VoR-Tree: R-trees with Voronoi Diagrams for Efficient Processing of Spatial Nearest Neighbor Queries*

Mehdi Sharifzadeh †
Google
mehdish@google.com

Cyrus Shahabi
University of Southern California
shahabi@usc.edu

ABSTRACT

A very important class of spatial queries consists of nearest-neighbor (NN) query and its variations. Many studies in the past decade utilize R-trees as their underlying index structures to address NN queries efficiently. The general approach is to use R-tree in two phases. First, R-tree's hierarchical structure is used to quickly arrive to the neighborhood of the result set. Second, the R-tree nodes intersecting with the local neighborhood (*Search Region*) of an initial answer are investigated to find all the members of the result set. While R-trees are very efficient for the first phase, they usually result in the unnecessary investigation of many nodes that none or only a small subset of their including points belongs to the actual result set.

On the other hand, several recent studies showed that the Voronoi diagrams are extremely efficient in exploring an NN search region, while due to lack of an efficient access method, their arrival to this region is slow. In this paper, we propose a new index structure, termed VoR-Tree that incorporates Voronoi diagrams into R-tree, benefiting from the best of both worlds. The coarse granule rectangle nodes of R-tree enable us to get to the search region in logarithmic time while the fine granule polygons of Voronoi diagram allow us to efficiently tile or cover the region and find the result. Utilizing VoR-Tree, we propose efficient algorithms for various Nearest Neighbor queries, and show that our algorithms have better I/O complexity than their best competitors.

1. INTRODUCTION

*This research has been funded in part by NSF grants IIS-0238560 (PECASE), IIS-0534761, and CNS-0831505 (CyberTrust), the NSF Center for Embedded Networked Sensing (CCR-0120778) and in part from the METRANS Transportation Center, under grants from USDOT and Caltrans. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

†The work was completed when the author was studying PhD at USC's InfoLab.

Proceedings of the VLDB Endowment, Vol. 3, No. 1

An important class of queries, especially in the geospatial domain, is the class of nearest neighbor (NN) queries. These queries search for data objects that minimize a distance-based function with reference to one or more query objects. Examples are k Nearest Neighbor (kNN) [13, 5, 7], Reverse k Nearest Neighbor (RkNN) [8, 15, 16], k Aggregate Nearest Neighbor (kANN) [12] and skyline queries [1, 11, 14]. The applications of NN queries are numerous in geospatial decision making, location-based services, and sensor networks.

The introduction of R-trees [3] (and their extensions) for indexing multi-dimensional data marked a new era in developing novel R-tree-based algorithms for various forms of Nearest Neighbor (NN) queries. These algorithms utilize the simple rectangular grouping principle used by R-tree that represents close data points with their Minimum Bounding Rectangle (MBR). They generally use R-tree in two phases. In the first phase, starting from the root node, they iteratively search for an initial result a. To find a, they visit/extract the nodes that minimize a function of the distance(s) between the query points(s) and the MBR of each node. Meanwhile, they use heuristics to prune the nodes that cannot possibly contain the answer. During this phase, R-tree's hierarchical structure enables these algorithms to find the initial result a in logarithmic time.

Next, the local neighborhood of a, *Search Region* (SR) of a for an NN query [5], must be explored further for any possibly better result. The best approach is to visit/examine only the points in SR of a in the direction that most likely contains a better result (e.g., from a towards the query point q for a better NN). However, with R-tree-based algorithms the *only* way to retrieve a point in this neighborhood is through R-tree's leaf nodes. Hence, in the second phase, a *blind* traversal must repeatedly go up the tree to visit higher-level nodes and then come down the tree to visit their descendants and the leaves to explore this neighborhood. This traversal is combined with pruning those nodes that are not intersecting with SR of a and hence contain no point of SR. Here, different algorithms use alternative thresholds and heuristics to decide which R-tree nodes should be investigated further and which ones should be pruned. While the employed heuristics are always safe to cover the entire SR and hence guarantee the completeness of result, they are *highly conservative* for two reasons: 1) They use the distance to the coarse granule MBR of points in a node N as a lower-bound for the actual distances to the points in N. This lower-bound metric is not tight enough for many queries (e.g., RkNN) 2) With some queries (e.g., kANN), the irreg-

ular shape of SR makes it difficult to identify intersecting nodes using a heuristic. As a result, the algorithm examines all the nodes intersecting with a larger *superset* of SR. That is, the conservative heuristics prevent the algorithm to prune many nodes/points that are not even close to the actual result.

A data structure that is extremely efficient in exploring a local neighborhood in a geometric space is Voronoi diagram [10]. Given a set of points, a general Voronoi diagram uniquely partitions the space into disjoint regions. The region (cell) corresponding to a point o covers the points in space that are closer to o than to any other point. The dual representation, Delaunay graph, connects any two points whose corresponding cells share a border (and hence are *close* in a certain *direction*). Thus, to explore the neighborhood of a point a it suffices to start from the Voronoi cell containing a and repeatedly traverse unvisited neighboring Voronoi cells (as if we tile the space using visited cells). The fine granule polygons of Voronoi diagram allows an efficient coverage of any complex-shaped neighborhood. This makes Voronoi diagrams efficient structures to explore the search region during processing NN queries. Moreover, the search region of many NN queries can be redefined as a limited neighborhood through the edges of Delaunay graphs. Consequently, an algorithm can traverse any complex-shaped SR without requiring the MBR-based heuristics of R-tree (e.g, in Section 4 we prove that the reverse kth NN of a point p is at most k edges far from p in the Delaunay graph of data).

In this paper, we propose to incorporate Voronoi diagrams into the R-tree index structure. The resulting data structure, termed *VoR-Tree*, is a regular R-tree enriched by the Voronoi cells and *pointers* to Voronoi neighbors of each point stored together with the point's geometry in its data record. VoR-Tree uses more disk space than a regular R-Tree but instead it highly facilitates NN query processing. VoR-Tree is different from an access method for Voronoi diagrams such as Voronoi history graph [4], os-tree [9], and D-tree [17]. Instead, VoR-Tree benefits from the best of two worlds: coarse granule hierarchical grouping of R-trees and fine granule exploration capability of Voronoi diagrams. Unlike similar approaches that index the Voronoi cells [18, 6] or their approximations in higher dimensions [7], VoR-Tree indexes the actual data objects. Hence, all R-tree-based query processing algorithms are still feasible with VoR-Trees. However, adding the connectivity provided by the Voronoi diagrams enables us to propose I/O-efficient algorithms for different NN queries. Our algorithms use the information provided in VoR-tree to find the query result by performing the least number of I/O operations. That is, at each step they examine only the points inside the current search region. This processing strategy is used by R-tree-based algorithms for I/O-optimal processing of NN queries [5]. While both Voronoi diagrams and R-trees are defined for the space of $\mathbb{R}^d$ which makes *VoR-Tree* applicable for higher dimensions, we focus on 2-d points that are widely available/queried in geospatial applications.

We study three types of NN query and their state-of-the-art R-tree-based algorithms: 1) kNN and Best-First Search (BFS) [5], 2) RkNN and TPL [16], and 3) kANN and MBM [12] (see Appendix F for more queries). For each query, we propose our VoR-Tree-based algorithm, the proof of its correctness and its complexities. Finally, through extensive experiments using three real-world datasest, we evaluate the performance of our algorithms.

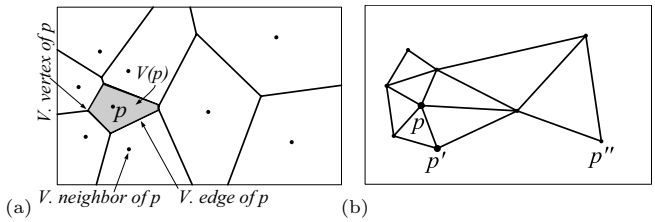

Figure 1: a) Voronoi diagram, b) Delaunay graph

For kNN queries, our incremental algorithm uses an important property of Voronoi diagrams to retrieve/examine only the points neighboring the $(k-1)$-th closest points to the query point. Our experiments verify that our algorithm outperforms BFS [5] in terms of I/O cost (number of accessed disk pages; up to 18% improvement). For RkNN queries, we show that unlike TPL [16], our algorithm is scalable with respect to k and outperforms TPL in terms of I/O cost by at least 3 orders of magnitude. For kANN queries, our algorithm through a diffusive exploration of the irregular-shaped SR prunes many nodes/points examined by the MBM algorithm [12]. It accesses a small fraction of disk pages accessed by MBM (50% decrease in I/O).

2. BACKGROUND

The Voronoi diagram of a given set $P = \{p_1, \ldots, p_n\}$ of n points in $\mathbb{R}^d$ partitions the space of $\mathbb{R}^d$ into n regions. Each region includes all points in $\mathbb{R}^d$ with a common closest point in the given set P according to a distance metric $D(.,.)$ [10]. That is, the region corresponding to the point $p \in P$ contains all the points $q \in \mathbb{R}^d$ for which we have

$$\forall p' \in P, p' \neq p,\ D(q,p) \leq D(q,p') \qquad (1)$$

The equality holds for the points on the borders of p's and p''s regions. Figure 1a shows the *ordinary* Voronoi diagram of nine points in $\mathbb{R}^2$ where the distance metric is Euclidean. We refer to the region $V(p)$ containing the point p as its Voronoi *cell*. With Euclidean distance in $\mathbb{R}^2$, $V(p)$ is a convex polygon. Each edge of this polygon is a segment of the perpendicular bisector of the line segment connecting p to another point of the set P. We call each of these edges a *Voronoi edge* and each of its end-points (vertices of the polygon) a *Voronoi vertex* of the point p. For each Voronoi edge of the point p, we refer to the corresponding point in the set P as a *Voronoi neighbor* of p. We use $VN(p)$ to denote the set of all Voronoi neighbors of p. We also refer to point p as the *generator* of Voronoi cell $V(p)$. Finally, the set given by $VD(P) = \{V(p_1), ..., V(p_n)\}$ is called the Voronoi diagram generated by P with respect to the distance function $D(.,.)$. Throughout this paper, we use Euclidean distance function in $\mathbb{R}^2$. Also, we simply use Voronoi diagram to denote ordinary Voronoi diagram of a set of points in $\mathbb{R}^2$.

