Reverse Furthest Neighbors in Spatial Databases

Bin Yao, Feifei Li, Piyush Kumar

Computer Science Department, Florida State University
Tallahassee, Florida, USA
{yao, lifeifei, piyush}@cs.fsu.edu

***Abstract*— Given a set of points P and a query point q, the *reverse furthest neighbor* (RFN) query fetches the set of points $p \in P$ such that q is their furthest neighbor among all points in $P \cup \{q\}$. This is the monochromatic RFN (MRFN) query. Another interesting version of RFN query is the *bichromatic reverse furthest neighbor* (BRFN) query. Given a set of points P, a query set Q and a query point $q \in Q$, a BRFN query fetches the set of points $p \in P$ such that q is the furthest neighbor of p among all points in Q. The RFN query has many interesting applications in spatial databases and beyond. For instance, given a large residential database (as P) and a set of potential sites (as Q) for building a chemical plant complex, the construction site should be selected as the one that has the maximum number of reverse furthest neighbors. This is an instance of the BRFN query. This paper presents the challenges associated with such queries and proposes efficient, R-tree based algorithms for both monochromatic and bichromatic versions of the RFN queries. We analyze properties of the RFN query that differentiate it from the widely studied reverse nearest neighbor queries and enable the design of novel algorithms. Our approach takes advantage of the furthest Voronoi diagrams as well as the convex hulls of either the data set P (in the MRFN case) or the query set Q (in the BRFN case). For the BRFN queries, we also extend the analysis to the situation when Q is large in size and becomes disk-resident. Experiments on both synthetic and real data sets confirm the efficiency and scalability of proposed algorithms over the brute-force search based approach.**

I. Introduction

Spatial databases have offered a large number of applications in last decade that shape the horizon of computing services from people's daily life to scientific research. For example, people rely on online map services to plan their trips; the deployment and query processing in large sensor networks [15] often require the design of location-aware algorithms. Driven by these increasing number of applications, efficient processing of important and novel query types in spatial databases has always been a focal point.

In this work, we identify a query type that has wide applications and requires novel algorithms for processing. For a large data set P and any random query point q, we are interested in retrieving the set of points in P that take q as their furthest neighbors comparing to all points in P, i.e., collecting q's reverse further neighbors (RFN). This problem is referred as the *monochromatic reverse furthest neighbor* (MRFN) queries. It naturally has a bichromatic version as well (BRFN). Specifically, the query contains a set of query points Q and one point $q \in Q$. The goal in this case is to find the set of points $p \in P$ so that they all take q as their furthest neighbors *compared to all points in Q*.

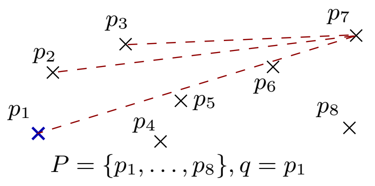


Fig. 1. An MRFN query example.

Consider the data set P in Figure 1, suppose the query point q is p_1, then its RFN includes $\{p_6, p_7, p_8\}$. Using p_7 as an example, visually its potential furthest neighbor is among p_1, p_2 and p_3. The dotted lines indicate the distances of these candidates to p_7 and among them p_1 is the furthest. Hence, p_7 is included in the RFN of p_1. On the other hand, p_5 is not included as p_5's furthest neighbor becomes p_7. This is an instance of the MRFN query. Note that the query point q could be different from any existing point in P.

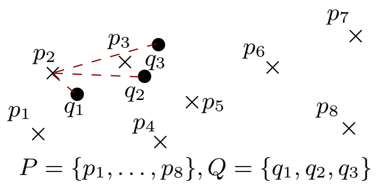


Fig. 2. An BRFN query example.

The BRFN query takes a set of query points as the input. With the same data set P, Figure 2 demonstrates the BRFN query. The BRFN of q_3 w.r.t Q in Figure 2 is $\{p_1, p_2, p_4\}$. Using p_2 as an illustration, its distance to q_1, q_2, q_3 are denoted by the dotted lines and q_3 is the furthest. Similarly, q_1's BRFN is $\{p_3, p_5, p_6, p_7, p_8\}$. q_2's BRFN in this case is empty. A careful examination will confirm that no point in P takes q_2 as its furthest neighbor from points in Q.

The motivation to study the RFN queries is largely inspired by an important query type that has been extensively studied recently, namely, the reverse nearest neighbor queries (RNN) [29], [20], [34], [5], [1], [35], [28], [26], [23], [33], [22], [27]. Intuitively, an RNN query finds the set of points taking the query point as their nearest neighbors and it also exists in both the monochromatic and bichromatic versions. Many applications that are behind the studies of the RNN queries naturally have the corresponding "furthest" versions. Consider next two examples for the MRFN and BRFN queries:

Example 1 For a large collection of points of interest in a region, every point would like to learn the set of sites that take itself as their furthest neighbors compared to other points of interest. This has an implication that visitors to those sites (i.e., its reverse furthest neighbors) are highly unlikely to visit this point. Ideally, it should put more efforts in advertising itself to those sites.

Example 2 Given a database of residential sites as P and a collection of potential locations Q for building a chemical plant, due to the hazardous materials that may be produced by such a plant, we should select the location in Q that is further away (compared to other choices in Q) from as many residential sites as possible, i.e., the point $q \in Q$ that has the largest number of reverse furthest neighbors in P.

The above example does not limit itself to spatial data sets, as long as there is a similarity metric between any two objects, the BRFN queries are applicable.

Example 3 Let P be a large set of customers and Q be a set of business competitors offering similar products in the market, suppose for each customer p and each competitor q there is a distance measure reflecting the rating of p to q's product. The smaller value indicates a higher preference. For any competitor in Q, an interesting query is to discover the customers that dislike his product the most among all competing products in the market. This company could then carry out specialized analysis on this group of customers to identify potential drawbacks in its marketing strategy.

In the last example, an interesting observation is that Q could be large and becomes disk-resident data set as well. As a result, the algorithm for the BRFN problem needs to be extended to the case where both P and Q are in external memory.

To the best of our knowledge, both the MRFN and BRFN have not been studied in the literature. The brute-force search algorithms for these problems are obviously too expensive to be of any practical use. It is worth mentioning that the BRFN problem has been briefly examined from a theoretical point of view recently [10]. However, large scale spatial databases are calling for a practical, efficient algorithm for this problem. More importantly, by taking the furthest neighbors, the geometric nature of the RFN problems has been changed from the RNN problems. Hence, we need to design new algorithms to process the RFN queries more efficiently by taking the new geometric perspectives into account.

Our Contributions. This work presents efficient algorithms for the MRFN and BRFN problems. We identify the important insights for the RFN problems based on the convex hulls of either P or Q and extend the basic idea to work with dynamic data sets as well as disk resident query groups. The paper is organized as follows: 1) We formulate the problem of the reverse furthest neighbors in Section II and survey the related work in Section III; 2) We propose effective pruning techniques based on several geometric insights to obtain two efficient, R-tree based exact algorithms for the MRFN problem in Section IV; In particular, we design R-tree based algorithm to maintain P's convex hull dynamically so that our query algorithms work efficiently with dynamic data sets; 3) We present a practical and efficient R-tree based algorithm for the BRFN problem in Section V; 4) We discuss the generalization to handle query groups with large size for the BRFN problem in Section V-A; 5) We report a comprehensive experimental study with both synthetic and real data sets in Section VI.

II. Problem Formulation

Let P denote the spatial database in d-dimensional *Euclidean space*. Our techniques could be easily generalized for any metric space where the distance between points satisfies the triangle inequality. For simplicity, we concentrate on the Euclidean distance where the distance between any two points p and q is denoted by $||p - q||$. The furthest neighbor of any point p w.r.t a set of points P is simply defined as:

Definition 1 The furthest neighbor of p to a data set P is defined as $\text{fn}(p, P) = p^*$ s.t. $p^* \in P$, for $\forall p' \in P$ and $p' \neq p^*$, $||p^* - p|| \geq ||p' - p||$. Ties are broken arbitrarily.

The monochromatic RFN query is formally defined as:

Definition 2 The MRFN of q w.r.t the data set P is a set of points from P that take q as their furthest neighbors comparing to all points in P, i.e., MRFN (q, P)= $\{p | p \in P,\ \text{fn}(p, P \bigcup \{q\}) = q$.

The bichromatic RFN query takes additionally a set of query points Q as the input, and is formally defined in the follows.

Definition 3 The BRFN of $q \in Q$ w.r.t the data set P and the query set Q is a set of points from P that take q as their furthest neighbors comparing to all other points in Q, i.e., BRFN (q, Q, P)= $\{p | p \in P,\ \text{fn}(p, Q) = q\}$.

Assume that $|P| = n$, $|Q| = m$ and a page size of B, the brute-force search based approach for the MRFN (BRFN) problem takes $\mathcal{O}(n^2)$ ($\mathcal{O}(mn)$) time complexity with $\mathcal{O}(n^2/B)$ ($\mathcal{O}(n/B)$ if Q fits in memory or otherwise $\mathcal{O}(mn/B)$) I/Os.

Other Notations. We summarize some of our notations here. The convex hull of a point set is defined as the smallest convex polygon containing all points. Intuitively, the convex hull is obtained by spanning an elastic band around the point set. The points touched by the elastic band become the vertices of the convex hull. We use $\mathcal{C}_P$ to denote both the ordered set of vertices and the convex polygon (the area enclosed by the elastic band) for the convex hull of P. Its meaning is clear from the context. An ordered list of points $p_1 p_2 \cdots p_t p_1$ represents a convex polygon defined by line segments $\{p_1 p_2$, $p_2 p_3$, ..., $p_t p_1\}$ with t vertices. The concepts of convex hull and convex polygon work with any dimension d. We also used the furthest Voronoi diagram for a set of points ($\mathcal{FD}(P)$) and the furthest Voronoi cell for a single point w.r.t a set of points (fvc(p, P)). The details will be defined when the related discussion emerges.

