Voronoi-Neighboring Regions Tree for Efficient Processing of Location Dependent Queries

Hassenet Slimani[1], Faïza Najjar[2], Yahya Slimani[3]

[1, 3] *Faculty of SciencesComputer Science Dep.1060, Tunisia*

[2] *National School of Computer Sciences2010, Manouba, Tunisia*

Hassenet.Slimani@gmail.com Faïza.Najjar@ensi.rnu.tn Yahya.Slimani@fst.rnu.tn

Abstract

As a data management technique, indexing aims to judicious organization of data allowing efficient query processing. In the context of location based services (LBSs), indexing techniques are, substantially, affected by location dependency and modes of data access.

This paper focuses on the processing of location dependent queries (nearest neighbors and range queries) based on indexing structures with respect to the context of LBSs. In fact, traditional indexes (R-trees, kd-trees …) adopt either overlapped partitioning schemes or backtracked search algorithms. These features increase response times for an in-memory index within an on-demand data access mode. They, also, make such indexes impractical for a broadcast data access mode. As a solution, we propose a novel index called VNR-tree (Voronoi-Neighboring Regions tree), based on the Delaunay Triangulation. VNR-tree adopts a non-overlapping partitioning scheme, supports backtracking-free search algorithms and generates a special clustering of the Voronoi-neighboring regions. This clustering is adequate to the processing of several types of location dependent queries based on the exploration of the mobile client's Voronoi-neighborhoods. We conduct various experiments to compare VNR-tree with R-tree as in-memory indexes in the on-demand data access mode. Main results show that VNR-tree outperforms R*-tree significantly.*

Keywords: *Location-dependent queries, Indexing structures, Range Search, Nearest neighbors search, Continuous nearest neighbors search, Voronoi Diagram, Delaunay Triangulation.*

1. Introduction

Leveraging the Internet and the tremendous advances in wireless communications and localization technologies (GPS and others), mobile network operators rely on location based services (LBSs) as a key-asset to provide highly personalized and profitable services. An example of such services is the location dependent information services which provide a mobile client with relevant information regarding her/his current location. Examples of this information may include:

- 'The three nearest hotels' (Nearest Neighbors search—NNs search).
- 'The nearest sights along his/her route to a given destination' (Continuous-Nearest Neighbors search—CNNs search).
- 'The sights within three miles or at two to three miles' (Distance-Range search—DR search).

These examples correspond to queries whose answers depend on their issuer's location and they are called location dependent queries (LDQs). The processing of such queries needs a pre-processing step that organizes the set of data objects (like hotels, sights, gas stations…) into an indexing structure that would make the retrieval of answer objects more efficient than the exhaustive search. Data objects are often called points of interest or point sites or sites, for short.

In this paper, we focus on indexing techniques that allow efficient processing of these LDQs queries within the context of LBSs. Such a context has several particular parameters [1, 2] that differentiate it from the classic spatial context. These parameters mainly include the mobility of objects and the wireless data access mode. The last parameter sets the way in which data are disseminated to the mobile client. In LBSs, there are two modes of data access:

- On-demand access: queries are triggered by mobile users and processed by the server (client/server data access). At the server side, indexes are stored either in main-memory (in-memory indexes) or in disk (in-disk indexes). With the advent of big and cheap technologies in main memories, interest in in-memory indexes is growing [3], since they allow faster query processing and more predictable performances than in-disk indexes.
- Periodic broadcast access: data are periodically broadcast by the server on a wireless channel and the client is responsible for filtering its desirable data. The role of an index which is broadcast before the indexed data is to make this filtering operation fast and efficient (only pertinent data is downloaded on the client device) [2, 4]. Such a broadcast index is called an air-index.

Generally, a mobile environment needs faster response time than the classic spatial environment so as the answer remains pertinent with regard to the mobile client's location. Also, in a periodic broadcast data access mode, an index with backtracked search algorithms is impractical since it needs to process a query on an unpredictably increasing number of broadcast cycles.

Most of existing indexes, including the well-known R*-tree [5] rely on search algorithms that cannot avoid backtracking. Often, this backtracking is increased by an overlapping partitioning scheme. These shortcomings increase response times for the in-memory usage of the index in an on-demand data access mode. They also make such an index impractical for a periodic broadcast data access mode.

As a solution, we propose the VNR-tree (Voronoi-Neighboring Regions tree), a tree index based on the Voronoi diagram of the indexed set of sites.