Now consider an undirected graph $DG(P) = G(V, E)$ with the set of vertices $V = P$. For each two points p and p' in V, there is an edge connecting p and p' in G iff p' is a Voronoi neighbor of p in the Voronoi diagram of P. The graph G is called the *Delaunay graph* of points in P. This graph is a connected planar graph. Figure 1b illustrates the Delaunay graph corresponding to the points of Figure 1a. In Section 4.2, we traverse the Delaunay graph of the database points to find the set of reverse nearest neighbors of a point.

We review important properties of Voronoi diagrams [10].
Property V-1: The Voronoi diagram of a set P of points,

$VD(P)$, is unique.

Property V-2: Given the Voronoi diagram of P, the nearest point of P to point $p \in P$ is among the Voronoi neighbors of p. That is, the closest point to p is one of generator points whose Voronoi cells share a Voronoi edge with $V(p)$. In Section 4.1, we utilize a generalization of this property in our kNN query processing algorithm.

Property V-3: The average number of vertices per Voronoi cells of the Voronoi diagram of a set of points in $\mathbb{R}^2$ does not exceed six. That is, the average number of Voronoi neighbors of each point of P is at most six. We use this property to derive the complexity of our query processing algorithms.

3. VOR-TREE

In this section, we show how we use an R-tree (see Appendix A for definition of R-Tree) to index the Voronoi diagram of a set of points together with the actual points. We refer to the resulting index structure as *VoR-Tree*, an R-tree of point data augmented with the points' Voronoi diagram.

Suppose that we have stored all the data points of set P in an R-tree. For now, assume that we have pre-built $VD(P)$, the Voronoi diagram of P. Each leaf node of R-tree stores a subset of data points of P. The leaves also include the data records containing extra information about the corresponding points. In the record of the point p, we store the pointer to the location of each Voronoi neighbor of p (i.e., $VN(p)$) and also the vertices of the Voronoi cell of p (i.e., $V(p)$). The above instance of R-tree built using points in P is the *VoR-Tree* of P.

Figure 2a shows the Voronoi diagram of the same points shown in Figure 11. To bound the Voronoi cells with infinite edges (e.g., $V(p_3)$), we clip them using a large rectangle bounding the points in P (the dotted rectangle). Figure 2b illustrates the VoR-Tree of the points of P. For simplicity, it shows only the contents of leaf node N_2 including points p_4, p_5, and p_6, the generators of grey Voronoi cells depicted in Figure 2a. The record associated with each point p in N_2 includes both Voronoi neighbors and vertices of p in a common sequential order. We refer to this record as *Voronoi record* of p. Each Voronoi neighbor p' of p maintained in this record is actually a pointer to the disk page storing p''s information (including its Voronoi record). In Section 4, we use these pointers to navigate within the Voronoi diagram.

In sum, VoR-Tree of P is an R-tree on P blended with Voronoi diagram $VD(P)$ and Delaunay graph $DG(P)$ of P. Trivially, Voronoi neighbors and Voronoi cells of the same point can be computed from each other. However, VoR-Tree stores both of these sets to avoid the computation cost when both are required. For applications focusing on specific queries, only the set required by the corresponding query processing algorithm can be stored.

4. QUERY PROCESSING

In this section, we discuss our algorithms to process different nearest neighbor queries using VoR-Trees. For each query, we first review its state-of-the-art algorithm. Then, we present our algorithm showing how maintaining Voronoi records in VoR-Tree boosts the query processing capabilities of the corresponding R-tree.

4.1 k Nearest Neighbor Query (kNN)

Given a query point q, k Nearest Neighbor (kNN) query finds the k closest data points to q. Given the data set P, it finds k points $p_i \in P$ for which we have $D(q, p_i) \leq$

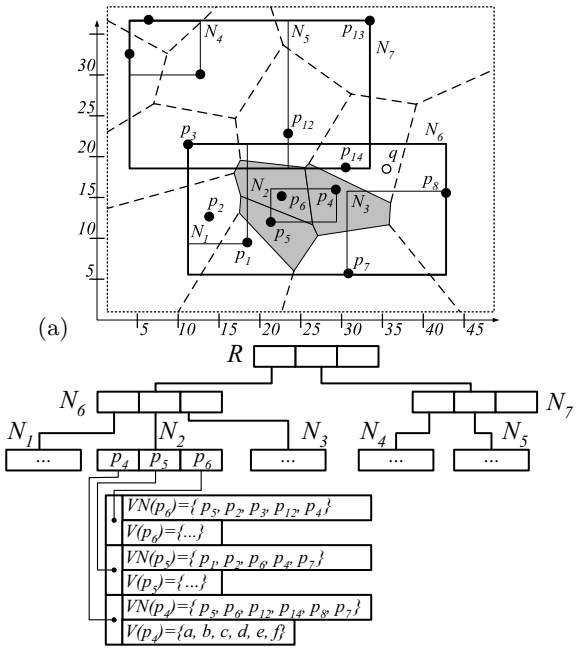

Figure 2: a) Voronoi diagram and b) the VoR-Tree of the points shown in Figure 11

$D(q, p)$ for all points $p \in P \setminus \{p_1, \ldots, p_k\}$ [13]. We use $k\text{NN}(q)=\{p_1, \ldots, p_k\}$ to denote the *ordered* result set; p_i is the i-th NN of q.

The I/O-optimal algorithm for finding kNNs using an R-tree is the *Best-First Search* (BFS) [5]. BFS traverses the nodes of R-tree from the root down to the leaves. It maintains the visited nodes N in a minheap sorted by their $mindist(N, q)$. Consider the R-tree of Figure 2a (VoR-Tree without Voronoi records) and query point q. BFS first visits the root R and adds its entries together with their $mindist()$ values to the heap H ($H=\{(N_6, 0),(N_7, 2)\}$). Then, at each step BFS accesses the node at the top of H and adds all its entries into H. Extracting N_6, we get $H=\{(N_7, 2),(N_3, 3), (N_2, 7),(N_1, 17)\}$. Then, we extract N_7 to get $H=\{(N_5, 2), (N_3, 3),(N_2, 7),(N_1, 17),(N_4, 26)\}$. Next, we extract the point entries of N_5 where we find p_{14} as the first potential NN of q. Now, as $mindist()$ of the first entry of H is less than the distance of the closest point to q we have found so far ($bestdist=D(p_{14}, q)=5$), we must visit N_3 to explore any of its potentially better points. Extracting N_3 we realize that its closest point to q is p_8 with $D(p_8, q)=8>bestdist$ and hence we return p_{14} as the nearest neighbor (NN) of q ($NN(q)=p_{14}$). As an incremental algorithm, we can continue the iterations of BFS to return all k nearest neighbors of q in their ascending order to q. Here, *bestdist* is the distance of the k-th closest point found so far to q .

For a general query $\mathbf{Q}$ on set P, we define the search region (SR) of a point $p \in P$ as the portion of $\mathbb{R}^2$ that may contain a result better than p in P. BFS is considered I/O-optimal as at each iteration it visits only the nodes intersecting the SR of its best candidate result p (i.e., the circle centered at q and with radius equal to $D(q, p)$). However, as the above example shows nodes such as N_3 while intersecting SR of p_{14} might have no point closer than p_{14} to q. We show how one can utilize Voronoi records of VoR-Tree to avoid visiting these nodes.

First, we show our VR-1NN algorithm for processing 1NN

queries (see Figure 13 for the pseudo-code). VR-1NN works similar to BFS. The only difference is that once VR-1NN finds a candidate point p, it accesses the Voronoi record of p. Then, it checks whether the Voronoi cell of p contains the query point q (Line 8 of Figure 13). If the answer is positive, it returns p (and exits) as p is the closest point to q according to the definition of a Voronoi cell. Incorporating this containment check in VR-1NN, avoids visiting (i.e., prunes) node N_3 in the above example as $V(p_{14})$ contains q.

To extend VR-1NN for general kNN processing, we utilize the following property of Voronoi diagrams:

Property V-4: Let $p_1, \ldots, p_k$ be the $k > 1$ nearest points of P to a point q (i.e., p_i is the i-th nearest neighbor of q). Then, p_k is a Voronoi neighbor of at least one point $p_i \in \{p_1, \ldots, p_{k-1}\}$ ($p_k \in VN(p_i)$; see [6] for a proof).

This property states that in Figure 2 where the first NN of q is p_{14}, the second NN of q (p_4) is a Voronoi neighbor of p_{14}. Also, its third NN (p_8) is a Voronoi neighbor of either p_{14} or p_8 (or both as in this example). Therefore, once we find the first NN of q we can easily explore a *limited* neighborhood around its Voronoi cell to find other NNs (e.g., we examine only Voronoi neighbors of $NN(q)$ to find the second NN of q). Figure 14 shows the pseudo-code of our VR-kNN algorithm. It first uses VR-1NN to find the first NN of q (p_{14} in Figure 2a). Then, it adds this point to a minheap H sorted on the ascending distance of each point entry to q (H=$(p_{14}, 5)$). Subsequently, each following iteration removes the first entry from H, returns it as the next NN of q and adds all its Voronoi neighbors to H. Assuming $k = 3$ in the above example, the trace of VR-kNN iterations is:
1) p_{14} = 1st NN, add $VN(p_{14}) \Rightarrow H$=$((p_4, 7), (p_8, 8), (p_{12}, 13), (p_{13}, 18))$.
2) p_4 = 2nd NN, add $VN(p_4) \Rightarrow H$=$((p_8, 8), (p_{12}, 13), (p_6, 13), (p_7, 14), (p_5, 16), (p_{13}, 18))$. 3) p_8 = 3rd NN, terminate.