Symbol	Description
$\|\|p-q\|\|$	Euclidean distance between p and q
$\|\cdot\|$	Size of a set
$(\subset)\ \subseteq$	Both (strict) set and geometric containment
$\mathcal{C}_P$ $(\mathcal{C}_Q)$	Ordered set of vertices of P's (Q's) convex hull, also is the convex polygon of P's (Q's) convex hull
fn(p, P)	The furthest neighbor of p in P
fvc(p, P)	The furthest Voronoi cell of p w.r.t P
$\mathcal{FD}(P)$, $\mathcal{FD}(Q)$	The furthest Voronoi diagram of P (Q)
n, m	$\|P\|$ and $\|Q\|$ respectively
p_1p_2	A line segment between p_1 and p_2
$p_1p_2\cdots p_tp_1$	The convex polygon by $\{p_1p_2, \ldots, p_tp_1\}$

TABLE I

NOTATION USED.

When $A \subseteq B$ is used with geometric objects A and B, it implies that all points from the area enclosed by A are part of the points from the area enclosed by B. When $\subset$ is used in this context, it further requires all points from the area enclosed by A are inside (while not touching) the boundary edges of B. Table I provides a quick reference to the main notations.

In this work we focuses on the two dimensional space. The main ideas developed could be generalized to higher dimensions and the details of such generalization is an interesting problem we will look at.

III. Background and Related Work

R-tree [17] and its variants (R*-tree as the representative [4]) have been the most widely deployed indexing structure for the spatial database, or data in multi-dimensions in general. Intuitively, R-tree is an extension of the B$^+$-tree to higher dimensions. Points are grouped into minimum bounding rectangles (MBRs) which are recursively grouped into MBRs in higher levels of the tree. The grouping is based on data locality and bounded by the page size. An example of the R-tree is illustrated in Figure 3. R-tree has facilitated processing many important query types for spatial databases and data in multi-dimensional space. Two important classes of queries that are related to this work and have been extensively studied in the literature are nearest neighbor (NN) queries and range queries.

NN search has been thoroughly studied for the Euclidean space [24], [30], [19], [32], [21], [6], [14], [25], [18], [12]. In the Euclidean space, R-tree demonstrates efficient algorithms using either the depth-first [25] or the best-first [18] approach. The main idea behind these algorithms is to utilize branch and bound pruning techniques based on the relative distances between a query point q to a given MBR N. Such distances include the mindist, the minmaxdist and the maxdist. The mindist measures the minimum possible distance for a point q to any point in an MBR N; the minmaxdist measures the lower bound on the maximum distance for a point q to any point in an MBR N; and finally, the maxdist simply measures the maximum possible distance between q and any point in an MBR N. These distances are easy to compute arithmetically given q and N. An example for these distances has been provided in Figure 3. The principle for utilizing them to prune the search space for NN search in R-tree is straightforward, e.g., when an MBR N_a's mindist to q is larger than another MBR N_b's minmaxdist, we can safely prune N_a entirely.

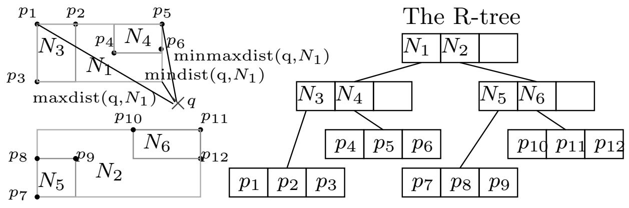


Fig. 3. The R-tree.

Another classical query type is the range query where the goal is to return all points that are fully contained by the query range (often as a rectangle). R-tree yields good performance for answering range queries [8]. The basic idea is to search all MBRs that intersect with the query range.

An interesting query type that has close relationship with NN search was defined in [22], in which the goal is to find the set of points from P that take the query point q as their nearest neighbors among all points in the data set P. This is the monochromatic reverse nearest neighbor query (monochromatic RNN). Due to its wide applications, RNN queries have received considerable attention since its appearance. In the monochromatic version [28], [26], [33], [22], [29], [27], the state of the art is the TPL method from [29]. TPL recursively filters the data by finding perpendicular bisectors between the query point and its nearest object and taking the half plane that is closer to the query point.

The bichromatic RNN also finds many applications [20], [1], [35], [26], [22], [27]. In this case, the query takes a set of query points Q and a query point $q \in Q$. The set of points returned from P all take q as their nearest neighbors w.r.t other points in Q. The basic idea here is to use the Voronoi diagram and find the region that corresponds to the query point.

Most of the work for the RNN queries focused on the euclidean space. Many interesting variants have been studied. The RNN problem in graphs and road network was studied in [35]. Generalization to any metric space appeared in [1]. Continuous RNN was explored by [20], [31]. The RNN for moving objects was studied in [5]. Reverse kNN search was examined by [29], [28]. Finally, the RNN for Ad-Hoc subspaces was solved by [34].

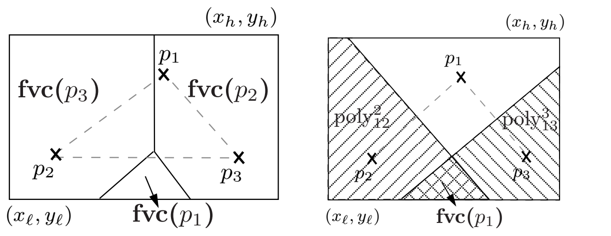
Our work explores the unique geometric property of the reverse furthest neighbors and proposes efficient, novel pruning techniques that are suitable in this case. In particular, we utilize the convex hull of either the data set in the case of MRFN or the query set in the case of BRFN for pruning. Hence, it is related with the RNN problem but based on significant different insights.

IV. Monochromatic Reverse Furthest Neighbors

We search for efficient, R-tree based algorithms for the MRFN queries in this section. Our approaches are inspired by the furthest Voronoi diagram and the convex hull.

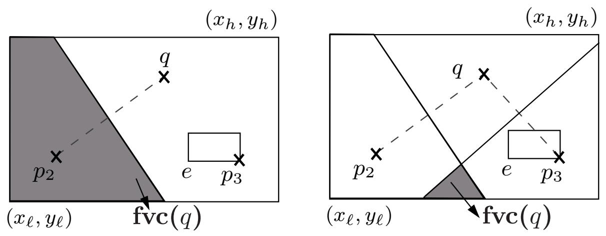
A. PFC: the Progressive Furthest Cell algorithm

For a set of points P, the *furthest Voronoi diagram* of P, denoted as $\mathcal{FD}(P)$, is similar to the well-known Voronoi



(a) Furthest Voronoi diagram. (b) Computing fvc(p_1, P).

Fig. 4. $\mathcal{FD}(P)$ and its derivation.



(a) Update V_c and V_p. (b) Prune false positive.

Fig. 5. PFC algorithm.

diagram [3] of P except that the space is partitioned with the *furthest Voronoi cells* instead of the *nearest Voronoi cells*.

Definition 4 For a space S, a point p and a set of points P ($p \in P$ and $P \subseteq \mathcal{S}$), a convex polygon fvc(p, P)$\subseteq \mathcal{S}$ is the *furthest Voronoi cell* of p iff: $\forall$ point $p' \subseteq$fvc(p, P), fn(p', P)$=$ p, i.e., any point inside fvc(p, P) will take p as its furthest neighbor in P. p' is not necessarily a point in P. The furthest Voronoi diagram of P ($\mathcal{FD}(P)$) is simply the collection of furthest Voronoi cells for all points in P.

An example of the furthest Voronoi diagram on an input space $\mathcal{S} : (x_\ell, y_\ell) \times (x_h, y_h)$ is illustrated in Figure 4(a) where fvc(p_i, P) is marked as region fvc(p_i). Computing the $\mathcal{FD}(P)$ can be done by modifying the algorithm for the Voronoi diagram . To find the fvc(p_i, P), for each point $p_j \in P$ and $j \neq i$, the bisector line of the line segment p_ip_j separates the space into two polygons and denote the one that does not contain p_i as poly$^j_{ij}$. Then fvc(p_i, P) is simply the intersection of poly$^j_{ij}$'s for all j's in P. The fact that poly$^j_{ij}$ is the polygon that is further away from p_i among the two polygons (covering the whole space) partitioned by the bisector line of p_ip_j straightforwardly yields the correctness of this algorithm. Figure 4(b) illustrates this idea for computing fvc(p_1, P). One first finds poly$^2_{12}$ with point p_2, followed by identifying poly$^3_{13}$. The fvc(p_1, P) is simply the poly$^2_{12} \bigcap$ poly$^3_{13}$.

An important property for the $\mathcal{FD}(P)$ is that any point from the furthest Voronoi cell of p_i takes p_i as its furthest neighbor among all points in P. Formally,

Lemma 1 *$\forall p' \subseteq$ fvc(p, P), fn(p', P) $= p$. In addition, for a random point p' in space $\mathcal{S}$, if fn(p', P) $= p$, then $p' \subseteq$ fvc(p, P).*

Proof: It is immediate by the construction of $\mathcal{FD}(P)$. ■

By Lemma 1, for a query point q, if we can compute the fvc($q, P \bigcup \{q\}$), then the RFN of q is simply those points in P that are contained by the fvc($q, P \bigcup \{q\}$). The challenge is how to compute fvc($q, P \bigcup \{q\}$) *efficiently* when P is large and stored on the disk. Obviously, a very expensive approach is to linear scan points in P and apply the algorithm above, then perform a range query with the discovered furthest Voronoi cell of q. Next, we present a progressive version of this idea, using the help of the R-tree, where the fvc($q, P \bigcup \{q\}$) is computed incrementally and the points in P that are contained by the fvc($q, P \bigcup \{q\}$) are also obtained progressively in one pass.