The Voronoi diagram (VD) is an interesting structure in computational geometry [6]. It is the spatial solution for the nearest neighbor problem. Given a set of N point sites S (S={S1,S2,…,SN}), the VD of S (VD(S)) is the subdivision of the plane into N disjoint regions/cells according to the nearest-neighbor rule: each region/cell corresponds to one site and is composed by the set of the plane's points that have this site as the nearest site among all sites of S (Figure 1). VD(S) has a dual graph called the Delaunay triangulation of S (DT(S)). Two sites Si and Sj are connected in DT(S) if and only if their Voronoi cells share a Voronoi edge i.e. they are neighbors. Both of the VD and the DT are efficient for exploring the spatial neighborhoods of a given query point (which, actually, corresponds to the query issuer i.e. the mobile client).

The proposed index, VNR-tree, has several interesting characteristics:

- It is constructed based on an equitable binary partitioning that makes VNR-tree a balanced binary search tree. This is likely to make its performances more predictable.
- It adopts a non-overlapping partitioning scheme and supports backtracking-free search algorithms. This feature is to enhance response times and to widen usability of the structure to the broadcast data access mode especially that it hasn't an overhead in storage cost compared to the existing indexes.
- It generates a special clustering, at leaf nodes, of the indexed sites in groups of neighboring sites of the solution space (the voronoi diagram). Linking the leaf nodes of the index in a way that captures these neighborships allows for an efficient processing of several classes of LDQs based on the exploration of the mobile client's Voronoi-neighborhoods. While, linking the leaf nodes of a tree index is not a novel technique in the design of indexing structures, VNR-tree is the first proposal in the spatial context, to the best of our knowledge.
- It consistently provides better response time than R*-tree when used as in-memory indexes in the on-demand data access mode.

The remainder of the paper is organized as follows: In Section 2, we review the related work. Section 3 defines the new index VNR-tree. Section 4 presents VNR-tree based search algorithms. Section 5 evaluates performances of the proposed index. Finally, Section 6 concludes the paper with a summary and a glance at future work.

2. Related Work

There are many indexes that offer support for nearest neighbors and range queries: kd-trees, quad-trees [6] and so on… Nevertheless, R-trees [7] are the most used in a spatial context. They are preferred for their good performances and their efficient support for many types of queries [8, 9, 10].

R-Trees index spatial objects approximated by their Minimal Bounding Rectangles (MBRs) [7, 10]. An R-tree index is a balanced tree corresponding to a hierarchy of non-disjoint rectangles where the leaves are MBRs that enclose directly the indexed data objects (data MBRs). Whatever R-tree variant is used, using the index for data points consists in considering degenerated rectangles. The problem of overlapping between MBRs in internal nodes of the index persists even when data MBRs are degenerated.

R-trees are originally in-disk indexes. Nevertheless, in-memory R-trees have regained interest later [3] with the proliferation of big and cheap technologies of memory. For in-disk R-trees, maximum number of entries per node (fan-out) is set as a function of the disk page size. For in-memory R-trees, the fan-out is set as a function of the pre-fixed node size in main memory.

An R-tree-based processing of any spatial query relies on branch-and-bound search algorithms. These algorithms are backtracked, they are based on some pruning strategies (distance metrics) to reduce the number of visited index nodes and, so, to filter the relevant data.

R-trees support nearest neighbors (NNs) search and distance range (DR) search through two types of backtracked-algorithms. The non-incremental variant [1, 11, 12] consists in depth-first order traversal algorithms which use distance metrics to prune irrelevant branches.

The incremental variant (best first search) [13] reports answer sites one by one, so that the algorithm can be used in a pipelined fashion for complex queries involving proximity.

Recently, in [14], the authors propose an enhancement of R-tree's support for nearest neighbors search based on the Voronoi diagram. The proposal, still, resorts to the backtracked best first search algorithm in order to localize the nearest neighbor (NN). Besides, this enhancement targets an in-disk usage of R-trees.

R-trees' support for continuous nearest neighbors (C-NNs) search had been studied in [15] where a backtracked search algorithm that solves the problem by performing a single query for the whole input segment was proposed.

Also, in [16], the authors propose an index that supports search of the nearest neighbor and of the continuous nearest neighbor with better performances than the R*-tree within disk and broadcast data access modes. Nevertheless, the index doesn't have straightforward usages in nearest neighbors search, continuous nearest neighbors search or DR search.