Correctness: The correctness of VR-kNN follows the correctness of BFS and the definition of Voronoi diagrams.

Complexity: We compute I/O complexities in terms of Voronoi records and R-tree nodes retrieved by the algorithm. VR-kNN once finds NN(q) executes exactly k iterations each extracting Voronoi neighbors of one point. Property V-3 states that the average number of these neighbors is constant. Hence, the I/O complexity of VR-kNN is $O(\Phi(|P|) + k)$ where $\Phi(|P|)$ is the complexity of finding 1st NN of q using VoR-Tree (or R-tree). The time complexity can be determined similarly.

Improvement over BFS: We show how, for the same kNN query, VR-kNN accesses less number of disk pages (or VoR-Tree nodes) comparing to BFS. Figure 3a shows a query point q and 3 nodes of a VoR-Tree with 8 entries per node. With the corresponding R-tree, BFS accesses node N_1 where it finds p_1, the first NN of q. To find q's 2nd NN, BFS visits both nodes N_2 and N_3 as their *mindist* is less than $D(p_2, q)$ (p_2 is the closest 2nd NN found in N_1). However, VR-kNN does not access N_2 and N_3. It looks for 2nd NN in Voronoi neighbors of p_1 which are all stored in N_1. Even when it returns p_2 as 2nd NN, it looks for 3rd NN in the same node as N_1 contains all Voronoi neighbors of both p_1 and p_2. The above example represents a sample of many kNN query scenarios where VR-kNN achieves a better I/O performance than BFS.

4.2 Reverse k Nearest Neighbor Query (RkNN)

Given a query point q, Reverse k Nearest Neighbor (RkNN) query retrieves all the data points $p \in P$ that have q as one of their k nearest neighbors. Given the data set P, it finds all $p \in P$ for which we have $D(q, p) \leq D(q, p_k)$ where p_k is the k-th nearest neighbor of p in P [16]. We use RkNN(q) to denote the result set. Figure 3b shows point p together with p_1 and p_2 as p's 1st and 2nd NNs, respectively. The point p is closer to p_1 than to q and hence p is not in R1NN(q). However, q is inside the circle centered at p with radius $D(p, p_2)$ and hence it is closer to p than p_2 to p. As p_2 is the 2nd NN of p, p is in R2NN(q).

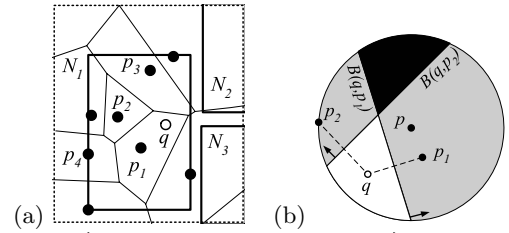

Figure 3: a) Improving over BFS, b) $p \in$ R2NN(q)

The TPL algorithm for RkNN search proposed by Tao et al. in [16] uses a two-step filter-refinement approach on an R-tree of points. TPL first finds a set of candidate RNN points S_{cnd} by a single traversal of the R-tree, visiting its nodes in ascending distance from q and performing smart pruning. The pruned nodes/points are kept in a set S_{rfn} which are used during the refinement step to eliminate false positives from S_{cnd}. We review TPL starting with its filtering criteria to prune the nodes/points that cannot be in the result. In Figure 3b, consider the perpendicular bisector $B(q, p_1)$ which divides the space into two half-planes. Any point such as p locating on the same half-plane as p_1 (denoted as $B_{p_1}(q, p_1)$) is closer to p_1 than to q. Hence, p cannot be in R1NN(q). That is, any point p in the half-plane $B_{p_1}(q, p_1)$ defined by the bisector of qp_1 for an arbitrary point p_1 cannot be in R1NN(q). With R1NN queries, TPL uses this criteria to prune the points that are in $B_{p_1}(q, p_1)$ of another point p_1. It also prunes the nodes N that are in the union of $B_{p_i}(q, p_i)$ for a set of candidate points p_i. The reason is that each point in N is closer to one of p_i's than to q and hence cannot be in R1NN(q). A similar reasoning holds for general RkNN queries. Considering $B_{p_1}(q, p_1)$ and $B_{p_2}(q, p_2)$, p is not inside the intersection of these two half-planes (the region in black in Figure 3b). It is also outside the intersection of the corresponding half-planes of any two arbitrary points of P. Hence, it is not closer to any two points than to q and therefore p is in R2NN(q).

With R1NN query, pruning a node N using the above criteria means incrementally clipping N by bisector lines of n candidate points into a convex polygon N_{res} which takes $O(n^2)$ times. The *residual region* N_{res} is the part of N that may contain candidate RNNs of q. If the computed N_{res} is empty, then it is safe to prune N (and add it to S_{rfn}). This filtering method is more complex with RkNN queries where it must clip N with each of $\binom{n}{k}$ combinations of bisectors of n candidate points. To overcome this complexity, TPL uses a conservative *trim* function which guarantees that no possible RNN is pruned. With R1NN, *trim* incrementally clips the MBR of the clipped N_{res} from the previous step. With RkNN, as clipping with $\binom{n}{k}$ combinations, each with k bisector lines, is prohibitive, *trim* utilizes heuristics and approximations. TPL's filter step is applied in rounds. Each round first eliminates candidates of S_{cnd} which are pruned by at least k entries in S_{rfn}. Then, it adds to the final result RkNN(q) those candidates which are guaranteed not to be

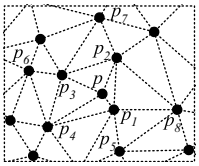

Figure 4: Lemma 1

pruned by any entry of S_{cnd}. Finally, it queues more nodes from S_{rfn} to be accessed in the next round as they might prune some candidate points. The iteration on refinement rounds is terminated when no candidate left (S_{cnd}=∅).

While TPL utilizes smart pruning techniques, there are two drawbacks: 1) For k>1, the conservative filtering of nodes in the trim function fails to prune the nodes that can be discarded. This results into increasing the number of candidate points [16]. 2) For many query scenarios, the number of entries kept in S_{rfn} is much higher than the number of candidate points which increases the workspace required for TPL. It also delays the termination of TPL as more refinement rounds must be performed.

Improvement over TPL: Similar to TPL, our VR-RkNN algorithm also utilizes a filter-refinement approach. However, utilizing properties of Voronoi diagram of P, it eliminates the exhaustive refinement rounds of TPL. It uses the Voronoi records of VoR-Tree of P to examine only a limited neighborhood around a query point to find its RNNs. First, the filter step utilizes two important properties of RNNs to define this neighborhood from which it extracts a set of candidate points and a set of points required to prune false hits (see Lemmas 2 and 3 below). Then, the refinement step finds kNNs of each candidate in the latter set and eliminates those candidates that are closer to their k-th NN than to q.

We discuss the properties used by the filter step. Consider the Delaunay graph $DG(P)$. We define $gd(p, p')$ the *graph distance* between two vertices p and p' of $DG(P)$ (points of P) as the minimum number of edges connecting p and p' in $DG(P)$. For example, in Figure 1b we have $gd(p, p')$=1 and $gd(p, p'')$=2.

LEMMA 1. *Let $p_k \neq p$ be the k-th closest point of set P to a point $p \in P$. The upper bound of the graph distance between vertices p and p_k in Delaunay graph of P is k (i.e. $gd(p, p_k) \leq k$).*

PROOF. The proof is by induction. Consider the point p in Figure 4. First, for k=1, we show that $gd(p, p_1)\leq 1$. Property V-4 of Section 4.1 states that p_1 is a Voronoi neighbor of p; p_1 is an immediate neighbor of p in Delaunay graph of P and hence we have $gd(p, p_1)$=1. Now, assuming $gd(p, p_i)\leq i$ for $0\leq i\leq k$, we show that $gd(p, p_{k+1})\leq k+1$. Property V-4 states that p_{k+1} is a Voronoi neighbor of at least one $p_i \in \{p_1, \ldots, p_k\}$. Therefore, we have $gd(p, p_{k+1}) \leq max(gd(p, p_i)) + 1 \leq k+1$. □

In Figure 5, consider the query point q and the Voronoi diagram $VD(P\cup\{q\})$ (q added to $VD(P)$). Lemma 1 states that if q is one of the kNNs of a point p, then we have $gd(p, q)\leq k$; p is not farther than k distance from q in Delaunay graph of $P\cup\{q\}$. This yields the following lemma:

LEMMA 2. *If p is one of reverse kNNs of q, then we have $gd(p, q) \leq k$ in Delaunay graph $DG(P\cup\{q\})$.*

As the first property of RNN utilized by our filter step, Lemma 2 limits the local neighborhood around q that may

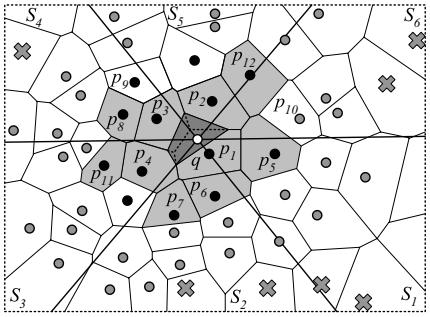

Figure 5: VR-RkNN for $k = 2$

contain q's RkNNs. In Figure 5, the non-black points cannot be R2NNs of q as they are farther than k=2 from q in $DG(P\cup\{q\})$. We must only examine the black points as candidate R2NNs of q. However, the number of points in k graph distance from q grows exponentially with k. Therefore, to further limit these candidates, the filter step also utilizes another property first proved in [15] for R1NNs and then generalized for RkNNs in [16]. In Figure 5, consider the 6 equi-sized partitions $S_1, \ldots, S_6$ defined by 6 vectors originating from q.