The progressive algorithm. The detail of the progressive furthest cell (PFC) algorithm is listed in Algorithm 1. Given the query point q and the R-tree index for the data set P, we maintain a priority queue $\mathcal{L}$ for storing MBRs and points. Objects in $\mathcal{L}$ are sorted in the decreasing order by their minmaxdist to q. The intuition is that the points that are further away from q have higher chances being the RFN of q, hence we should consider these candidates with higher priority. Meanwhile, fvc($q, P \bigcup \{q\}$) is initialized to the whole space $\mathcal{S}$ and we also maintain two vectors. V_c stores all points that are potentially the RFN of q and V_p stores all objects that may disqualify false positive candidates from V_c in the post-processing step. In the sequel, when the context is clear we simply use fvc(q) to denote q's up-to-date furthest Voronoi cell.

At each iteration, the head entry e of $\mathcal{L}$ is popped. If e is a point, we find the bisector line for the line segment qe. As shown in Figure 4(b), the bisector line cuts the space $\mathcal{S}$ into two polygons and we identify the one that is further away from q, i.e., poly$^e_{qe}$. The fvc(q) is updated as the intersection of the current fvc(q) with poly$^e_{qe}$. Next, we determine whether e is a potential candidate. If $e \subseteq$ fvc(q), then e is pushed into V_c. Otherwise, it is discarded.

If e is an MBR, we take the current furthest Voronoi cell of q and calculate $e \bigcap$fvc(q). If this is empty we push e into the pruning vector V_p, otherwise we retrieve each child of e and insert into $\mathcal{L}$ the ones that do intersect with fvc(q). The rest are pushed into V_p.

There are two insights for the PFC algorithm. The fvc(q) is updated progressively and it always shrinks after each update. Another critical observation (following the first observation) is that if an MBR or a point does not intersect with the current fvc(q), it has no chance to be contained by the final fvc(q). Hence, we can safely claim that an entry e is not a potential candidate as long as its intersection with the current fvc(q) is empty. However, e cannot be disregarded if it is an MBR. The reason is that e may contain points that could update (shrink) fvc(q) and disqualify those potential candidates identified in previous steps. As a result, a post-processing is required when $\mathcal{L}$ becomes empty, where entries in V_p are checked in sequence and points contained by those entries will be used to update fvc(q). Finally, all points in V_c are filtered once more using the final fvc(q).

Algorithm 1: PFC(Query q; R-tree T)

```
1  Initialize two empty vectors $V_C$ and $V_p$;
2  Initialize the priority queue $\mathcal{L}$ with $T$'s root node;
3  $\mathcal{L}$ orders entries in decreasing order of the minmaxdist;
4  Initialize fvc($q$)= $\mathcal{S}$;
5  while $\mathcal{L}$ is not empty do
6      Pop the head entry $e$ of $\mathcal{L}$;
7      if $e$ is a point then
8          Identify poly$_{qe}^{e}$ and set fvc($q$)=fvc($q$) $\bigcap$poly$_{qe}^{e}$;
9          if fvc($q$) $= \emptyset$ then return;
10         if $e \subseteq$fvc($q$) then  Push $e$ into $V_c$;
11     else
12         if $e \bigcap$ fvc($q$) is $\emptyset$ then push $e$ to $V_p$;
13         else
14             Let $u_1, \dots, u_f$ be children MBRs of node $e$;
15             for $i = 1, \dots, f$ do
16                 if $u_i \cap$fvc($q$) $\neq \emptyset$ then
17                     Insert the child node $u_i$ into $\mathcal{L}$;
18                 else  Insert the child node $u_i$ into $V_p$;
19 Update fvc($q$) using points contained by entries in $V_p$;
20 Filter points in $V_c$ using fvc($q$);
21 Output $V_c$; return;
```

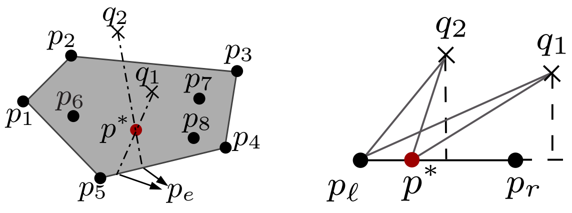
Consider the example in Figure 5, assuming p_2 is first processed and it updates fvc(q) as shown in Figure 5(a). Since p_2 is contained by fvc(q), it is added to V_c. Next, the MBR e is processed, obviously e could not offer any possible RFN of q as $e \bigcap$fvc(q) $= \emptyset$. However, e may contain points that update fvc(q) s.t. existing candidates are disqualified. Figure 5(b)) shows that p_3 from e could update fvc(q) and prune p_2. Hence, we need to add e to V_p for post-processing step.

PFC algorithm could terminate as soon as the fvc(q) becomes empty (Line 9 in Algorithm 1). In practice, many points due to their geometric locations will not have a furthest Voronoi cell in the space (this phenomena will be explained in details in the next Section). By searching the space in the *decreasing minmaxdist* fashion, for these points we could quickly shrink their furthest Voronoi cells to empty, leading to efficient early termination. In addition, when the query point does have a furthest Voronoi cell, the PFC algorithm calculates the RFN in one pass.

B. CHFC: the Convex Hull Furthest Cell algorithm

The PFC algorithm may scan a large number of points. Handling the false positives from the list of potential candidates in the post-processing step is expensive. An important lemma introduced next will significantly reduce the cost of searching for the RFN.

The convex hull $\mathcal{C}_P$ for a set of points P is the smallest convex polygon defined by points in P that fully contains P. We represent $\mathcal{C}_P$ as an *ordered set* of points that are vertices for the convex hull of P. For example, in Figure 6(a) $\mathcal{C}_P = \{p_1, p_2, p_3, p_4, p_5\}$. We abuse the notation of $\mathcal{C}_P$ a little bit



(a) from a convex. (b) from a line.

Fig. 6. Furthest point to p: Proof of Lemma 2

whenever the context is clear. It also represents the constrained area corresponding to P's convex hull. So $\mathcal{C}_P$ also denotes the shaded polygon $p_1p_2p_3p_4p_5p_1$ in Figure 6(a). Given a point q, we conjecture that the furthest point from P to q must be found in $\mathcal{C}_P$. This is indeed the case.

Lemma 2 *Given P and its convex hull $\mathcal{C}_P$, for a point q, let p^* =fn(q, P), then $p^* \in \mathcal{C}_P$.*

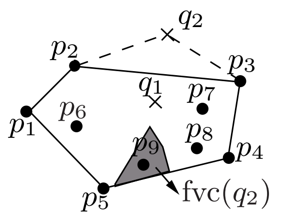
Proof: Suppose $\mathcal{C}_P = \{p_1, \dots, p_s\}$, then the boundary edges for $\mathcal{C}_P$ are $E(\mathcal{C}_P) = \{p_1p_2, \dots, p_sp_1\}$. First, we argue that p^* must locate on one of the boundary edges. Assume this is not correct, as shown in Figure 6(a), then $p^* \subset \mathcal{C}_P$. For both $q \subseteq \mathcal{C}_P$ (e.g. q_1) and $q \nsubseteq \mathcal{C}_P$ (e.g. q_2), the line that extends qp^* from p^* must intersect with an edge from $E(\mathcal{C}_P)$ as $\mathcal{C}_P$ is a convex. In our example, it intersect with p_4p_5 on a point p_e, then obviously $qp_e \geq qp^*$. Next, we can show that the furthest point p^* to q from a line segment $p_\ell p_r$ must be either p_ℓ or p_r. With the help of the perpendicular line from q to $p_\ell p_r$, as illustrated in Figure 6(b), this could be argued using proofs by contradiction again. This concludes that q^* must belong to $\mathcal{C}_P$. ■

Using Lemma 2 in a reverse angle, we can derive that only points in $\mathcal{C}_P$ will have reverse furthest neighbors w.r.t the data set P. In other words:

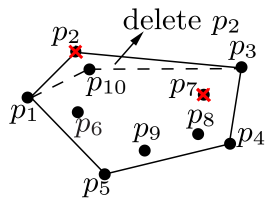
Lemma 3 *For a data set P and a point $p \in P$, fvc$(p, P) = \emptyset$ if $p \notin \mathcal{C}_P$. Furthermore, any point $p' \subset \mathcal{C}_P$ (here $\mathcal{C}_p$ denotes the area enclosed by the convex hull), fvc$(p', P \bigcup \{p'\}) = \emptyset$.*

Proof: The first claim is a special case of the second claim, as all points from $P - \mathcal{C}_P$ are *strictly* (not on the edges) contained by the polygon area defined by $\mathcal{C}_P$. Hence, consider a random point $p' \subset \mathcal{C}_P$, if $p' \notin P$, update P to $P' = P \bigcup \{p'\}$. By the property of the convex hull, it is straightforward to show that $\mathcal{C}_{P'} = \mathcal{C}_P$ as p' is strictly inside $\mathcal{C}_P$ (see Lemma 4). This indicates that $p' \notin \mathcal{C}_{P'}$. Now, for any random query point q in the space $\mathcal{S}$, by Lemma 2, fn$(q, P')\neq p'$ as $p' \notin \mathcal{C}'_P$. This indicates that there is not even a single point in the space will take p' as its furthest neighbor w.r.t P'. By the definition of the furthest Voronoi diagram, fvc$(p', P') = \emptyset$. ■

Combining this result with Lemma 1, another way to interpret Lemma 3 is that only the points from $\mathcal{C}_P$ will have the reverse furthest neighbors for a data set P. Given a query point q, we could view it as an insertion to P and obtain a new data set $P^* = P \bigcup \{q\}$. It is critical to decide whether



(a) Compute fvc(q) with only $\mathcal{C}_P$.