In many studies, R*-tree is preferred among other variants of R-tree [5, 10], since it has better search performance. So, in our study, we choose R*-tree as a representative of the existing solutions.

3. VNR-tree: The Tree Index Structure

In a 2-dimensional space, let S be a set of N points of interest, S= {S1, S2,…, SN}, situated within a zone of interest for a location based service. In general, this zone can be represented by a rectangular region called the search space. Let VD(S) be the Voronoi diagram (VD) of S and DT(S) be its Delaunay triangulation (DT) (example in Figure 1). For each site Si, Si.x and Si.y denote the x-coordinate and y-coordinate of Si, respectively.

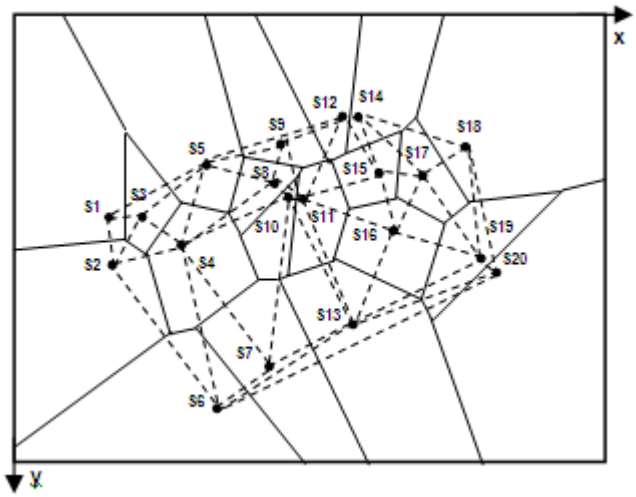

Figure 1. The VD of 20 Sites Randomly-Generated (continued lines), its DT (Dashed Lines).

The construction process of the VNR-tree needs as input the set of sites, S, and its Delaunay triangulation represented as a set of *neighborhood edges*, DT(S)={*SiSj*, where Si and Sj are two *Voronoi-neighboring sites* of S}.

Nevertheless, we present illustrations with the Voronoi diagram of S, also, for comprehensiveness and easy reading. VNR-tree construction proceeds in two phases:

3.1. Phase 1: Clustering of Indexed Sites into Voronoi- Neighboring Regions

This is done through a recursive binary partitioning of the search space's sites into two complementary subsets containing almost the same number of sites: the Left SubSet (LSS) and the Right SubSet (RSS). The partitioning frontier is defined as the set of the broken neighborhoods i.e. the edges of the Delaunay Triangulation between the separated sites.

A partitioning style corresponds to a sorting of the sites according to one dimension (increasing x-coordinates or increasing y-coordinates) and to their division into two almost equal subsets.

There are two or four possible partitioning styles according to whether the number of sites is even or odd. At each step of the equitable binary partitioning, the partitioning style that has the smallest partitioning frontier is chosen in order to keep the Voronoi-neighboring sites clustered in the same subset. In case of equality, the partitioning that maximizes the sum of lengths for the broken Delaunay triangulation edges is chosen. This is to promote the partitioning that is likely to keep near sites clustered.

The recursive binary partitioning ends when all the sites of the partitioned subset are on the partitioning frontier: This is a final partitioning step which corresponds to a Voronoi-Region composed of Voronoi-neighboring sites.

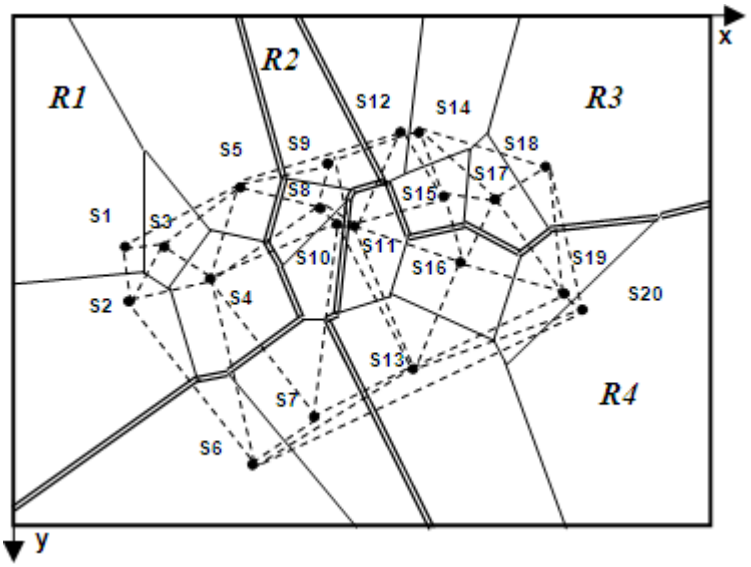

Figure 2. The Final Clustering of Sites Induced by VNR-tree (Edges in Double Line).