LEMMA 3. *Given a query point q in $\mathbb{R}^2$, the kNNs of q in each partition defined as in Figure 5 are the only possible RkNNs of q (see [16] for a proof).*

The filter step adds to its candidate set only those points that are closer than $k+1$ from q in $DG(P\cup\{q\})$ (Lemma 2). From all candidate points inside each partition S_i (defined as in Figure 5), it keeps only the k closest ones to p and discards the rest (Lemma 3). Notice that both filters are required. In Figure 5, the black point p_9 while in distance 2 from q is eliminated during our search for R2NN(q) as it is the 3rd closest point to q in partition S_4. Similarly, p_{10}, the closest point to q in S_6, is eliminated as $gd(q, p_{10})$ is 3.

To verify each candidate p, in refinement step we must examine whether p is closer to q than p's k-th NN (i.e., p_k). Lemma 1 states that the upper bound of $gd(p, p_k)$ is k ($gd(p, p_k)\leq k$). Candidate p can also be k edges far from q ($gd(p, q)\leq k$). Hence, p_k can be in distance $2k$ from q (all black and grey points in Figure 5). All other points (shown as grey crosses) are not required to filter the candidates. Thus, it suffices to visit/keep this set R and compare them to q with respect to the distance to each candidate p. However, visiting the points in R through $VD(P)$ takes exponential time as the size of R grows exponentially with k. To overcome this exponential behavior, VR-RkNN finds the k-th NN of each candidate p which takes only $O(k^2)$ time for at most $6k$ candidates.

Figure 15 shows the pseudo-code of the VR-RkNN algorithm. VR-RkNN maintains 6 sets $S_{cnd}(i)$ including candidate points of each partition (Line 2). Each set $S_{cnd}(i)$ is a minheap storing the (at most) k NNs of q inside partition S_i. First, VR-RkNN finds the Voronoi neighbors of q as if we add q into $VD(P)$ (Line 3). This is easily done using the insert operation of VoR-Tree without actual insertion of q (see Appendix B).

In the filter step, VR-RkNN uses a minheap H sorted on the graph distance of its point entries to q to traverse $VD(P)$ in ascending $gd()$ from q. It first adds all neighbors p_i of q to H with $gd(q, p_i)$=1 (Lines 4-5). In Figure 5, the points $p_1, \ldots, p_4$ are added to H. Then, VR-RkNN iterates over the top entry of H. At each iteration, it removes the top entry p.

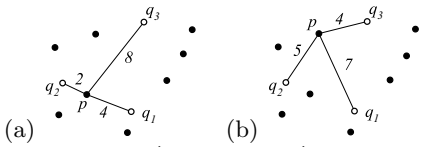

Figure 6: a) f=*sum* , b) f=*max*

If p, inside S_i, passes both filters defined by Lemma 2 and 3, the algorithm adds $(p, D(q,p))$ to the candidate set of partition S_i ($S_{cnd}(i)$; e.g., p_1 to S_1). It also accesses the Voronoi record of p through which it adds the Voronoi neighbors of p to H (incrementing its graph distance; Lines 12-15). The filter step terminates when H becomes empty. In our example, the first iteration adds p_1 to $S_{cnd}(1)$, and p_5, p_6 and p_7 with distance 2 to H. After the last iteration, we have $S_{cnd}(1) = \{p_1, p_5\}$, $S_{cnd}(2) = \{p_6, p_7\}$, $S_{cnd}(3) = \{p_4, p_{11}\}$, $S_{cnd}(4) = \{p_3, p_8\}$, $S_{cnd}(5) = \{p_2, p_{12}\}$, and $S_{cnd}(6) = \{\}$.

The refinement step (Line 17) examines the points in each $S_{cnd}(i)$ and adds them to the final result iff they are closer to their k-th NN than to q (R2NN=$\{p_1, p_2\}$). Finding the k-th NN is straightforward using an approach similar to VR-kNN of Section 4.1.

4.3 k Aggregate Nearest Neighbor Query (kANN)

Given the set Q=$\{q_1, \ldots, q_n\}$ of query points, k Aggregate Nearest Neighbor Query (kANN) finds the k data points in P with smallest *aggregate* distance to Q. We use kANN(q) to denote the result set. The aggregate distance $adist(p, Q)$ is defined as $f(D(p, q_1), \ldots, D(p, q_n))$ where f is a monotonically increasing function [12]. For example, considering P as meeting locations and Q as attendees' locations, 1ANN query with f=*sum* finds the meeting location traveling towards which minimizes the total travel distance of all attendees (p minimizes f=*sum* in Figure 6a). With f=*max*, it finds the location that leads to the the earliest time that *all* attendees arrive (assuming equal fixed speed; Figure 6b). Positive weights can also be assigned to query points (e.g., $adist(p, Q)$=$\sum_{i=1}^{n} w_i D(p, q_i)$ where $w_i \geq 0$). Throughout this section, we use functions f and $adist()$ interchangeably.

The best R-tree-based solution for kANN queries is the MBM algorithm [12]. Similar to BFS for kNN queries, MBM visits only the nodes of R-tree that may contain a result better than the best one found so far. Based on two heuristics, it utilizes two corresponding functions that return lower-bounds on the $adist()$ of any point in a node N to prune N: 1) $amindist(N, M)$=$f(nm, \ldots, nm)$ where nm=$mindist(N, M)$ is the minimum distance between the two rectangles N and M, the minimum bounding box of Q, and 2) $amindist(N, Q) = f(nq_1, \ldots, nq_n)$ where nq_i= $mindist(N, q_i)$. For each node N, MBM first examines if $amindist(N, M)$ is larger than the aggregate distance of the current best result p ($bestdist$= $adist(p, Q)$). If the answer is positive, MBM discards N. Otherwise, it examines if the second lower-bound $amindist(N, Q)$ is larger than $bestdist$. If yes, it discards N. Otherwise, MBM visits N's children. Once MBM finds a data point it updates its current best result and terminates when no better point can be found.

We show that MBM's conservative heuristics which are based on the rectangular grouping of points into nodes do not properly suit the shape of kANN's search region SR (the portion of space that *may* contain a better result). Hence, they fail to prune many nodes. Figures 7a and 7b illustrate SRs of a point p for kANN queries with aggregate functions f=*sum* and f=*max*, respectively (regions in grey). The

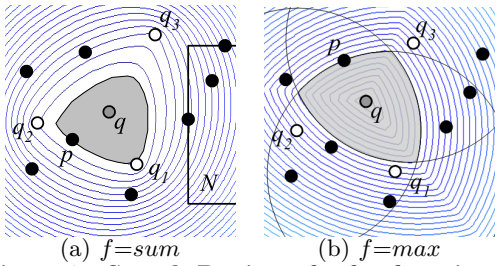

Figure 7: Search Region of p for function f

point p' is in SR of p iff we have $adist(p', Q) \leq adist(p, Q)$. The equality holds on SR's boundary (denoted as SRB). For f=*sum* (and weighted *sum*), SR has an irregular circular shape that depends on the query cardinality and distribution (an ellipse for 2 query points) [12]. For f=*max*, SR is the intersection of n circles centered at q_i's with *radius*= $max(D(p, q_i))$. The figure shows SRBs of several points as contour lines defined as the locus of points $p \in \mathbb{R}^2$ where $adist(p, Q)$=c (constant). As shown, SR of p' is completely inside SR of p iff we have $adist(p', Q) < adist(p, Q)$. The figure also shows the *centroid* q of Q defined as the point in $\mathbb{R}^2$ that minimizes $adist(q, Q)$. Notice that q is inside SRs of all points.

MBM once finds p as a candidate answer to kANN, *tries* to examine only the nodes N that intersect with SR of p. However, the conservative lower-bound function $amindist()$ causes MBM not to prune nodes such as N in Figure 7a that fall completely outside SR of p ($amindist(N, Q)$=3+8+2=13 $< adist(p, Q)$=14).

Now, we describe our VR-kANN algorithm for kANN queries using the example shown in Figure 8 (see Figure 16 for pseudo-code). VR-kANN uses VoR-Tree to traverse the Voronoi cells covering the search space of kANN. It starts with the Voronoi cell that contains the centroid q of Q (Line 1). The generator of this cell is the closest point of P to q (p_q=p_2 in Figure 8). VR-kANN uses a minheap H of points p sorted on the $amindist(V(p), Q)$ which is the minimum possible $adist$ of any point in Voronoi cell of p (Line 2). The function plays the same role as the two lower-bound functions used in heuristics of MBM. It sets a lower-bound on $adist()$ of the points in a Voronoi cell. Later, we show how we efficiently compute this function. VR-kANN also keeps the k points with minimum $adist()$ in a result heap RH with their $adist()$ as keys (Line 3). With the example of Figure 8, VR-kANN first adds $(p_2, 0)$ and $(p_2, 40)$ into H and RH, respectively. At each iteration, VR-kANN first deheaps the first entry p from H (Line 6). It reports any point p_i in RH whose $adist()$ is less than $amindist(V(p), Q)$. The reason is that the SR of p_i has been already traversed by the points inserted in H and no result better than p_i can be found (see Lemma 5). Finally, it inserts all non-visited Voronoi neighbors of p into H and RH (p_1, p_3, p_4, p_6, and p_7; p_1 and p_2 are the best 1st and 2nd ANN in RH; Lines 11-15). The algorithm stops when it reports the k-th point of RH. In our example VR-kANN subsequently visits the neighbors of p_1, p_3, p_4, p_5, p_6, and p_8 where it reports p_1 and p_2 as the first two ANNs. It reports p_3 and stops when it deheaps p_{10} and p_7.

Appendix E discusses the functions FindCentroidNN() used to find the closest data point p_q to centroid q and $amindist(V(p), Q)$ that finds the minimum possible $adist()$ for the points in $V(p)$.