(b) Update $\mathcal{C}_P$ after deleting points.

Fig. 7. CHFC algorithm.

Algorithm 2: CHFC(Query q; R-tree T)

```
1 Compute $\mathcal{C}_P$ with $T$ using either the distance-priority or the depth-first algorithm [9];
2 if $q \subset \mathcal{C}_P$ then return $\emptyset$;
3 else
4     Compute $\mathcal{C}_{P^*}$ using $\mathcal{C}_P \bigcup \{q\}$;
5     Set fvc($q, P^*$) equal to fvc($q, \mathcal{C}_{P^*}$);
6     Execute a range query using fvc($q, P^*$) on $T$;
```

q belongs to the set of vertices defining the convex hull of P^*. A good news is that $\mathcal{C}_{P^*}$ could be computed efficiently from $\mathcal{C}_P$ and q alone, without looking at the rest of points in $P - \mathcal{C}_P$.

Lemma 4 *For a data set P and its convex hull $\mathcal{C}_P$, after adding a point q to P, if q is strictly contained by $\mathcal{C}_P$ ($q \subset \mathcal{C}_P$), then $\mathcal{C}_{P \bigcup \{q\}} = \mathcal{C}_P$; otherwise, $\mathcal{C}_{P \bigcup \{q\}} = \mathcal{C}_{\mathcal{C}_P \bigcup \{q\}}$.*

Proof: This straightforwardly follows from the definition of the convex hull. Figure 7(a) gives its intuition. ■

Finally, it remains as a problem how to compute q's furthest Voronoi cell efficiently when q is indeed a vertex for the convex hull of $P \bigcup \{q\}$. An important result is stated next that fvc(q, P) could be computed using only $\mathcal{C}_P$.

Lemma 5 *For a data set P and its convex hull $\mathcal{C}_P$, for a point $p \in \mathcal{C}_P$, fvc(p, P)=fvc($p, \mathcal{C}_P$).*

Proof: We use proof by contradiction. Suppose this claim is not correct. Then there must be a point $p' \in P - \mathcal{C}_P$ s.t. its furthest Voronoi cell could shrink the area enclosed by fvc($p, \mathcal{C}_P$). However, by Lemma 3, fvc($p', P = \emptyset$). This completes the proof. ■

1) The Main Algorithm: Lemma 3, 4 and 5 (together with Lemma 1) immediately yield an efficient algorithm for the MRFN problem. Assuming for now that we have obtained $\mathcal{C}_P$ (details will be discussed soon), given a query point q, we can quickly return empty if $q \subset \mathcal{C}_P$. Otherwise, we set $P^* = P \bigcup \{q\}$ and compute $\mathcal{C}_{P^*}$ using only $\mathcal{C}_P$ and q. Next, the furthest Voronoi cell of q in P^* is calculated using only $\mathcal{C}_{P^*}$. Finally, a range query using fvc(q, P^*) is executed on the R-tree of P to retrieve the RFN of q. This is referred as the convex hull furthest cell (CHFC) algorithm.

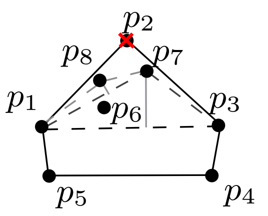
Figure 7(a) demonstrates CHFC's idea. If the query point is q_1, since $q_1 \subset \mathcal{C}_P$, CHFC algorithm will immediately return empty for q_1's RFN. Lemma 4 guarantees that $\mathcal{C}_{P \bigcup \{q_1\}} = \mathcal{C}_P$. Then, by Lemma 3 fvc$(q_1, P \bigcup \{q_1\}) = \emptyset$. Hence, by Lemma 1, there will have no points taking q_1 as their furthest neighbors among all points in P. On the other hand, if the query point is q_2 where $q_2 \not\subset \mathcal{C}_P$, CHFC algorithm will obtain $P \bigcup \{q_2\}$'s convex hull by applying Lemma 4. Essentially, only points from $\mathcal{C}_P$ and q_2 (i.e., $\{p_1, p_2, p_3, p_4, p_5, q_2\}$) will be used to compute the convex hull of $P \bigcup \{q_2\}$. The updated convex hull is simply $p_1p_2q_2p_3p_4p_5p_1$. Next, since q_2 is one of the points in this convex hull, we need to compute its furthest Voronoi cell and Lemma 5 achieves this using only points from $\mathcal{C}_{P \bigcup \{q_2\}}$. Finally, q_2's RFN includes all points in the area covered by its furthest Voronoi cell as Lemma 1 states. In this case, its RFN p_9 will be successfully retrieved.

The efficiency of the CHFC algorithm is achieved in two folds. First of all, all query points that are enclosed by $\mathcal{C}_P$ are extremely fast to deal with. Secondly, for the rest of query points, computing their furthest Voronoi cells becomes much more efficient by taking into account only the points in $\mathcal{C}_P$ (instead of doing this via P). In practice, $|\mathcal{C}_P| << |P|$. Of course, this reduction in size may not always be the case. It is fairly easy to construct examples where $P = \mathcal{C}_P$, e.g., P only contains points on edges of a rectangle. Nevertheless, for most real life data sets we expect that $|\mathcal{C}_P| << |P|$ (or at least $|\mathcal{C}_P| < |P|$) holds and we could store $\mathcal{C}_P$ in main memory. This fact is verified in our experiments. For the case when this is violated, i.e., $\mathcal{C}_P$ is still too large to be stored in main memory, one can use the PFC algorithm or use the approximate technique we will discuss in Section V-A.

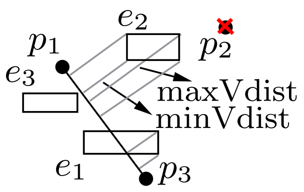
2) Computing P's convex hull: There are I/O efficient algorithms for computing the convex hulls of disk-based data sets. Specifically, convex hulls in two dimension can be computed in external memory with the help of sorting in external memory [16]. This means that we can find the convex hull of the disk-based data set P ($|P| = n$) with $\mathcal{O}(\frac{n}{B} \log_M \frac{n}{MB})$ I/Os for a main memory buffer with M pages, assuming the page size is B. From a practical point of view and to deal with data in higher dimensions, one would like to compute the convex hull of P using a R-tree. Fortunately, this problem has been studied in the literature and one can apply either the distance-priority or the depth-first algorithm from [9]. The complete CHFC algorithm is presented in Algorithm 2.

If P is static, an obvious optimization for the CHFC algorithm is to pre-compute $\mathcal{C}_P$ and avoid the call to the distance-priority (or depth-first) algorithm at Line 1 completely. As argued above, we assume $\mathcal{C}_P$ can fit into main memory in this case, hence, any main memory convex hull algorithm could be applied in Line 4. In fact, one could apply main memory dynamic convex hull algorithm [13] here to get an almost logarithm computation bound.

3) Dynamically Maintaining $\mathcal{C}_P$: When P is static, CHFC algorithm is extremely efficient after pre-computing $\mathcal{C}_P$ once. However, under dynamic insertion and deletion to P, CHFC has to compute $\mathcal{C}_P$ for every query and this could greatly degrade the query performance. If P fits in main memory, recent theoretical study has confirmed that dynamically maintaining



(a) The qhull algorithm.



(b) Adapt qhull to R-tree.

Fig. 8. Dynamic update $\mathcal{C}_P$ using R-tree.

the convex hull of P could be done almost in logarithm time [13]. The challenge is how to do this efficiently utilizing R-tree when P is disk-resident.

We distinguish two cases, namely, point insertion and point deletion. Point insertion is easy to handle as suggested by the Lemma 4, assuming $\mathcal{C}_P$ fits in memory. The main obstacle is when point is deleted as shown in Figure 7(b). Once again, the easy case is when the deleted point p is inside the convex hull of P ($p \subset \mathcal{C}_P$), e.g., p_7 in Figure 7(b). Obviously, in this situation $\mathcal{C}_P$ will not be affected by the deletion. On the other hand, when the deleted point p is an existing vertex for the convex hull of P ($p \in \mathcal{C}_P$), updating $\mathcal{C}_P$ is necessary. For example, deleting p_2 in Figure 7(b) will eventually lead to adding p_{10} to $\mathcal{C}_P$.

When $p \in \mathcal{C}_P$ is deleted, we observe that most part of $\mathcal{C}_P$ is intact except for its left and right neighbors (denoted as p_ℓ and p_r). New vertices may be introduced in between p_ℓ and p_r. A straightforward solution is to retrieve all points, using R-tree, contained by the polygon $p_\ell p p_r p_\ell$ ($p_1p_2p_3p_1$ in Figure 7(b)) and find the local convex hull for these points in main memory. But this could be potentially expensive (if $p_\ell p p_r p_\ell$ encloses large number of points).