In Figure 2, the double-lined edges show the final Voronoi regions (R1, R2, R3 and R4) obtained through VNR-tree's recursive binary partitioning:

- A first x-dimensional partitioning divides the sites {S1,S2,…,S20} into a Left Subset LSS={S1,S2,…,S10} and a right subset RSS={S11,…,S20}. The partitioning frontier is composed of 8 Delaunay triangulation edges: {S9S12, S8S11, S10S11, S10S13, S7S13, S7S20, S5S12, S6S13}.
- Recursively, in the left subset {S1,…,S10}, another x-dimensional partitioning is chosen. It divides the sites into a Left Subset LSS={S1,S2,…,S5} and a Right Subset RSS={S6,…,S10}.

- The next partitioning step in the left subset {S1,S2,…,S5} is final since all sites of the subset are on the partitioning frontier (4 broken Delaunay triangulation edges {S1S5,S3S5,S3S4,S2S4}). This step determines the Voronoi Region R1.
- Similarly, Voronoi regions R2, R3 and R4 are obtained.

Now, comes the mapping of this partitioning to a binary search tree. A non-final partitioning step corresponds to an internal node of the VNR-tree and its partitioning frontier is represented by an axis-parallel line of the same direction; it is the median line between the rightmost point in the left subset and the leftmost point in the right subset having the same direction as the partitioning frontier.

Figure 3 illustrates the median lines corresponding to the partitioning frontiers of our example.

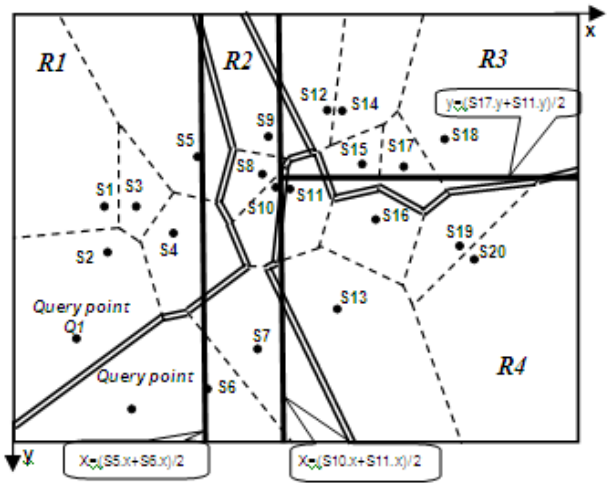

Figure 3. VNR-tree Partitioning with Partitioning Frontiers (doubled-bold lines) Represented as *axis-parallel* Median Lines (bold *axis parallel* lines).

A leaf node (Li) represents a final partitioning step. It corresponds to a Voronoi region (Ri) whose total sites fall on the partitioning frontier. It is composed of the sites of its region.

Figure 4 shows the first part of the VNR-tree under construction, Ni denotes an internal node and Li denotes a leaf node. The structure of an internal node comprises the dimension (x or y) of the partitioning frontier it represents and the median value; both give the equation of the median line representing the partitioning frontier. The structure of a leaf node comprises the sites of its corresponding Voronoi region.

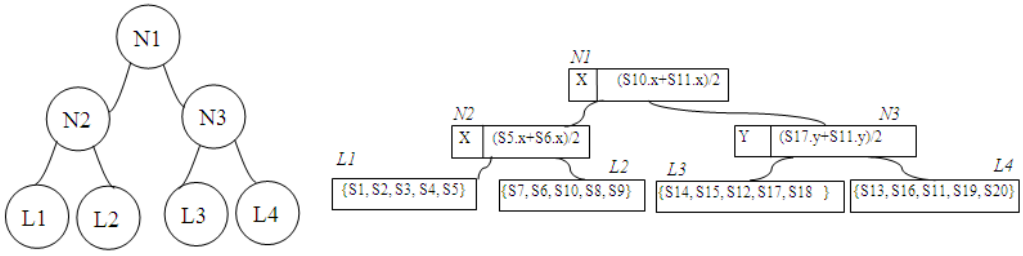

Figure 4. Clustering of the Indexed Sites into Voronoi-neighboring Regions.

