k*GN algorithm.

8.5. The potential of our system

As we can see from the experiments, our system can output the answer for a small *k* ($k \leq 30$) in a few seconds (i.e., 12 seconds) even when the database size (namely, the number of data objects in a dataset) is increased to two million (i.e., 2048 K). For the most parameters, i.e., *m, N*, and *CR*, they have a little impact on the wall clock time and the number of node access. As for the parameter *k*, the processing time slightly increases with the increase of *k*. Therefore, this indicates that our system has good scalability and its time complexity is relatively low.

In addition, our system also shows some performance merits in term of scalability. For example, (i) in order to obtain the top-*k* query results, it only needs to incrementally produce only hundreds of combinations. for example, the maximum of *EC* is 1371 for $m = 5$ in bichromatic case; (ii) the number of *MH* is also very acceptable because it is less than 1000; (iii) the maximum memory usages are less than 70 KB over all experiments. From managerial insights, our system can apply to many important fields, including resource allocation, tripping planning, product design, and so on.



(a) *PP*(*m*=3, *N*=9K, *CR*=%16) (b) *NE*(*m*=3, *N*=60K, *CR*=16%)

(c) *CO*(*m*=3, *N*=60K, *CR*=16%) (d) *IN*(*m*=3, *N*=60K, *CR*=16%)

Fig. 18. The performance of constrained MR*k*GN vs. *k*.

9. Conclusion

For some expert and intelligent systems, group nearest neighbor (GNN) query can be regarded as an important tool to capture their geographic information. These applications include location-based service (LBS) and business support systems. However, they only process the query from users' perspective; we focus on managers' perspective in this paper. Our proposed system is applicable to many important domains, such as resource allocation, product data analysis, tripping planning, and disaster management.

To this end, we present a new type of query, namely, *reverse top-k group nearest neighbor* (R*k*GNN) query for monochromatic and bichromatic cases. But, this query needs a large number of computations owing to its exponential time and space complexities although it is very useful. We develop efficient algorithms to address the R*k*GNN query. In theory, we propose STP, MP, and WP techniques to incrementally generate combinations for consideration and quickly prune unqualified candidate combinations. The proposed techniques result in a significant improvement of answering the R*k*GNN query. At last, we have also conducted an extensive experimental evaluation over both real and synthetic datasets, and the results demonstrate the effectiveness and efficiency of our proposed algorithms.

In the future, we intend to devise more efficient algorithm(s) for answering R*k*GNN queries. Another interesting direction for future work is to extend our approaches to tackle other variants of R*k*GNN queries, i.e., the R*k*GNN queries with parallelization and the R*k*GNN queries in metric spaces. Finally, we plan to solve the R*k*GNN queries in road networks.

Acknowledgment

Bin Zhang was supported in part by ZJNSF Grant LY14F020038. Tao Jiang was supported in part by ZJNSF Grant LY16F020026. Li Chen was supported in part by ZJNSF Grant LY15F020040.

References

Chuang, Y.-C., Su, I.-F., & Lee, C. (2013). Efficient computation of combinatorial skyline queries. *Information Systems, 38*(3), 369–387.

Deng, K., Sadiq, S., Zhou, X., Xu, H., Fung, G. P. C., & Lu, Y. (2012). On group nearest group query processing. *IEEE Transaction on Knowledge and Data Engineering, 24*(2), 295–308.

Drosou, M., & Pitoura, E. (2010). Search result diversification. *SIGMOD Record, 39*(1), 41–47.

Fagin, R., Lotem, A., & Naor, M. (2001). Optimal aggregation algorithms for middleware. In *Proceedings of the 20th symposium on principles of database systems* (p. 102–113).

Fu, Z., Sun, X., Liu, Q., Zhou, L., & Shu, J. (2015). Achieving efficient cloud search services: multi-keyword ranked search over encrypted cloud data supporting parallel computing. *IEICE Transactions on Communications, E98-B*(1), 190–200.

Gao, Y., Liu, Q., Zheng, B., & Chen, G. (2014). On efficient reverse skyline query processing. *Expert Systems with Applications, 41*(7), 3237–3249.