A better approach is to adapt the qhull algorithm [7] to R-tree in this setting. In the qhull algorithm, it is shown that given two vertices p_ℓ and p_r in the convex hull, a vertex (if it indeed exists) in the final convex hull that locates in between p_ℓ and p_r could be located by finding the point that has the largest perpendicular distance to $p_\ell p_r$. This is true for both sides of $p_\ell p_r$. Recursively applying this step, all vertices in the final convex hull that locate in between p_ℓ and p_r could be identified.

In our case, we only need to search for points on one side of the $p_\ell p_r$, i.e., the side that the deleted point p locates at. An example demonstrating this idea is shown in Figure 8(a). When p_2 is deleted, we find $p_\ell = p_1$ and $p_r = p_3$ from $\mathcal{C}_P$. Searching along the side that p_2 locates at, the point with the largest perpendicular distance to p_1p_3 is p_7. It is added to $\mathcal{C}_P$. Next, we recursively apply the same step to p_1p_7 and p_7p_3, still only to the side that p_2 locates at. In this case, p_8 will be retrieved and one more recursion using p_1p_8 and p_8p_7 will terminate the search. The updated convex hull becomes $\mathcal{C}_P = \{p_1, p_8, p_7, p_3, p_4, p_5\}$. The only puzzle left is to doing these steps efficiently in R-tree.

Using an R-tree, our problem essentially reduce to the following. Given a line segment $p_\ell p_r$, find the point from the R-tree that has the largest perpendicular distance to $p_\ell p_r$ and it must locate on one specified side of $p_\ell p_r$. This further reduces to the next problem, given a line segment $p_\ell p_r$, and an MBR e, what is the maximum and minimum of the maximum possible perpendicular distances from a point in the MBR e to the line segment $p_\ell p_r$?

Algorithm 3: MaxVP(Query p_ℓ, p_r, p; R-tree T)

```
1  Initialize the priority queue $\mathcal{L}$ with $T$'s root node;
2  $\mathcal{L}$ orders entries in decreasing order of their maxVdist;
3  $\mathcal{L}$.minVdist keeps the max minVdist among all its entries;
4  while $\mathcal{L}$ is not empty do
5      Pop the head entry $e$ of $\mathcal{L}$;
6      if $e$ is point then return $e$;
7      else
8          for each child $u_i$ of $e$ do
9              minVdist$_{u_i} = \infty$, maxVdist$_{u_i} = -1$;
10             for each corner point $x$ of $u_i$ do
11                 if $x$ and $p$ are on the same side of $p_\ell p_r$ then
12                     Get the perpendicular distance $s_x$ from $x$ to $p_\ell p_r$;
13                     $s_x$ updates minVdist$_{u_i}$, maxVdist$_{u_i}$;
14             if maxVdist$_{u_i} >$ $\mathcal{L}$.minVdist then
15                 Insert $u_i$ into $\mathcal{L}$;
16                 if minVdist$_{u_i} >$ $\mathcal{L}$.minVdist then
17                     $\mathcal{L}$.minVdist=minVdist$_{u_i}$;
```

Lemma 6 *For an axis-parallel MBR e, both the maximum and minimum of the maximum possible perpendicular distances from a point in the MBR e to any line segment $p_\ell p_r$ are bounded by the corner points of e.*

Proof: W.l.o.g. assume that $p_\ell p_r$ and e are in the positive quadrant of the plane. Let $\hat{c}$ be the unit vector perpendicular to $p_\ell p_r$ and point to the side of $p_\ell p_r$ we are interested in. We need to find $x^* = \arg\max_{x \in e} \hat{c}^T x$. Note that this is a linear program with four linear constraints, is feasible and bounded. Hence by the fundamental theorem of Linear Programming, one of the vertices of e must yield the optimal solution [11]. The minimization problem can be handled similarly. A similar problem has also been studied by [9]. ■

With Lemma 6, it is possible to migrate the aforementioned algorithm for the point deletion to an R-tree efficiently. We use minVdist and maxVdist to denote the minimum and maximum of the maximum possible perpendicular distance for a point from an MBR e to a line segment $p_\ell p_r$. A special note in our setting is that minVdist and maxVdist are only bounded by the corner points of e that locate on the same side of $p_\ell p_r$ as the deleted point p. Consider the example in Figure 8(b), suppose $p_2 \in \mathcal{C}_P$ has been deleted. Its left and right neighbors in $\mathcal{C}_P$ are p_1 and p_3 respectively. First, we can safely prune all MBRs that locate completely on the other side of p_1p_3, e.g., e_3. Next, for the rest of MBRs, we calculate their minVdist

Algorithm 4: DynamicCH($\mathcal{C}_P$, op, p; R-tree T)

```
1  if op is Insertion then
2      if $p \subset \mathcal{C}_P$ then return $\mathcal{C}_P$;
3      else return $\mathcal{C}_{\mathcal{C}_P \bigcup \{p\}}$;
4  else if op is Deletion then
5      if $p \subset \mathcal{C}_P$ then return $\mathcal{C}_P$;
6      else
7          Find $p$'s left (right) neighbor, $p_\ell$ ($p_r$), in $\mathcal{C}_P$;
8          $\mathcal{C}_P$: $\{\dots, p_\ell, p, p_r, \dots\} \longmapsto \{\dots, p_\ell, p_r, \dots\}$;
9          Call QHullRtree($p_\ell, p_r, p, \mathcal{C}_P, T$);
   /* qhull algorithm adapted to R-tree */
10 QHullRtree ($p_\ell, p_r, p, \mathcal{C}_P$; R-tree $T$)
11 ------------------------------------------------
12 Let $p'$ =MaxVP($p_\ell, p_r, p, T$);
13 if $p' = \emptyset$ then return;
14 else
15     $\mathcal{C}_P$: $\{\dots, p_\ell, p_r, \dots\} \longmapsto \{\dots, p_\ell, p', p_r, \dots\}$;
16     QHullRtree($p_\ell, p', p, \mathcal{C}_P, T$);
17     QHullRtree($p', p_r, p, \mathcal{C}_P, T$);
```

and maxVdist using their corner points that locate on the same side of p_1p_3 as p_2. Note that we can safely prune an MBR if its maxVdist is smaller than the minVdist of some other MBRs, e.g., e_1 can be pruned in the presence of e_2 as the best point from e_1 cannot possibly beat at least one point from e_2 (in terms of the perpendicular distance to the query line segment). We exploit the R-tree nodes in a priority queue that orders entries by the decreasing maxVdist value. The search could terminate as soon as we find one point at the head of the queue or when the queue becomes empty. The Max-Vertical-Point (MaxVP) algorithm is listed in details in Algorithm 3.

With these discussions, we could dynamically maintain $\mathcal{C}_P$ using the R-tree. This reduces the cost of the CHFC algorithm by avoiding the call on Line 1 in Algorithm 2. The DynamicCH algorithm in Algorithm 4 summarizes our idea presented above. We would like to point out that Algorithm 4 is presented for the two-dimensional case. For higher dimensions, certain generalization must be adapted. For example, instead of looking at one side for a segment from the convex hull, we need to examine one side of a plane.

V. Bichromatic Reverse Furthest Neighbors

After resolving all the difficulties for the MRFN problem in Section IV, solving the BRFN problem becomes almost immediate. From the discussion in Section IV, all points in P that are contained by fvc(q_i, Q) will have q_i as their furthest neighbor. This immediately implies that BRFN $(q, Q, P) = \{p \in P \wedge p \in \text{fvc}(q, Q)\}$. Furthermore, Lemma 5 guarantees that fvc(q, Q)=fvc($q, \mathcal{C}_Q$). For example, in Figure 9 the reverse furthest neighbor of q_1 is $\{p_4\}$ which is contained in the fvc(q_1, Q) $=$ fvc($q_1, \mathcal{C}_Q$). Lemma 3 indicates that only those points in $\mathcal{C}_Q$ will have reverse furthest neighbors. Hence, an efficient R-tree based query algorithm for finding the RFN of

Algorithm 5: BRFN(Query q, Q; R-tree T)

```
1  Compute the convex hull $\mathcal{C}_Q$ of $Q$;
2  if $q \subset \mathcal{C}_Q$ then return $\emptyset$; //$q \notin \mathcal{C}_Q$
3  else
4      Compute fvc($q, \mathcal{C}_Q$);
5      Execute a range query using fvc($q, \mathcal{C}_Q$) on $T$;
```

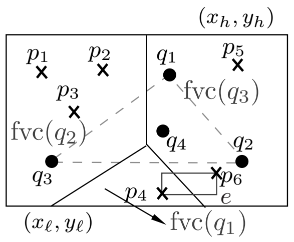


Fig. 9. The BRFN algorithm.

q is to only check those MBRs that intersect with fvc($q, \mathcal{C}_Q$) when $q \in \mathcal{C}_Q$, e.g., in Figure 9 we need to access the MBR e that intersects with fvc(q_1) to retrieve its reverse furthest neighbors. If $q = q_4$, we can return $\emptyset$ immediately. Algorithm 5 details this method.