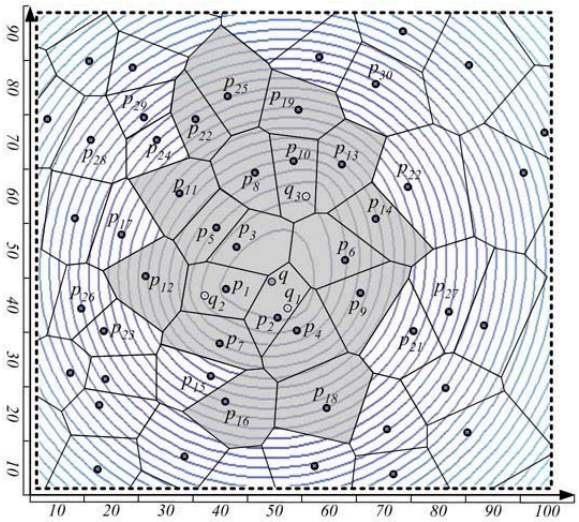

Figure 8: VR-kANN for $k = 3$

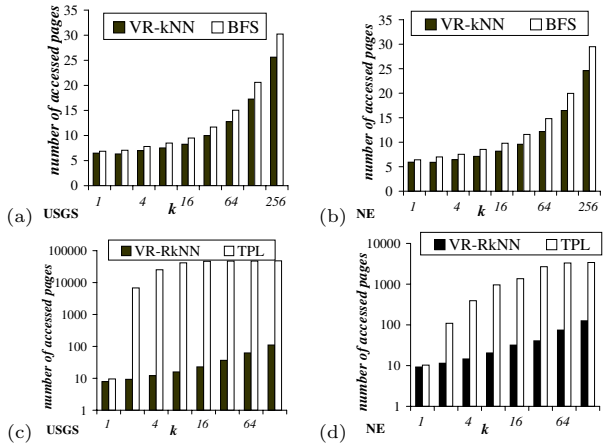

Figure 9: I/O vs. k for ab) kNN, and cd) RkNN

5. PERFORMANCE EVALUATION

We conducted several experiments to evaluate the performance of query processing using VoR-Tree. For each of four NN queries, we compared our algorithm with the competitor approach, with respect to the average number of disk I/O (page accesses incurred by the underlying R-tree/VoR-Tree). For R-tree-based algorithms, this is the number of accessed R-tree nodes. For VoR-Tree-based algorithms, the number of disk pages accessed to retrieve Voronoi records is also counted. Here, we do not report CPU costs as all algorithms are mostly I/O-bound. We investigated the effect of the following parameters on performance: 1) number of NNs k for kNN, kANN, and RkNN queries, 2) number of query points ($|Q|$) and the size of MBR of Q for kANN, and 3) cardinality of the dataset for all queries.

We used three real-world datasets indexed by both R*-tree and VoR-Tree (same *page size*=1K bytes, *node capacity*=30). **USGS** dataset, obtained from the U.S. Geological Survey (USGS), consists of 950,000 locations of different businesses in the entire U.S.. **NE** dataset contains 123,593 locations in New York, Philadelphia and Boston [1]. **GRC** dataset includes the locations of 5,922 cities and villages in Greece (we omit the experimental result of this dataset because of similar behavior and space limitation). The experiments were performed by issuing 1000 *random* instances of each query type on a DELL Precision 470 with Xeon 3.2 GHz processor and 3GB of RAM (*buffer size*=100K bytes). For convex hull computation in VR-S^2, we used the Graham scan algorithm.

In the first set of experiments, we measured the average number of disk pages accessed (I/O cost) by VR-kNN and BFS algorithms varying values of k. Figure 9a illustrates the I/O cost of both algorithms using **USGS**. As the figure shows, utilizing Voronoi cells in VR-kNN enables the algorithm to prune nodes that are accessed by BFS. Hence, VR-kNN accesses less number of pages comparing to BFS, especially for larger values of k. With k=128, VR-kNN discards almost 17% of the nodes which BFS finds intersecting with SR. This improvement over BFS is increasing when k increases. The reason is that the radius of SR used by BFS's pruning is first initialized to $D(q, p)$ where p is the k-th visited point. This distance increases when k increases and causes many nodes intersect with SR and hence not be pruned by BFS. VR-kNN however uses Property V-4 to define a tighter SR. We also realized that this difference in I/O costs increases if we use smaller node capacities for the utilized R-tree/VoR-Tree. Figure 9b shows a similar observation for the result of **NE**.

The second set of experiments evaluates the I/O cost of VR-RkNN and TPL for RkNN queries. Figures 9c and 9d depict the I/O costs of both algorithms for different values of k using **USGS** and **NE**, respectively (the scale of y-axis is logarithmic). As shown, VR-RkNN significantly outperforms TPL by at least 3 orders of magnitude, especially for k>1 (to find R4NN with **USGS**, TPL takes 8 seconds (on average) while VR-RkNN takes only 4 milliseconds). TPL's filter step fails to prune many nodes as *k_trim* function is highly conservative. It uses a conservative approximation of the intersection between a node and SR. Moreover, to avoid exhaustive examinations it prunes using only n combinations of $\binom{n}{k}$ combinations of n candidate points. Also, TPL keeps many pruned (non-candidate) nodes/points for further use in its refinement step. VR-RkNN's I/O cost is determined by the number of Voronoi/Delaunay edges traversed from q and the distance $D(q, p_k)$ between q and p_k=k-th closest point to q in each one of 6 directions. Unlike TPL, VR-RkNN does not require to keep any non-candidate node/point. Instead, it performs single traversals around its candidate points to refine its results. VR-RkNN's I/O cost increases very slowly when k increases. The reason is that $D(q, p_q)$ (and hence the size of SR utilized by VR-RkNN) increases very slowly with k. TPL's performance is variable for different data cardinalities. Hence, our result is different from the corresponding result shown in [16] as we use different datasets with different R-tree parameters.

Our next set of experiments studies the I/O costs of VR-kANN and MBM for kANN queries. We used f=*sum* and $|Q|$=8 query points all inside an MBR covering 4% of the entire dataset and varied k. Figures 10a and 10b show the average number of disk pages accessed by both algorithms using **USGS** and **NE**, respectively. Similar to previous results, VR-kANN is the superior approach. Its I/O cost is almost half of that of MBM when $k \leq 16$ with **USGS** ($k \leq 128$ with **NE** dataset). This verifies that VR-kANN's traversal of SR from the centroid point effectively covers the circular irregular shape of SR of *sum* (see Figure 7a). That is,

[1] http://geonames.usgs.gov/, http://www.rtreeportal.org/

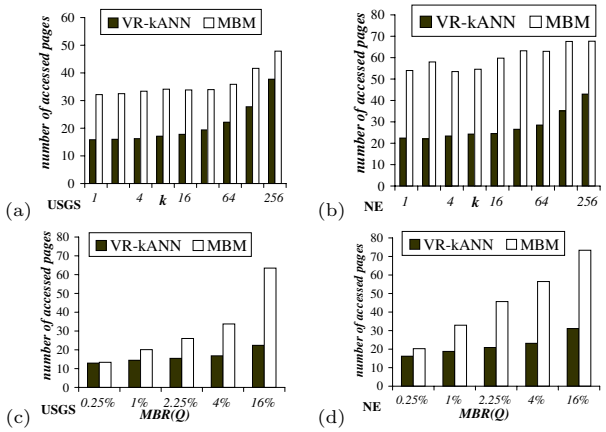

Figure 10: I/O vs. ab) k, and cd) $MBR(Q)$ for kANN

the traversal does not continue beyond a limited neighborhood enclosing SR. However, MBM's conservative heuristic explores the nodes intersecting a wide margin around SR (a superset of SR). Increasing k decreases the difference between the performance of VR-kANN and that of MBM. The intuition here is that with large k, SR converges to a circle around the centroid point q (see outer contours in Figure 7a). That is, SR becomes equivalent to the SR of kNN with query point q. Hence, the I/O costs of VR-kANN and MBM converge to that of their corresponding algorithms for kNN queries with the same value for k.

The last set of experiments investigates the impact of closeness of the query points on the performance of each kANN algorithm. We varied the area covered by the $MBR(Q)$ from 0.25% to 16% of the entire dataset. With f=*sum* and $|Q|$=k=8, we measured the I/O cost of VR-kANN and MBM. As Figures 10c and 10d show, when the area covered by query points increases, VR-kANN accesses much less disk pages comparing to MBM. The reason is a faster increase in MBM's I/O cost. When the query points are distributed in a larger area, the SR is also proportionally large. Hence, the larger wide margin around SR intersects with much more nodes leaving them unpruned by MBM. We also observed that changing the number of query points in the same MBR does not change the I/O cost of MBM. This observation matches the result reported in [12]. Similarly, VR-kANN's I/O cost is the same for different query sizes. The reason is that VR-kANN's I/O is affected only by the size of SR. Increasing the number of query points in the same MBR only changes the shape of SR not its size.

The extra space required to store Voronoi records of data points in VoR-Trees increases the overall disk space comparing to the corresponding R-Tree index structure for the same dataset. The Vor-Trees (R*-trees) of **USGS** and **NE** datasets are 160MB (38MB) and 23MB (4MB), respectively. That is, a Vor-Tree needs at least 5 times more disk space than the space needed to index the same dataset using an R*-Tree. Considering the low cost of storage devices and high demand for prompt query processing, this space overhead is acceptable to modern applications of today.