Gao, Y., Zheng, B., Chen, G., Chen, C., & Li, Q. (2011). Continuous nearest-neighbor search in the presence of obstacles. *ACM Transaction on Database System, 36*(2), Article No. 9.

Gollapudi, S., & Sharma, A. (2009). An axiomatic approach for result diversification. In *Proceedings of WWW Conference* (p. 381–390).

Guo, X., Xiao, C., & Ishikawa, Y. (2012). Combination skyline queries. *Transaction on Large-Scale Data- and Knowledge-Centered System, 6*, 1–30.

Guttman, A. (1984). R-Trees: a dynamic index structure for spatial searching. In *Proceedings of ACM SIGMOD Conference* (p. 47–57).

Hashem, T., Kulik, L., & Zhang, R. (2010). Privacy preserving group nearest neighbor queries. In *Proceedings of the international conference on extending database technology* (p. 489–500).

Hjaltason, G., & Samet, H. (1999). Distance browsing in spatial databases. *ACM Transaction on Database Systems, 24*(2), 265–318.

Im, H., & Park, S. (2012). Group skyline computation. *Information Science, 188*, 151−169.

Jiang, T., Gao, Y., Zhang, B., Lin, D., & Li, Q. (2014). Monochromatic and bichromatic mutual skyline queries. *Expert Systems with Applications, 41*(4), 1885−1900.

Jiang, T., Gao, Y., Zhang, B., Liu, Q., & Chen, L. (2013). Reverse top-*k* group nearest neighbor search. In *Proceedings of the 14th international conference on web-age information management.* (p. 429−439).

Jiang, T., Zhang, B., Lin, D., Gao, Y., & Li, Q. (2015). Incremental evaluation of top-*k* combinatorial metric skyline query. *Knowledge-Based Systems, 74*(1), 89−105.

Kolahdouzan, M., & Shahabi, C. (2004). Voronoi-based *k* nearest neighbor search for spatial network databases. In *Proceedings of the international conference on very large data base* (pp. 840–851).

Korn, F., & Muthukrishnan, S. (2000). Influence sets based on reverse nearest neighbor queries. In *Proceedings of ACM SIGMOD conference* (pp. 201–212).

Li, Y., Li, F., Yi, K., Yao, B., & Wang, M. (2011). Flexible aggregate similarity search. In *Proceedings of ACM SIGMOD conference* (pp. 1009–1020).

Li, F., Yao, B., & Kumar, P. (2010). Group enclosing queries. *IEEE Transaction on Knowledge and Data Engineering, 23*(10), 1526−1540.

Lian, X., & Chen, L. (2008). Probabilistic group nearest neighbor queries in uncertain databases. *IEEE Transaction on Knowledge and Data Engineering, 20*(6), 809−824.

Magnani, M., & Assent, I. (2013). From stars to galaxies: Skyline queries on aggregate data. In *Proceedings of the international conference on extending database technology* (pp. 477–488).

Mouratidis, K., Yiu, M. L., Papadias, D., & Mamoulis, N. (2006). Continuous nearest neighbor monitoring in road networks. In *Proceedings of the international conference on very large data base* (pp. 43–54).

Papadias, D., Shen, Q., Tao, Y., & Mouratidis, K. (2004). Group nearest neighbor queries. In *Proceedings of the international conference on data engineering* (p. 301−312).

Papadias, D., Tao, Y., Mouratidis, K., & Hui, C. K. (2005). Aggregate nearest neighbor queries in spatial databases. *ACM Transaction on Database Systems, 30*(2), 529–576.

Park, Y., Min, J.-K., & Shim, K. (2013). Parallel computation of skyline and reverse skyline queries using MapReduce. *Proceeding of the VLDB Endowment, 6*(14), 2002–2013.

Qin, L., Yu, Jeffrey, X., & Chang, L. (2012). Diversifying top-*k* results. *Journal Proceedings of the VLDB Endowment, 5*(11), 1124–1135.