A. Disk-Resident Query Group

One limitation with our discussions so far is the problem of handling the query group with a massive size that do not fit in internal memory. As we have discussed, one could use the convex hull of Q to reduce the number of points in the query group. There are I/O efficient algorithms for computing the convex hulls of disk-based data sets. In two dimension $\mathcal{C}_Q$ could be found with $\mathcal{O}(\frac{m}{B} \log_M \frac{m}{MB})$ I/Os [16] for a main memory buffer with M pages, assuming the page size is B and $|Q| = m$. Alternatively, if Q is indexed by R-tree, we could simply use the algorithm from [9]. For most cases, we expect that $|\mathcal{C}_Q| \ll |Q|$. However, one could easily construct special cases where $|\mathcal{C}_Q| = |Q|$, e.g., all points in Q are vertices of a convex polygon. To handle such special instances, we propose to obtain an approximate convex hull of Q using Dudley's approximation [36]. Dudley's construction generates an approximate convex hull of Q (denote it as $\mathcal{AC}_Q$) with $\mathcal{O}(1/\epsilon^{(d-1)/2})$ vertices with maximum *Hausdorff distance* of ϵ to the convex hull of Q. The *Hausdorff distance* measures how far two convex polygons S_1 and S_2 are from each other. Informally, the Hausdorff distance between S_1 and S_2, is the longest distance an adversary can force one to travel by choosing a point in one of the two sets, from where one then must travel to the other set. Formally, let X and Y be the vertices of S_1 and S_2 respectively, then:

$$d_H(S_1, S_2) = \max\big(\sup_{x \in X} \inf_{y \in Y} ||x - y||, \sup_{y \in Y} \inf_{x \in X} ||x - y||\big)$$

For fixed dimensions, this means that we can always get an approximate convex hull of Q with constant number of vertices. Obviously, the smaller ϵ is, the more accurate the approximation is and the larger the size of the approximate convex hull is (since Dudley's approximation is an inner approximation of the convex hull).

Roughly speaking, the edges of of $\mathcal{AC}_Q$ are within ϵ distances from edges of $\mathcal{C}_Q$. Clearly, there is a trade-off between the approximation quality and the size of $\mathcal{AC}_Q$. Henceforth, for a disk-based query group Q, we first compute its convex hull using the I/O efficient algorithm. If $\mathcal{C}_Q$ is still too large to fit in main memory, we replace $\mathcal{C}_Q$ with Dudley's approximation and specify the size of $\mathcal{AC}_Q$ of our choice.

The Dudley's approximation was proposed to work with main memory data sets [36]. For our purpose, we need to extend it to work with external data sets. To compute $\mathcal{AC}_Q$ in external memory, one can build an index that supports nearest neighbor search with logarithm IOs guarantees (such as the BBD tree[2]) on $\mathcal{O}(1/\epsilon^{(d-1)/2})$ points scattered on the sphere containing the convex hull of Q and then do a scan of the convex hull and compute the reverse nearest neighbor of each point read from $\mathcal{C}_Q$. This approach requires only linear I/Os on $\mathcal{C}_Q$ (excluding the I/Os to compute $\mathcal{C}_Q$). The following lemma summarizes this result. The details of this approach will appear in the full version.

Lemma 7 *For a disk-based query group Q, one could always get a query group Q' with size $\mathcal{O}(1/\epsilon^{(d-1)/2})$ using $\mathcal{O}(\frac{m}{B}\log_M \frac{m}{MB})$ I/Os, where $m = |Q|$, B is the page size and M is the number of pages in the main memory, s.t. the Hausdorff distance between edges in $\mathcal{C}_{Q'}$ and $\mathcal{C}_Q$ is at most ϵ.*

The introduction of Dudley's approximation only creates an additive error ϵ on the corresponding edges between $\mathcal{C}_Q$ and $\mathcal{AC}_Q$. Hence, for a query point q and query group Q, if $\mathcal{C}_Q$ is too large to fit in main memory, we calculate $\mathcal{AC}_Q$. If q is inside the convex polygon defined by $\mathcal{AC}_Q$ and away from any edge of this polygon by more than ϵ, then it does not have reverse furthest neighbors. Otherwise, we find the nearest neighbor of q in the vertices of $\mathcal{AC}_Q$, say q', and apply algorithm 5 using q' and $\mathcal{AC}_Q$. This algorithm is referred as the A-BRFN algorithm and our experiment confirms that it has high accuracy.

As a final note, the approximate convex hull idea could be also applied to the CHFC algorithm for the MRFN problem in Section IV-B when $\mathcal{C}_P$ is too large to fit in main memory.

VI. Experiment

All proposed algorithms have been implemented into the widely used, disk-based spatial index library [1], Standard geometric operations, e.g. convex polygon intersection, are provided by the CGAL library [2]. Finally, our I/O efficient approximate convex hull algorithm is developed based on the library from [36]. All experiments were executed on a Linux machine with an Intel 2GHz cpu. For both R-tree and heapfiles the page size B is set to 4KB.

Data sets. The real data sets were obtained from the *digital chart of the world server* where points define the road networks from California (CA), San Francisco (SF) and USA (US). CA also contains large number of points of interest (e.g, restaurants, resorts). These data sets are available online [3]. To create a data set of larger size, we merge them into one data set (denoted as *Map*) after normalizing each data set into the space $L = (0, 0) \times (100000, 100000)$. Points in CA, SF and US have various degrees of skew distributions in the space. Map contains $476,578$ number of points. There are three kinds of synthetic data sets used in the experiment: uncorrelated uniformly generated (UN) points, correlated bivariate (CB) points and random-cluster distribution (R-Cluster) points in L. An example of the CB and R-cluster data sets is illustrated in Figure 10. In this work we concentrate on the two-dimensional space. How to generalize our algorithms to higher dimensions as well as the experiments for those are interesting extensions to this work.

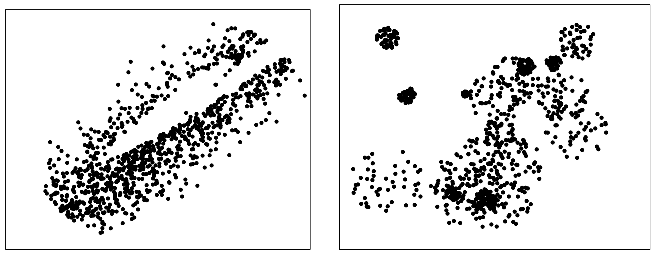
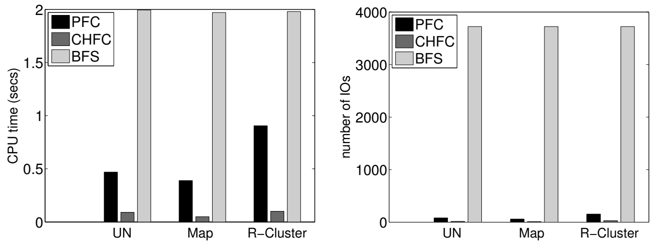

(a) correlated bivariate (CB). (b) random clusters (R-Cluster).

Fig. 10. Different distribution types of synthetic data sets and query groups.



(a) cpu time. (b) Number of IOs.

Fig. 11. MRFN algorithms: cpu computation and IOs analysis.

Performance measurement. For all algorithms, we measured their performance using two metrics, namely, the *CPU time* and the *number of IOs*. We would like to highlight that the CPU time measure the *pure computation cost* of an algorithm. It is NOT the total execution time of the algorithm. The total execution time is simply the CPU time plus the IO cost. In most cases, the IO cost dominates the overall execution time. Finally, by default *1,000* queries were generated for each experiment and we report the average for *one query*.

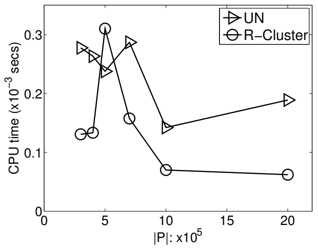
A. Algorithms for the MRFN *problem*

Experiment Setup. For the MRFN problem, we generated synthetic data sets whose size are equal to the size of Map data set. The query point is randomly selected from the space L. For brevity, only the results from the UN, R-Cluster and Map data sets are reported.

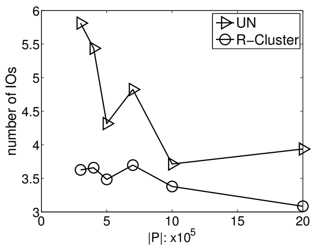
[1]www.research.att.com/∼marioh/spatialindex/index.html

[2]www.cgal.org

[3]www.cs.fsu.edu/∼lifeifei/SpatialDataset.htm

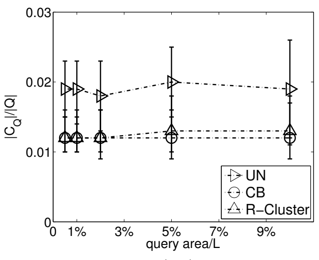


(a) cpu time.

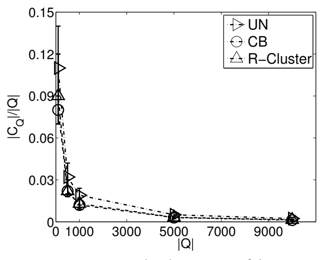


(b) Number of IOs.

Fig. 12. DynamicCH algorithms: deleting vertex of $\mathcal{C}_P$.



(a) vary $\mathcal{A}$, $|Q| = 1000$.



(b) vary $|Q|$, $\mathcal{A} = 3\%$.

Fig. 13. Pruning with the convex hull of Q.

The PFC and CHFC algorithms. We first study the case when P is static. In this setting, we can optimize the CHFC algorithm by pre-computing P's convex hull using R-tree and store $\mathcal{C}_P$ for query processing. Figure 11 compares the cost of these two algorithms against the brute-force search (BFS) based approach. To make the comparison fair, we optimize the BFS algorithm by terminating the search as early as possible. For a point $p \in P$, we first calculate its distance to q and then calculate its distance to every other point p' in P. However, whenever there is a p' s.t. $|p - p'| > |p - q|$, we know for sure that $p \notin$MRFN $(q, P \bigcup \{q\})$ and the BFS algorithm continues to the next point in P. Clearly, Figure 11 indicates that both PFC and CHFC outperform the BFS algorithm in terms of both the CPU cost and the number of IOs. Especially, for the dominant cost of IOs, both algorithms are at least two orders of magnitude better.