This mapping allows:

- A pertinent clustering, at leaf nodes, of the Voronoi-neighboring sites into Voronoi- neighboring Regions (VNR). Linking these leaf nodes, in a way that captures neighborhood links between underlying regions, allows the exploration of a query point's Voronoi-neighborhoods. This represents an adequate feature for efficient processing of many types of queries: nearest neighbors queries, continuous neighbors queries, range queries, directional queries, window queries…
- A fast guiding of a query point into an Initial Leaf (IL) through the comparison of its coordinates with the median lines of the internal nodes. The identified Initial Leaf corresponds to an Initial Voronoi-Region (IVR) which may correspond to the Voronoi region that contains the query point (in Figure 3, query point Q1 is localized in the Voronoi region R1 which corresponds to leaf L1) or to a neighboring region (in Figure 3, query point Q2 is localized, through the comparison of its coordinates with the median lines, in region R1 which is a neighboring region to Q2's region, R2). In the 1st case, the nearest neighbor (NN) is the site of the Initial Leaf that corresponds to the least distance from Q1. In the 2nd case, the nearest neighbor (NN) of Q2 is initialized to the nearest site among sites of the Initial Leaf (L1 for Q2) and corrected through neighborhoods exploration once the Initial Leaf is linked to its neighboring leaf nodes (L2). This will be explained in nearest neighbors (NNs) search algorithm.

3.2. Phase 2: Capturing Neighborships between Voronoi-regions Through the Linking of the Index Leaves

Linking the leaf nodes of VNR-tree in a way that captures neighborhood links between underlying regions allows search of candidate sites based on the scanning of the query point's neighborhoods. We propose a linking procedure that allows exploring the whole search space starting from any leaf node.

- A leaf Lj is linked to another leaf Li if and only if they share at least one VD edge. This neighborhood link can be according to one of the following directions:
- RIGHT: the right side along x dimension (RX-neighbor) if all Lj sites have greater x-coordinates than all Li sites.
- LEFT: the left side along x dimension (LX-neighbor) if all Lj sites have lower x-coordinates than all Li sites.
- DOWN: the right side along y dimension (RY-neighbor) if all Lj sites have greater y-coordinates than all Li sites.
- UP: the left side along y dimension (LY-neighbor) if all Lj sites have lower y-coordinates than all Li sites.

A neighborhood link from leaf node Li to leaf Lj is represented, at the leaf node Li, by three attributes:

- A pointer to leaf Lj denoted by P_Lj.
- The direction of the neighborhood link: LX, RX, LY or RY.

- A Minimal Bounding Rectangle (MBR) of the neighborhood link's bordering sites, which are sites of leaf Lj that share at least one Voronoi edge with leaf Li's sites. This MBR is denoted by MBR_Li/Lj. It is based on the MINimum DISTance (the MINDIST metric defined for R-trees [5, 7, 11]) between the query point and this MBR that a search algorithm decides to expand neighborhoods scanning from leaf Li to leaf Lj or not.

The VNR-tree of our running example is shown in Figure 5. It adds, to the first part (Figure 4), neighborhood links between leaf nodes.

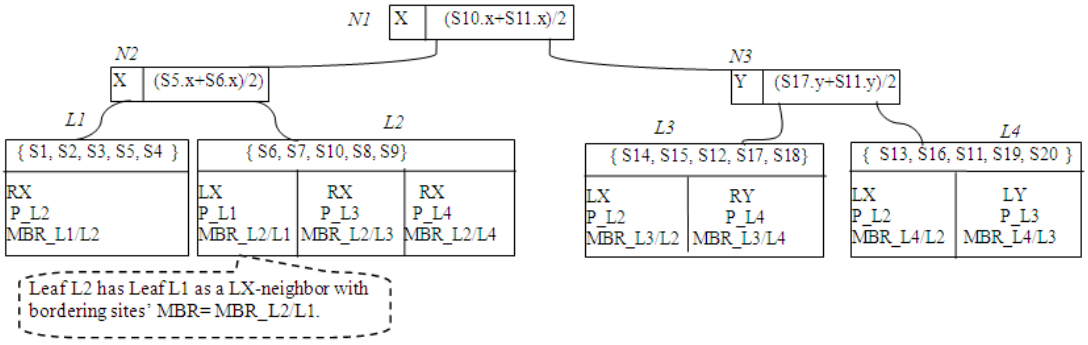

Figure 5. VNR-tree Index.