6. CONCLUSIONS

We introduced VoR-Tree, an index structure that incorporates Voronoi diagram and Delaunay graph of a set of data points into an R-tree that indexes their geometries. VoR-Tree benefits from both the neighborhood exploration capability of Voronoi diagrams and the hierarchical structure of R-trees. For various NN queries, we proposed I/O-efficient algorithms utilizing VoR-Trees. All our algorithms utilize the hierarchy of VoR-Tree to access the portion of data space that contains the query result. Subsequently, they use the Voronoi information associated with the points at the leaves of VoR-Tree to traverse the space towards the actual result. Founded on geometric properties of Voronoi diagrams, our algorithms also redefine the search region of NN queries to expedite this traversal.

Our theoretical analysis and extensive experiments with real-world datasets prove that VoR-Trees enable I/O-efficient processing of kNN, Reverse kNN, k Aggregate NN, and spatial skyline queries on point data. Comparing to the competitor R-tree-based algorithms, our VoR-Tree-based algorithms exhibit performance improvements up to 18% for kNN, 99.9% for RkNN, and 64% for kANN queries.

7. REFERENCES

[1] S. Börzsönyi, D. Kossmann, and K. Stocker. The Skyline Operator. In *ICDE'01*, pages 421–430, 2001.

[2] J. V. den Bercken, B. Seeger, and P. Widmayer. A Generic Approach to Bulk Loading Multidimensional Index Structures. In *VLDB'97*, 1997.

[3] A. Guttman. R-trees: a Dynamic Index Structure for Spatial Searching. In *ACM SIGMOD '84*, pages 47–57, USA, 1984. ACM Press.

[4] M. Hagedoorn. Nearest Neighbors can be Found Efficiently if the Dimension is Small relative to the input size. In *ICDT'03*, volume 2572 of *Lecture Notes in Computer Science*, pages 440–454. Springer, January 2003.

[5] G. R. Hjaltason and H. Samet. Distance Browsing in Spatial Databases. *ACM TODS*, 24(2):265–318, 1999.

[6] M. Kolahdouzan and C. Shahabi. Voronoi-Based K Nearest Neighbor Search for Spatial Network Databases. In *VLDB'04*, pages 840–851, Toronto, Canada, 2004.

[7] S. Berchtold, B. Ertl, D. A. Keim, H. Kriegel, and T. Seidl. Fast Nearest Neighbor Search in High-Dimensional Space. In *ICDE'98*, pages 209–218, 1998.

[8] F. Korn and S. Muthukrishnan. Influence Sets based on Reverse Nearest Neighbor Queries. In *ACM SIGMOD'02*, pages 201–212. ACM Press, 2000.

[9] S. Maneewongvatana. *Multi-dimensional Nearest Neighbor Searching with Low-dimensional Data*. PhD thesis, Computer Science Department, University of Maryland, College Park, MD, 2001.

[10] A. Okabe, B. Boots, K. Sugihara, and S. N. Chiu. *Spatial Tessellations, Concepts and Applications of Voronoi Diagrams*. John Wiley and Sons Ltd., 2nd edition, 2000.

[11] D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive Skyline Computation in Database Systems. *ACM TODS*, 30(1):41–82, 2005.

[12] D. Papadias, Y. Tao, K. Mouratidis, and C. K. Hui. Aggregate Nearest Neighbor Queries in Spatial Databases. *ACM TODS*, 30(2):529–576, 2005.

[13] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest Neighbor Queries. In *SIGMOD'95*, pages 71–79, USA, 1995.

[14] M. Sharifzadeh and C. Shahabi. The Spatial Skyline Queries. In *VLDB'06*, September 2006.

[15] I. Stanoi, D. Agrawal, and A. E. Abbadi. Reverse Nearest Neighbor Queries for Dynamic Databases. In *ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, pages 44–53, 2000.

[16] Y. Tao, D. Papadias, and X. Lian. Reverse kNN Search in Arbitrary Dimensionality. In *VLDB'04*, pages 744–755, 2004.

[17] J. Xu, B. Zheng, W.-C. Lee, and D. L. Lee. The D-Tree: An Index Structure for Planar Point Queries in Location Based Wireless Services. *IEEE TKDE*, 16(12):1526–1542, 2004.

[18] B. Zheng and D. L. Lee. Semantic Caching in Location -dependent Query Processing. In *SSTD'01*, pages 97–116.

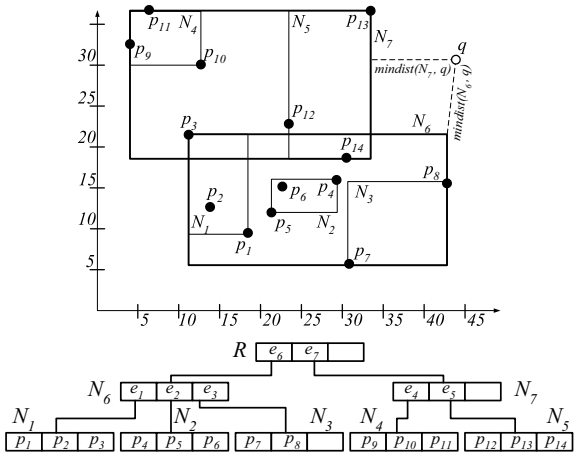

Figure 11: Points indexed by an R-tree

APPENDIX

A. R-TREES

R-tree [3] is the most prominent index structure widely used for spatial query processing. R-trees group the data points in $\mathbb{R}^d$ using d-dimensional rectangles, based on the closeness of the points. Figure 11 shows the R-tree built using the set $P = \{p_1, \ldots, p_{13}\}$ of points in $\mathbb{R}^2$. Here, the capacity of each node is three entries. The leaf nodes $N_1, \ldots, N_5$ store the coordinates of the grouped points together with optional pointers to their corresponding records. Each intermediate node (e.g., N_6) contains the Minimum Bounding Rectangle (MBR) of each of its child nodes (e.g., e_1 for node N_1) and a pointer to the disk page storing the child. The same grouping criteria is used to group intermediate nodes into upper level nodes. Therefore, the MBRs stored in the single root of R-tree collectively cover the entire data set P. In Figure 11, the root node R contains MBRs e_6 and e_7 enclosing the points in nodes N_6 and N_7, respectively.

R-tree-based algorithms utilize some metrics to bound their search space using the MBRs stored in the nodes. The widely used function is $mindist(N, q)$ which returns the minimum possible distance between a point q and any point in the MBR of node N. Figure 11 shows $mindist(N_6, q)$ and $mindist(N_7, q)$ for q. In Section 4, we show how R-tree-based approaches use this lower-bound function.

B. VOR-TREE MAINTENANCE

Given the set of points P, the batch operation to build the VoR-Tree of P first uses a classic approach such as Fortune's sweepline algorithm [10] to build the Voronoi diagram of P. The Voronoi neighbors and the vertices of the cell of each point is then stored in its record. Finally, it easily uses a bulk construction approach for R-trees [2] to index the points in P considering their Voronoi records. The resulted R-tree is the VoR-Tree of P.

To *insert* a new point x in a VoR-Tree, we first use VoR-Tree (or the corresponding R-tree) to locate p, the closest point to x in the set P. This is the point whose Voronoi cell includes x. Then, we insert x in the corresponding R-tree. Finally, we need to build/store Voronoi cell/neighbors of x and subsequently update those of its neighbors in their Voronoi records. We use the algorithm for incremental building of Voronoi diagrams presented in [10]. Figure 12 shows this scenario. Inserting the point x residing in the Voronoi cell of p_1, changes the Voronoi cells/neighbors (and hence the Voronoi records) of points p_1, p_2, p_3 and p_4. We first clip $V(p_1)$ using the perpendicular bisector of line segment xp_1 (i.e., line $B(x, p_1)$) and store the new cell in p_1's record. We also update Voronoi neighbors of p_1 to include the new point x. Then, we select the Voronoi neighbor of p_1 corresponding to one of (possibly two) Voronoi edges of p_1 that intersect with $B(x, p_1)$ (e.g., p_2). We apply the same process using the bisector line $B(x, p_2)$ to clip and update $V(p_2)$. Subsequently, we add x to the Voronoi neighbors of p_2. Similarly, we iteratively apply the process to p_3 and p_4 until p_1 is selected again. At this point the algorithm terminates and as the result it updates the Voronoi records of points p_i and computes $V(x)$ as the regions removed from the clipped cells. The Voronoi neighbors of x are also set to the set of generator points of updated Voronoi cells. Finally, we store the updated Voronoi cells and neighbors in the Voronoi records corresponding to the affected points. Notice that finding the affected points $p_1, \ldots, p_4$ is straightforward using Voronoi neighbors and the geometry of Voronoi cells stored in VoR-Tree.

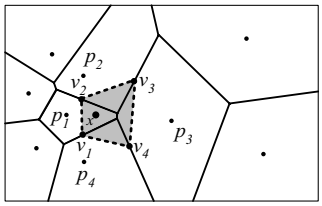

Figure 12: Inserting the point x in VoR-Tree

Algorithm VR-1NN (point q)
01. minheap $H = \{(R, 0)\}$; $bestdist = \infty$;
02. WHILE H is not empty DO
03. remove the first entry e from H;
04. IF e is a leaf node THEN
05. FOR each point p of e DO
06. IF $D(p, q) < bestdist$ THEN
07. $bestNN = p$; $bestdist = D(p, q)$;
08. IF $V(bestNN)$ contains q THEN RETURN $bestNN$;
09. ELSE // *e is an intermediate node*
10. FOR each child node e' of e DO
11. insert $(e', mindist(e', q))$ into H;

Figure 13: 1NN algorithm using VoR-Tree

To *delete* a point x from VoR-Tree, we first locate the Voronoi record of x using the corresponding R-tree. Then, we access its Voronoi neighbors through this record. The cells and neighbors of these points must be updated after deletion of x. To perform this update, we use the algorithm in [10]. It simply uses the intersections of perpendicular bisectors of each pair of neighbors of x to update their Voronoi cells. We also remove x from the records of its neighbors and add any possible new neighbor to these records. At this point, it is safe to delete x from the corresponding R-tree.