Among the three data sets, BFS algorithm has almost constant computation cost as well as the number of IOs. For both the PFC and CHFC algorithms, R-Cluster data set results in higher costs. This is easily explained by the more scattered geometric locations of points in the R-Cluster case. When $|P|$ is static and $\mathcal{C}_P$ is known, it is easy to expect that CHFC outperforms PFC, as for a large number of query points CHFC could be extremely efficient. This is confirmed by Figure 11.

Dynamically maintaining $\mathcal{C}_P$. One advantage of the PFC algorithm is that it requires no additional effort to work with dynamic data set. On the other hand, when insertion and deletion of points happen in P, CHFC algorithm has to dynamically maintain $\mathcal{C}_P$ using Algorithm 4. Following our discussions, insertion of points is easy to handle and the worst case is the deletion of a point p that happens to be one of the vertices in $\mathcal{C}_P$. Section IV-B.3 discussed this problem in details and we executed an experiment to verify the efficiency of the DynamicCH algorithm in this worst case. For a data set P, we first computed its $\mathcal{C}_P$ using the R-tree based algorithm from [9]. Then, for each point p in $\mathcal{C}_p$, we deleted p and computed the new convex hull for $P - \{p\}$. Note that, for each such point p we always started with the original $\mathcal{C}_P$ and P. The average cost of maintaining the convex hull after deleting one point in $\mathcal{C}_P$, in terms of both the cpu computation and number of IOs, of our DynamicCH algorithm is shown in Figure 12. They essentially reflect the cost of QHullRtree algorithm based on the MaxVP algorithm when we randomly delete one point from $\mathcal{C}_P$. We repeated this experiment for data sets of different sizes as well as different distributions. Figure 12 clearly shows that our approach is highly effective, with cpu computation cost at around 0.2 milliseconds and IOs lower than 10. More importantly, our algorithm scales nicely with the increase in size for the data set P. Uniform distributed data set generates higher maintenance cost as the chance of containing more points in the polygon area $p_\ell p p_r p_\ell$ is higher.

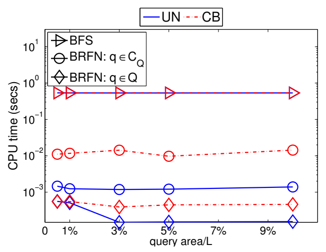
Given these results, we can safely claim that CHFC is extremely efficient in practice. Of course, there definitely exists data sets such that dynamically maintaining the convex hull could be expensive ($p_\ell p p_r p_\ell$ contains large number of points), or $\mathcal{C}_P$ does not fit into main memory. We could use PFC algorithm in those cases.

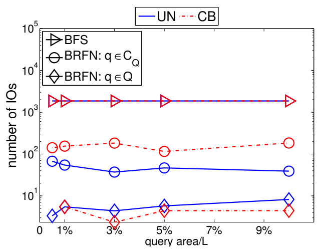
B. *The* BRFN *algorithm*

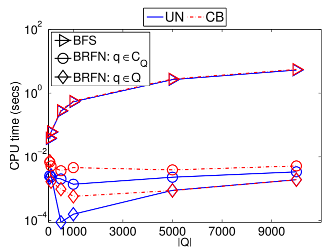
Experiment Setup. For the BRFN problem, the cost of the query depends on several critical factors. They include the location of Q, the range (its area $\mathcal{A}$) of Q and how points are distributed within the range. Lastly, the size of Q also plays a role. Given these observations, a random query is generated as follows. We specify $|Q|$ and its $\mathcal{A}$ (as a percentage of the area for the entire space L). Next a random location in L is selected as the center of the query group. Three types of distributions discussed above were used to generate $|Q|$ number of points within the specified area. Examples of query groups with these distributions are given in Figure 10. In addition, we randomly select a query point $q \in Q$ after generating Q. We distinguish two cases: q is randomly selected from Q (R_Q strategy) or q is randomly selected from $\mathcal{C}_Q$ ($R_{\mathcal{C}_Q}$ strategy).

Pruning power of $\mathcal{C}_Q$. Figure 13 shows that for various query distributions, $\mathcal{C}_Q$ could significantly reduce the size of the query group. We plot the averages together with the $5\% - 95\%$ confidence interval. For $Q = 1000$, $|\mathcal{C}_Q|$ is only about 2% of $|Q|$. Furthermore, the pruning power of $\mathcal{C}_Q$ increases (see Figure 13(b)) for larger $|Q|$'s as $|\mathcal{C}_Q|$ grows at a much slower pace than $|Q|$ does. Lastly, Figure 13(a) indicates that $\frac{|\mathcal{C}_Q|}{|Q|}$ is roughly a constant over the query area and UN distributed query groups have larger $|\mathcal{C}_Q|$ over correlated and clustered groups. In most of our experiments, the results from the CB and R-Cluster query groups are quite similar. Hence, we only report the results from UN and CB query groups.

Algorithm BRFN**.** The BRFN algorithm (Algorithm 5) takes advantage of the convex hull of Q and converts the BRFN


(a) cpu: vary $\mathcal{A}$, $Q = 1000$.


(b) IOs: vary $\mathcal{A}$, $Q = 1000$.


(c) cpu: vary $|Q|$, $\mathcal{A} = 3\%$.

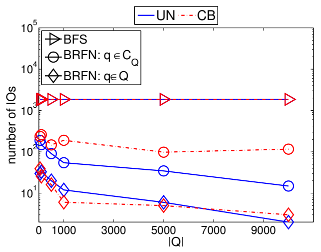
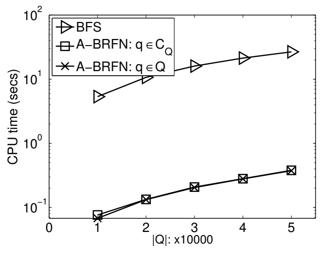

(d) IOs: vary $|Q|$, $\mathcal{A} = 3\%$.

Fig. 14. BRFN algorithms: cpu computation and IOs analysis, Map data set.


(a) cpu time.

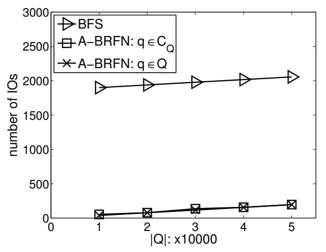

(b) IOs.

Fig. 15. Approximate convex hulls: $\mathcal{A} = 3\%$, Map data set.

problem into a simple range query with a convex polygon. Figure 14 confirms its superior performance over the brute-force search (BFS) algorithm. The BFS algorithm in the BRFN case simply calculates the furthest neighbor among all points in Q for each point $p \in P$. If it is equal to q, then p is included in the BRFN (q, Q, P). Note that in Figure 14 all the y-axes are plotted in the log scale. The BRFN algorithm is in general 2 to 4 orders of magnitude more efficient than the naive approach and the gap in performance is enlarging as $|Q|$ increases (Figure 14(c) and 14(d)). The cost of the BRFN algorithm does increase w.r.t the increase in the area of the query group (Figure 14(a) and 14(b)), but it is not very sensitive to such increases as shown in the log scale. For the BRFN algorithm, if q is a random point from $Q - \mathcal{C}_Q$, obviously no range query is required on the R-tree and its query cost is tiny. Given the fact that majority of points in Q will not be a member of $\mathcal{C}_Q$ (as shown by the convex hull pruning experiments from this section), the BRFN algorithm will be highly efficient for the case of $q \in Q$ as reflected by Figure 14. The UN type of query groups reduces the cost of the BRFN algorithm compared to the CB type in Figure 14, in terms of both cpu computation and IOs. The furthest cells of query points are distributed more evenly in the UN query groups due to the uniformity. This indicates smaller furthest cells than those from the CB query groups. Hence, BRFN is more efficient on the UN type of query groups.

Finally, we observe an interesting pattern for the IOs cost of the BRFN algorithm for query groups of different sizes. It is in fact decreasing while $|Q|$ increases (Figure 14(d)). Recall that $|\mathcal{C}_Q|$ increases much slower than $|Q|$ does. Hence, for the R_Q strategy, the chance that $q \in \mathcal{C}_Q$ is getting smaller. This will reduce the overall cost. On the other hand, for the $R_{\mathcal{C}_Q}$ strategy, as $|\mathcal{C}_Q|$ still contains more points when $|Q|$ increases, the furthest cell of an individual point $q \in \mathcal{C}_Q$ is smaller which leads to the reduction in the query cost.

C. *Query groups of large size*

Experiment Setup. Concentrating on the BRFN problem, we use the same setup as in Section VI-B, but with larger query groups. The algorithm proposed in Section V-A for the BRFN query has much lower cost than the brute-force search and very good approximation qualities. For brevity those results were omitted and we focused on reporting its query efficiency.