In Figure 5, leaf L2 has three neighboring leaf nodes L1, L3 and L4 which reflect the three neighboring regions that the corresponding region R2 has (Figure 2 or Figure 3). In fact, region R2 has:

- Region R1 as a LX-neighbor with bordering sites {S2,S4,S5}. This is reflected at leaf L2 in the VNR-tree index of Figure 5 by a pointer to leaf L1 (p_L1) and an MBR of these bordering sites (MBR_L2/L1),
- Region R3 as a RX-neighbor with bordering sites {S12}. This is reflected at leaf L2 in the VNR-tree index of Figure 5 by a pointer to leaf L3 (p_L3) and an MBR of these bordering sites (MBR_L2/L3),
- Region R4 as an other RX-neighbor with bordering sites {S13,S11,S20}. This is reflected at leaf L2 in the VNR-tree index of Figure 5 by a pointer to leaf L4 (p_L4) and an MBR of these bordering sites (MBR_L2/L4).

Overall, the linking procedure follows these steps:

- It visits all the tree index leaf nodes starting from its extreme-left leaf and in the order they appear at the level of leaf nodes in the tree index.
- For each leaf, it determines its neighboring leaves according to two directions: RX and RY. The neighboring regions according to the remaining directions (LX and LY) are deduced directly through symmetry from other leaves.
- When looking for the RX (resp. RY)-neighboring leaves of the current leaf, the linking procedure goes upward the index from current leaf's father node until reaching the first ancestor node having an X (resp. a Y)-dimensional partitioning and for which the current leaf is in the left subtree. From the ancestor node, it finds all bordering sites that the current leaf shares with its RX (resp. RY)-neighboring leaves. This is done through identifying Delaunay triangulation

edges of the ancestor's partitioning frontier having sites of the current leaf as left sites; right sites of these edges are the RX-(resp. RY-) bordering sites of the current leaf. Then, the linking algorithm goes downward the ancestor's Right Child-rooted subtree to identify the leaves containing these bordering sites and link them to the current leaf as RX (resp. RY)-neighboring leaves. These leaves have the current leaf as a LX (resp. LY)-neighboring leaf.

4. Search Algorithms and Analytical Comparison

We present VNR-tree- based search algorithms for three types of queries. A common input of these algorithms is the VNR-tree of a given set (S) of N sites.

4.1. Nearest neighbors search (NNs search)

Having a query point q, let k be the number of the looked for nearest neighbors ($k \geq 1$). Search of the k nearest neighbors of q, based on the VNR-tree is two-phased:

1. Search of q's Initial Leaf (IL): This starts from the root node of the VNR-tree. Then, at an internal node, it decides to follow, exclusively, the left subtree or the right subtree depending on whether the query point is situated to the left or to the right of the node's median line. It ends by reaching an Initial Leaf (IL) which corresponds to the Voronoi region that contains q or to a neighboring region.
2. Neighborhoods scanning: starting from the Initial Leaf (IL), neighborhoods are scanned progressively such that leaf nodes corresponding to nearer neighboring regions from q are always visited before others. This arrangement of leaf nodes during neighborhoods scanning is implemented through the use of a FIFO queue (Q):
    a. Enqueue the Initial Leaf (IL) into Q.
    b. While Q is not empty do
        i. Dequeue from Q (the current leaf).
        ii. Check sites of the current leaf: Every site which is closer to q than the k^{th} current nearest neighbor is inserted into the Result set.
        iii. For each neighborhood link of the current leaf, the pointer to the target leaf is enqueued whenever this requirement is fulfilled: the target leaf is neither visited nor enqueued yet and the MINDIST of q to the neighborhood link's MBR of bordering sites is lower than the distance between q and its k^{th} current nearest neighbor.

4.2. Distance Range Search (DR search)

Having a query point q and a distance range [d1-d2], distance range search allows identifying the set of sites situated from q within the distance range [d1-d2]. VNR-tree supports such a search through a two-phased procedure:

1. Search of q's Initial Leaf (IL): This starts from the root node of the VNR-tree. Then, at an internal node, it decides to follow, exclusively, the left subtree or the

right subtree depending on whether the query point is situated to the left or to the right of the node's median line. It ends by reaching an Initial Leaf (IL) which corresponds to the Voronoi region that contains q or to a neighboring region.