The *update* operation of VoR-Tree to change the location of x is performed using a delete followed by an insert. The average time and I/O complexities of all three operations are constant. With both insert and delete operations, only Voronoi neighbors of the point x (and hence its Voronoi record) are changed. These changes must also be applied to the Voronoi records of these points which are directly accessible through that of x. According to Property V-3, the average number of Voronoi neighbors of a point is six. Therefore, the average time and I/O complexities of insert/delete/update operations on VoR-Trees are constant.

C. KNN QUERY

Figures 13 and 14 show the pseudo-code of VR-1NN and VR-kNN, respectively.

Algorithm VR-kNN (point q, integer k)
01. $NN(q)$ = VR-1NN(q);
02. minheap $H = \{(NN(q), D(NN(q), q))\}$;
03. $Visited = \{NN(q)\}$; $counter = 0$;
04. WHILE $counter < k$ DO
05. remove the first entry p from H;
06. OUTPUT p; increment $counter$;
07. FOR each Voronoi neighbor of p such as p' DO
08. IF $p' \notin Visited$ THEN
09. add $(p', D(p', q))$ into H and p' into $Visited$;

Figure 14: kNN algorithm using VoR-Tree

Algorithm VR-RkNN (point q, integer k)
01. minheap $H = \{\}$; $Visited = \{\}$;
02. FOR $1 \leq i \leq 6$ DO minheap $S_{cnd}(i) = \{\}$;
03. $VN(q)$ = FindVoronoiNeighbors(q);
04. FOR each point p in $VN(q)$ DO
05. add $(p, 1)$ into H; add p into $Visited$;
06. WHILE H is not empty DO
07. remove the first entry $(p, gd(q,p))$ from H;
08. i = sector around q that contains p ;
09. p_n = last point in $S_{cnd}(i)$ (infinity if empty);
10. IF $gd(q,p) \leq k$ and $D(q,p) \leq D(q,p_n)$ THEN
11. add $(p, D(q,p))$ to $S_{cnd}(i)$;
12. FOR each Voronoi neighbor of p such as p' DO
13. IF $p' \notin Visited$ THEN
14. $gd(q,p') = gd(q,p) + 1$;
15. add $(p', gd(q,p'))$ into H and p' into $Visited$;
16. FOR each candidate set $S_{cnd}(i)$ DO
17. FOR the first k points in $S_{cnd}(i)$ such as p DO
18. p_k = k-th NN of p;
19. IF $D(q,p) \leq D(p_k,p)$ THEN OUTPUT p ;

Figure 15: RkNN algorithm using VoR-Tree

Algorithm VR-kANN (set Q, integer k, function f)
01. p_q = FindCentroidNN(Q, f);
02. minheap $H = \{(p_q, 0)\}$;
03. minheap $RH = \{(p_q, adist(p_q, Q))\}$;
04. $Visited = \{p_q\}$; $counter = 0$;
05. WHILE H is not empty DO
06. remove the first entry p from H;
07. WHILE the first entry p' of RH has
08. $adist(p', Q) \leq amindist(V(p), Q)$ DO
09. remove p' from RH; output p';
10. increment $counter$; if $counter = k$ terminate;
11. FOR each Voronoi neighbor of p such as p' DO
12. IF $p' \notin Visited$ THEN
13. add $(p', amindist(V(p'), Q))$ into H;
14. add $(p', adist(p', Q))$ into RH;
15. add p' into $Visited$;

Figure 16: kANN algorithm using VoR-Tree

D. R$_K$NN QUERY

Figure 15 shows the pseudo-code of VR-RkNN.

Correctness:

LEMMA 4. *Given a query point q, VR-RkNN correctly finds RkNN(q).*

PROOF. It suffices to show that the filter step of VR-RkNN is safe. This follows building the filter based on Lemmas 1, 2, and 3. □

Complexity: VR-RkNN once finds NN(q) starts finding k closest points to q in each partition. It requires retrieving $O(k)$ Voronoi records to find these candidate points as they are at most k edges far from q. Finding k NN of each candidate point also requires accessing $O(k)$ Voronoi records. Therefore, the I/O complexity of VR-RkNN is $O(\Phi(|P|) + k^2)$ where $\Phi(|P|)$ is the complexity of finding NN(q).

E. KANN QUERY

Figure 16 shows the pseudo-code of VR-kANN.

Centroid Computation: When q can be exactly computed (e.g., for f=*max* it is the center of smallest circle containing Q), VR-kANN performs a 1NN search using VoR-Tree and retrieves p_q. However, for many functions f, the centroid q cannot be precisely computed [12]. With f=*sum*, q is the Fermat-Weber point which is only approximated numerically. As VR-kANN only requires the closest point to q (not q itself), we provide an algorithm similar to *gradient descent* to find p_q[2]. Figure 17 illustrates this algorithm. We first start from a point close to q and find its closest point p_1 using VR-1NN (e.g., the *geometric centroid* of Q with $x = (1/n)\sum_{i=1}^{n} q_i.x$ and $y = (1/n)\sum_{i=1}^{n} q_i.y$) for f=*sum*). Second, we compute the partial derivatives of f=$adist(q, Q)$ with respect to variables $q.x$ and $q.y$:

$$\partial_x = \frac{\partial adist(q,Q)}{\partial x} = \sum_{i=1}^{n} \frac{(x-x_i)}{\sqrt{(x-x_i)^2+(y-y_i)^2}}$$
$$\partial_y = \frac{\partial adist(q,Q)}{\partial y} = \sum_{i=1}^{n} \frac{(y-y_i)}{\sqrt{(x-x_i)^2+(y-y_i)^2}} \quad (2)$$

Computing ∂_x and ∂_y at point p_1, we get a direction d_1. Drawing a ray r_1 originating from p_1 in direction d_1 enters the Voronoi cell of p_2 intesecting its boundary at point x_1. We compute the direction d_2 at x_1 and repeat the same process using a ray r_2 originating from x_1 in direction d_2 which enters $V(p_q)$ at x_2. Now, as we are inside $V(p_q)$ that includes centroid q, all other rays consecutively circulate inside $V(p_q)$. Detecting this situation, we return p_q as the closest point to q.

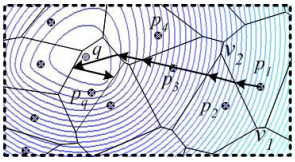

Figure 17: Finding the cell containing centroid q

Minimum aggregate distance in a Voronoi cell: The function $amindist(V(p), Q)$ can be conservatively computed as $adist(vq_1, \ldots, vq_n)$ where vq_i=$mindist(V(p), q_i)$ is the minimum distance between q_i and any point in $V(p)$. However, when the centroid q is outside $V(p)$, minimum $adist()$ happens on the boundary of $V(p)$. Based on this fact, we find a better lower-bound for $amindist(V(p), Q)$. For point p_1 in Figure 17, if we compute the direction d_1 and ray r_1 as stated for centroid computation, we realize that $amindist(p', Q)$ ($p' \in V(p)$) is minimum for a point p' on the edge v_1v_2 that intersects with r_1. The reason is the circular convex shape of SRs for $adist()$. Therefore, $amindist(V(p), Q)$ returns $adist(vq_1, \ldots, vq_n)$ where vq_i=$mindist(v_1v_2, q_i)$ is the minimum distance between q_i and the Voronoi edge v_1v_2.

Correctness:

LEMMA 5. *Given a query point q, VR-kANN correctly and incrementally finds kANN(q) in the ascending order of their $adist()$ values.*

PROOF. It suffices to show that when VR-kANN reports p, it has already examined/reported all the points p' where $adist(p', Q) \leq adist(p, Q)$. VR-kANN reports p when for all the cells V in H, we have $amindist(V, Q) \geq adist(p, Q)$. That is, all these visited cells are outside SR of p. In Figure 17, p_2 is reported when H contains only the cells on the boundary of the grey area which contains SR of p_2. As VR-kANN starts visiting the cells from the $V(p_q)$ containing centroid q, when reporting any point p, it has already examined/inserted all points in SR of p in RH. As RH is a minheap on $adist()$, the results are in the ascending order of their $adist()$ values. □

Complexity: The Voronoi cells of visited points constitute an almost-minimal set of cells covering SR of result (including k points). These cells are in a close edge distance to the returned k points. Hence, the number of points visited by VR-kANN is $O(k)$. Therefore, the I/O complexity of VR-kANN is $O(\Phi(|P|) + k)$ where $\Phi(|P|)$ is the complexity of finding the closest point to centroid q.

General aggregate functions: In general, any kANN query with an aggregate function for which SR of a point is continuous is supported by the pseudo-code provided for VR-kANN. This covers a large category of widely used functions such as *sum*, *max* and weighted *sum*. With functions such as f=*min*, each SR consists of n different circles centered at query points of Q.

[2][12] uses a similar approach to approximate the centroid.

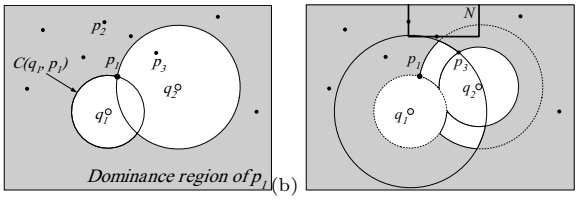

Figure 18: Dominance regions of a) p_1, and b) $\{p_1, p_3\}$

As the result, Q has more than one centroid for function f. To answer a kANN query with these functions, we need to change VR-kANN to perform parallel traversal of $VD(P)$ starting from the cells containing each of n_c centroids.