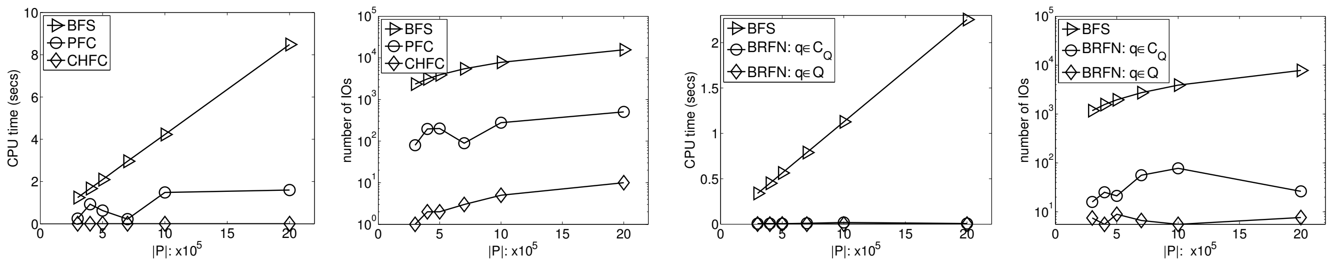
Approximate convex hull based algorithm. There are cases where not only $|Q|$ is large, but also $|\mathcal{C}_Q|$. These cases rarely happen in practice. But it is not entirely impossible. Efforts have been devoted in Section V-A to deal with such scenarios when they do arise. The idea is to develop IO efficient approximate convex hulls ($\mathcal{AC}_Q$). Figure 15 reports the performance of such an approach. The algorithm we have developed could be applied on the $\mathcal{AC}_Q$ instead of the $\mathcal{C}_Q$ and we denote it as A-BRFN. Our algorithm outperforms the basic brute-force search by orders of magnitude. By treating Q as external memory data sets, computing its approximate convex hull incurs a cost that is similar to the sorting in external memory. Hence, we see an increase in terms of both cpu computation and IOs for our algorithm when $|Q|$ increases.

From the discussion in Section V-A, the error introduced by the approximation is independent of $|Q|$ and is only determined by $|\mathcal{AC}_Q|$. When $|\mathcal{AC}_Q| = 20$, ϵ is 0.01 in two dimension. Hence, our algorithm achieves excellent approximation qualities.

D. *Scalability of various algorithms*

Experiment Setup. Finally, we investigate the scalability of our algorithms w.r.t the size of the data set P. We use the same setup as in Section VI-A and VI-B, but with R-Cluster data sets of different sizes as P.

Scalability. For the BRFN problem, we fix the query area as $\mathcal{A} = 3\%$ and $|Q| = 1000$. The CB query type is used. Figure 16 shows the scalability of various algorithms. More specifically, Figure 16(a) and 16(b) show the results for the MRFN queries. Figure 16(c) and 16(d) show the results for the BRFN queries. Note that in both Figure 16(b) and 16(d), the number of IOs were plotted in the log scale. Compared to the brute-force search, all of our algorithms scale nicely with the increase in size of $|P|$. When $|P| = 2,000,000$, BFS takes



(a) MRFN: cpu time. (b) MRFN: number of IOs. (c) BRFN: cpu time. (d) BRFN: number of IOs.

Fig. 16. Scalability: cpu computation and IOs, R-Cluster data set.

more than $10,000$ IOs whereas our algorithms take from only a few to less than 100 IOs.

VII. Conclusion

This paper studies the reverse furthest neighbor queries that has many real life applications. Our work solves the RFN queries in both monochromatic and bichromatic versions. We present R-tree based, efficient algorithms for both MRFN and BRFN problems with excellent pruning capability. All of our algorithms allow dynamic updates to the data sets. Furthermore, it has been adapted to work with disk-resident query groups in the BRFN case. Future work includes generalizing our algorithms to higher dimensions, dealing with moving points and continuous queries, and answering RFN queries in a road-network or any Ad-Hoc subspaces.

VIII. Acknowledgment

We would like to thank the anonymous reviewers for their valuable comments that improve this work. Bin Yao is supported in part by CRC Planning Grant from FSU. Piyush Kumar is supported in part by NSF grant CCF-0643593.

References

[1] E. Achtert, C. Böhm, P. Kroger, P. Kunath, A. Pryakhin, and M. Renz, "Efficient reverse k-nearest neighbor search in arbitrary metric spaces," in *SIGMOD*, 2006.

[2] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu, "An optimal algorithm for approximate nearest neighbor searching in fixed dimensions," *Journal of ACM*, vol. 45, no. 6, pp. 891–923, 1998.

[3] F. Aurenhammer, "Voronoi diagrams - a survey of a fundamental geometric data structure," *ACM Computing Survey*, vol. 23, no. 3, pp. 345–405, 1991.

[4] N. Beckmann, H. P. Kriegel, R. Schneider, and B. Seeger, "The R*-tree: an efficient and robust access method for points and rectangles," in *SIGMOD*, 1990.

[5] R. Benetis, C. S. Jensen, G. Karciauskas, and S. Ŝaltenis, "Nearest and reverse nearest neighbor queries for moving objects," *The VLDB Journal*, vol. 15, no. 3, pp. 229–249, 2006.

[6] S. Berchtold, C. Böhm, D. A. Keim, and H. P. Kriegel, "A cost model for nearest neighbor search in high-dimensional data space," in *PODS*, 1997.

[7] M. Berg, M. Kreveld, M. Overmars, and O. Schwarzkopf, *Computational geometry: algorithms and applications*. Springer, 1997.

[8] C. Böhm, "A cost model for query processing in high dimensional data spaces," *ACM Transaction on Database Systems*, vol. 25, no. 2, pp. 129–178, 2000.

[9] C. Böhm and H.-P. Kriegel, "Determining the convex hull in large multidimensional databases," in *DaWaK*, 2001.

[10] S. Cabello, J. M. Diaz-Banez, S. Langerman, C. Seara, and I. Ventura, "Reverse facility location problems," *European Journal of Operational Research*, vol. to appear, 2008.

[11] Chvatal and Vasek, *Linear Programming*, ser. A Series of Books in the Mathematical Sciences. New York: W. H. Freeman and Company, 1983.

[12] B. Cui, B. C. Ooi, J. Su, and K.-L. Tan, "Contorting high dimensional data for efficient main memory kNN processing," in *SIGMOD*, 2003.

[13] E. D. Demaine and M. Patrascu, "Tight bounds for dynamic convex hull queries (again)," in *SoCG*, 2007.

[14] K. Deng, X. Zhou, H. T. Shen, K. Xu, and X. Lin, "Surface k-NN query processing," in *ICDE*, 2006.

[15] S. P. Fekete and A. Kroller, "Geometry-based reasoning for a large sensor network," in *SoCG*, 2006.

[16] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter, "External-memory computational geometry," in *FOCS*, 1993.

[17] A. Guttman, "R-trees: a dynamic index structure for spatial searching," in *SIGMOD*, 1984.

[18] G. R. Hjaltason and H. Samet, "Distance browsing in spatial databases," *ACM Trans. Database Syst.*, vol. 24, no. 2, 1999.

[19] H. V. Jagadish, B. C. Ooi, K.-L. Tan, C. Yu, and R. Zhang, "iDistance: An adaptive B$^+$-tree based indexing method for nearest neighbor search," *ACM Trans. Database Syst.*, vol. 30, no. 2, pp. 364–397, 2005.

[20] J. M. Kang, M. F. Mokbel, S. Shekhar, T. Xia, and D. Zhang, "Continuous evaluation of monochromatic and bichromatic reverse nearest neighbors," in *ICDE*, 2007.

[21] M. R. Kolahdouzan and C. Shahabi, "Voronoi-based k nearest neighbor search for spatial network databases," in *VLDB*, 2004.

[22] F. Korn and S. Muthukrishnan, "Influence sets based on reverse nearest neighbor queries," in *SIGMOD*, 2000.

[23] F. Korn, S. Muthukrishnan, and D. Srivastava, "Reverse nearest neighbor aggregates over data streams," in *VLDB*, 2002.

[24] K. Mouratidis, M. L. Yiu, D. Papadias, and N. Mamoulis, "Continuous nearest neighbor monitoring in road networks," in *VLDB*, 2006.

[25] N. Roussopoulos, S. Kelley, and F. Vincent, "Nearest neighbor queries," in *SIGMOD*, 1995.

[26] A. Singh, H. Ferhatosmanoglu, and A. Tosun, "High dimensional reverse nearest neighbor queries," in *CIKM*, 2003.

[27] I. Stanoi, M. Riedewald, D. Agrawal, and A. E. Abbadi, "Discovery of influence sets in frequently updated databases," in *VLDB*, 2001.

[28] Y. Tao, D. Papadias, and X. Lian, "Reverse kNN search in arbitrary dimensionality," in *VLDB*, 2004.

[29] Y. Tao, D. Papadias, X. Lian, and X. Xiao, "Multidimensional reverse kNN search," *The VLDB Journal*, vol. 16, no. 3, pp. 293–316, 2007.

[30] Y. Tao, D. Papadias, and Q. Shen, "Continuous nearest neighbor search," in *VLDB*, 2002.

[31] T. Xia and D. Zhang, "Continuous reverse nearest neighbor," in *ICDE*, 2006.

[32] X. Xiong, M. F. Mokbel, and W. G. Aref, "SEA-CNN: scalable processing of continuous k-nearest neighbor queries in spatio-temporal databases," in *ICDE*, 2005.

[33] C. Yang and K.-I. Lin, "An index structure for efficient reverse nearest neighbor queries," in *ICDE*, 2001.

[34] M. L. Yiu and N. Mamoulis, "Reverse nearest neighbors search in ad hoc subspaces," *IEEE Transactions on Knowledge and Data Engineering*, vol. 19, no. 3, pp. 412–426, 2007.

[35] M. L. Yiu, D. Papadias, N. Mamoulis, and Y. Tao, "Reverse nearest neighbors in large graphs," *IEEE Transactions on Knowledge and Data Engineering*, vol. 18, no. 4, pp. 540–553, 2006.

[36] H. Yu, P. K. Agarwal, R. Poreddy, and K. R. Varadarajan, "Practical methods for shape fitting and kinetic data structures using core sets," in *SoCG*, 2004.