Razente, H. L., Barioni, M. C. N., Traina, A. J. M., Faloutsos, C., Jr., & C. T. (2008). A novel optimization approach to efficiently process aggregate similarity queries in metric access methods. In *Proceedings of the 17th ACM conference on information and knowledge management* (p. 193−202).

Roussopoulos, N., Kelly, S., & Vincent, F. (1995). Nearest neighbor queries. In *Proceedings of ACM SIGMOD conference* (pp. 71–79).

Seidl, T., & Kriegel, H.-P. (1998). Optimal multi-step *k*-nearest neighbor search. In *Proceedings of ACM SIGMOD conference* (pp. 154–165).

Stanoi, I., Agrawal, D., & Abbadi, A. (2000). Reverse nearest neighbor queries for dynamic databases. In *Proceedings of SIGMOD workshop on research issues in data mining and knowledge discovery* (pp. 44–53).

Su, I.-F., Chuang, Y.-C., & Lee, C. (2010). Top-*k* combinatorial skyline queries. In *Proceedings of the international conference on database systems for advanced applications* (pp. 79–93).

Tao, Y., Ding, L., Lin, X., & Pei, J. (2009). Distance-based representative skyline. In *Proceedings of the international conference on data engineering* (pp. 892–903).

Tao, Y., Papadias, D., & Lian, X. (2004). Reverse *k*NN search in arbitrary dimensionality. In *Proceedings of the international conference on very large data base* (p. 744−755).

Tao, Y., Yiu, M. L., & Mamoulis, N. (2006). Reverse nearest neighbor search in metric spaces. *IEEE Transaction on Knowledge and Data Engineering, 18*(9), 1239–1252.

Vieira, M. R., Razente, H. L., & Barioni, M. C. N. (2011). On query result diversification. In *Proceedings of the international conference on data engineering* (pp. 1163–1174).

Wong, R. C.-W., Ozsu, M. T., Yu, P. S., Fu, A. W.-C., & Liu, L. (2009). Efficient method for maximizing bichromatic reverse nearest neighbor. *PVLDB, 2*(1), 1126–1137.

Xia, Z., Wang, X., Sun, X., & Wang, Q. (2015). A secured and dynamic multi-keyword ranked search scheme over encrypted cloud data. *IEEE Transactions on Parallel and Distributed Systems, 27*(2) 2015. doi:10.1109/TPDS.2015.2401003.

Xia, T., & Zhang, D. (2006). Continuous reverse nearest neighbor monitoring. In Proceedings of the 22nd *international conference on data engineering* (p. 77).

Xiao, X., Yao, B., & Li, F. (2011). Optimal location queries in road network databases. In *Proceedings of the international conference on data engineering* (p. 804−815).

Yiu, M. L., & Mamoulis, N. (2006). Reverse nearest neighbors search in ad-hoc subspaces. In *Proceedings of the 22nd international conference on data engineering* (p. 76).

Yiu, M. L., Mamoulis, N., & Papadias, D. (2005). Aggregate nearest neighbor queries in road networks. *IEEE Transaction on Knowledge and Data Engineering, 17*(6), 820−833.

Yiu, M. L., Papadias, D., Mamoulis, N., & Tao, Y. (2006). Reverse nearest neighbors in large graphs. *IEEE Transaction on Knowledge and Data Engineering, 18*(4), 540−553.

Yu, W. (2016). Spatial co-location pattern mining for location-based services in road networks. *Expert Systems with Applications, 46*, 324–335. doi:10.1016/j.eswa.2015.10.010.

Yu, X., Pu, K. Q., & Koudas, N. (2005). Monitoring *k*-nearest neighbor queries over moving objects. In *Proceedings of the 21nd international conference on data engineering* (p. 631−642).

Zhang, D., Chee, Y. M., Mondal, A., Tung, A. K. H., & Kitsuregawa, M. (2009). Keyword search in spatial databases: towards searching by document. In *Proceedings of the international conference on data engineering* (pp. 688–699).

Zhou, X., Wu, S., Chen, G., & Shou, L. (2014). *k*NN processing with co-space distance in SoLoMo systems. *Expert Systems with Applications, 41*(16), 6967–6982.