F. SPATIAL SKYLINE QUERY (SSQ)

Given the set $Q = \{q_1, \ldots, q_n\}$ of query points, the Spatial Skyline (SSQ) query returns the set $S(Q)$ including those points of P which are not *spatially dominated* by any other point of P. The point p spatially dominates p' iff we have $D(p, q_i) \leq D(p', q_i)$ for all $q_i \in Q$ and $D(p, q_j) < D(p', q_j)$ for at least one $q_j \in Q$ [14]. Figure 18a shows a set of nine data points and two query points q_1 and q_2. The point p_1 spatially dominates the point p_2 as both q_1 and q_2 are closer to p_1 than to p_2. Here, $S(Q)$ is $\{p_1, p_3\}$.

Consider circles $C(q_i, p_1)$ centered at the query point q_i with radius $D(q_i, p_1)$. Obviously, q_i is closer to p_1 than any point outside $C(q_i, p_1)$. Therefore, p_1 spatially dominates any point such as p_2 which is outside all circles $C(q_i, p_i)$ for all $q_i \in Q$ (the grey region in Figure 18a) For a point p, this region is referred as the *dominance region* of p [14].

SSQ was introduced in [14] in which two algorithms B^{2}S^2 and VS^2 were proposed. Both algorithms utilize the following facts:

LEMMA 6. *Any point $p \in P$ which is inside the convex hull of Q ($CH(Q)$)[3] or its Voronoi cell $V(p)$ intersects with $CH(Q)$ is a skyline point ($p \in S(Q)$). We use* definite skyline points *to refer to these points.*

LEMMA 7. *The set of skyline points of P depends only on the set of vertices of the convex hull of Q (denoted as $CH_v(Q)$).*

The R-tree-based B^{2}S^2 is a customization of a general skyline algorithm, termed BBS, [11] for SSQ. It tries to avoid expensive dominance checks for the definite skyline points inside $CH(Q)$, identified in Lemma 6, but also prunes unnecessary query points to reduce the cost of each examination (Lemma 7). VS^2 employs the Voronoi diagram of the data points to find the first skyline point whose local neighborhood contains all other points of the skyline. The algorithm traverses the Voronoi diagram of data points of P in the order specified by a monotone function of their distances to query points. Utilizing $VD(P)$, VS^2 finds *all* definite skyline points without any dominance check.

While both B^{2}S^2 and VS^2 are efficiently processing SSQs, there are two drawbacks: 1) B^{2}S^2 still uses the rectangular grouping of points together with conservative *mindist*() function in its filter step and hence, similar to MBM for kANN queries, it fails to prune many nodes. To show a scenario, we first define the dominance region of a set S as the union of the dominance regions of all points of S (grey region in Figure 18b for S=$\{p_1, p_3\}$). Any point in this region is spatial dominated by at least one point $p \in S$. Now, consider the MBR of R-tree node N in Figure 18b. B^{2}S^2 does not prune N (visits N) as we have $mindist(N, q_2)$<$D(p_1, q_2)$ and $mindist(N, q_1)$<$D(p_3, q_1)$, and hence N is not dominated by neither by p_1 nor by p_3. However, N is completely inside the dominance region of $\{p_1, p_3\}$ and cannot contain any skyline point. 2) VS^2, while computationally more efficient that B^{2}S^2, provides no guarantee on its I/O-efficiency. Also, the algorithm does not support arbitrary monotone functions for reporting the result.

We propose our VoR-Tree-based algorithm for SSQ which incrementally returns the skyline points ordered by a monotone function provided by the user (progressive similar to BBS [11] and B^{2}S^2). To start, we first study the search region of a set S in SSQ. This is the region that may contain points that are not spatially dominated by a point of S. Therefore, SR is easily the complement of the dominance region of S (white region in Figure 18b). It is straightforward to see that SR of S is a continuous region as it is defined based on the union of a set of concentric circles $C(q_i, p_i)$.

[3]The unique smallest convex polygon that contains Q.

Algorithm VR-S^2 (set Q, function f)
01. compute the convex hull $CH(Q)$;
02. p_q = FindCentroidNN(Q, f);
03. minheap $H = \{(p_q, 0)\}$;
04. minheap $RH = \{(p_q, adist(p_q, Q))\}$;
05. set $S(Q) = \{\}$; $Visited = \{p_q\}$;
06. WHILE H is not empty DO
07. remove the first entry p from H;
08. WHILE the first entry p' of RH has $adist(p', Q) \leq amindist(V(p), Q)$ DO
09. remove p' from RH;
10. IF p' is not dominated by $S(Q)$ THEN
11. add p' into $S(Q)$;
12. FOR each Voronoi neighbor of p such as p' DO
13. IF $p' \notin Visited$ THEN
14. add p' into $Visited$;
15. IF $V(p')$ is not dominated by $S(Q)$ and RH THEN
16. add $(p', amindist(V(p'), Q))$ into H;
17. IF p' is a definite skyline point or p' is not dominated by $S(Q)$ and RH THEN
18. add $(p', adist(p', Q))$ into RH;
19. WHILE RH is not empty DO
20. remove the first entry p' from RH;
21. IF p' is not dominated by $S(Q)$ THEN
22. add p' into $S(Q)$;

Figure 19: SSQ algorithm using VoR-Tree

An I/O-efficient algorithm once finds a set of skyline points S must examine only the points inside the SR of S. Our VR-S^2 algorithm shown in Figure 19 satisfies this principle. VR-S^2 reports skyline points in the ascending order of a user-provided monotone function f=*adist*(). It maintains a result minheap RH that includes the *candidate* skyline points sorted on *adist*() values. To maintain the order of output, we only add these candidate points into the final *ordered* $S(Q)$ when no point with less *adist*() can be found.

The algorithm's traversal of $VD(P)$ is the same as that of VR-kANN with aggregate function *adist*() (compare the two pseudo-codes). Likewise, VR-S^2 uses a minheap H sorted on *amindist*($V(p), Q$). It starts this traversal from a definite skyline point which is immediately added to the result heap RH. This is the point p_q whose Voronoi cell contains the centroid of function f (here, f=*sum*).

At each iteration, VR-S^2 deheaps the first entry p of H (Line 7). Similar to VR-kANN, it examines any point p' in RH whose *adist*() is less than p's key ($amindist(V(p), Q)$) (Line 8). If p' is not dominated by any point in $S(Q)$, it adds p' to $S(Q)$ (see Lemma 8). Similar to B^{2}S^2 and VS^2, for dominance checks VR-S^2 employs only the vertices of convex hull of Q ($CH_v(Q)$) instead of the entire Q (Lemma 7). Subsequently, accessing p's Voronoi records, it examines unvisited Voronoi neighbors of p (Line 12). For each neighbor p', if $V(p')$ is dominated by any point in $S(Q)$ or RH (discussed later), VR-S^2 discards p'. The reason is that $V(p')$ is entirely outside SR of current $S(Q) \cup RH$. Otherwise, it adds p' to H. At the end, if p' is a definite skyline point or is not dominated by any point in $S(Q)$ or RH, VR-S^2 adds it to RH. When the heap H becomes empty, any remaining point in RH is examined against the points of $S(Q)$ and if not dominated is added to $S(Q)$ (Line 19). In Figure 8, VR-S^2 visits p_1-p_{27} and incrementally returns the ordered set $S(Q)$=$\{p_1, p_2, p_3, p_6, p_8, p_9, p_{10}\}$.

Spatial domination of a Voronoi cell: To provide a safe pruning approach, we define a conservative heuristic for the dom-

ination of $V(p)$. We declare $V(p)$ as spatially dominated if we have $mindist(V(p), q_i) \geq D(s, q_i)$ for a point s in current candidate $S(Q)$. We show that all points of $V(p)$ are dominated. Assume that the above condition holds. For any point x in $V(p)$, we have $D(x, q_i) \geq mindist(V(p), q_i)$. By transitivity we get $D(x, q_i) \geq D(s, q_i)$. That is, each x in $V(p)$ is spatially dominated by $s \in S(Q)$. For example, $V(p_{13})$ is dominated by p_1 as we have

$mindist(V(p_{13}), q_1) = 12 > D(p_1, q_1) = 2,$
$mindist(V(p_{13}), q_2) = 16 > D(p_1, q_2) = 15,$
$mindist(V(p_{13}), q_3) = 33 > D(p_1, q_3) = 23.$

Correctness:

LEMMA 8. *Given a query set Q, VR-S^2 correctly and incrementally finds skyline points in the ascending order of their adist() values.*

PROOF. To prove the order, notice that VR-S^2's traversal is the same as VR-kANN's traversal. Thus, according to Lemma 5 the result is in the ascending order of $adist()$.

To prove the correctness, we first prove that if p is a skyline point ($p \in S(Q)$) then p is in the result returned by VR-S^2. The algorithm examines all the points in the SR of the result it returns which is a superset of the actual $S(Q)$. As any un-dominated point is in this SR, VR-S^2 adds p to its result. Then, we prove if VR-S^2 returns p, then we have p is a real skyline point. The proof is by contradiction. Assume that p is spatially dominated by a skyline point p'. Earlier, we proved that VR-S^2 returns p' at some point as it is a skyline point. We also proved that when VR-S^2 adds p to its result set, it has already reported p' as we have $adist(p', Q) < adist(p, Q)$. Therefore, while examining p, VR-S^2 has checked it against p' for dominance and has discarded p. This contradicts our assumption that p is in the result of VR-S^2. □

Complexity: Similar to VR-kANN, VR-S^2 also visits only the points neighboring a point in the result set $S(Q)$. Hence, it accesses only $O(|S(Q)|)$ Voronoi records. Therefore, the I/O complexity of VR-S^2 is $O(|S(Q)| + \Phi(|P|))$ where $\Phi(|P|)$ is the I/O complexity of finding the point from which VR-S^2 starts traversing $VD(P)$.

Our experiments with three datasets showed that VR-S^2 always outperforms the competitor algorithms. As the second best algorithm, VS^2 accesses almost the same number of disk pages as VR-S^2 does. However, notice that VR-S^2 not only provides proven bounds on its I/O cost but also utilizes arbitrary functions to order its result.