2. Neighborhoods scanning: starting from the Initial Leaf, neighborhoods are scanned progressively such that leaves corresponding to nearer neighboring regions are always visited first. This arrangement of leaves during neighborhoods scanning is implemented through the use of a queue (Q).
    a. Enqueue the Initial Leaf (IL) into Q.
    b. While Q is not empty do :
        i. Dequeue from Q (the current leaf).
        ii. Check sites of the current leaf: Every site whose distance to q falls within the query range [d1-d2] is inserted into the Result set.
        iii. For each neighborhood link of the current leaf, the pointer to the target leaf is enqueued whenever this requirement is fulfilled: the target leaf is neither visited nor enqueued yet and the MINDIST of q to the neighborhood link's MBR of bordering sites is lower than the upper bound of the query range (d2).

4.3. Continuous Nearest Neighbors Search (C-NNs search)

4.3.1. Notations and Hypotheses: Let's consider these two parameters:

- A Query Segment QS = [sPoint, ePoint] where sPoint and ePoint denote the starting point and the ending point of the query segment, respectively.
- The number, k (k>=1), of the looked for nearest neighbors.

Search of the k-nearest neighbors all along the query segment QS has, as an answer, a list of contiguous sub-segments of the query segment such that the k nearest neighbors of each point on the same sub-segment are identical (this is guaranteed once both endpoints of the sub-segment have the same k nearest neighbors).

4.3.2. Steps of the Search Algorithm: Each point p has two attributes: Its coordinates (p.x and p.y) and an array of its k nearest neighbors (this array is labeled *NNs* and denoted by p.NNs[]).

The algorithm goes downward the index starting from the root node.

- *At a non leaf node*, it studies the position of the query segment, QS, regarding the node's median line. There are two cases: either the query segment intersects the (current) median line or not.
    1. When there's no intersection, the query segment is to the left or to the right of the median line and search is forwarded to the left child node or to the right child node, accordingly.
    2. When the query segment intersects the current median line, the intersection point is determined and two sub-segments are generated: one sub-segment is situated to the left of the median line and another is

situated to its right. The search is forwarded for both of the left and the right children with the corresponding sub-segment.

- *At a leaf node*, the algorithm proceeds in two phases :

  I. It determines the k-nearest neighbors for each ending point of *the local sub-segment of QS, [sp, ep]*, (i.e. [sp, ep] is the sub-segment of the query segment situated within the region of the leaf node).

  II. It scans both sets of nearest neighbors of the ending points (*sp.NNs and ep.NNs*) in an ascending order (1^{st}, 2^{nd}, 3^{rd} …, k^{th}) to determine the first i^{th} nearest neighbor they do not share. If such a nearest neighbor doesn't exist (i.e. the ending points, *sp* and *ep*, have the same k nearest neighbors), the sub-segment *[sp, ep]* is added to the result set. Alternatively, the following recursive processing is started:

     a. If the ending points (*sp and ep*) have not the same i^{th} nearest neighbor, then

        i. Determine the intersection point (*interPt*) between the bisector of the segment defined by both i^{th} nearest neighbors and the local query sub-segment (i.e. *interPt=[sp.NNs[i], ep.NNs[i]]* ∩*[sp,ep]*).

        ii. Determine the k nearest neighbors of this intersection point (*interPt.NNs[]*).

        iii. There are two alternatives:

           1. The intersection point shares the same i^{th} nearest neighbor with one of the ending points (*i.e. interPt.NNs[i] is the same as sp.NNs[i] or ep.NNs[i]*). This means that the intersection point (*interPt*) is on an edge of the Voronoi diagram i.e. the i^{th} nearest neighbors of the ending points (*sp.NNs[i] and ep.NNs[i]*) are Voronoi-neighbors. Here, the processing of the i^{th} nearest neighbor is resumed through the dividing of query sub-segment (*[sp, ep]*) to two sub-segments (*[sp, interPt] and [interPt, ep]*). On *[sp, interPt], sp.NNs[i]* is the i^{th} nearest neighbor. On *[interPt, ep], ep.NNs[i]* is the i^{th} nearest neighbor. If ($i<k$) the processing from step (II.a) is recursively restarted for the $(i+1)^{th}$ nearest neighbor, else both subs-segments are added to the result set.

           2. The intersection point has an i^{th} nearest neighbor which is different from those of both ending points. Both sub-segments *[sp, interPt]* and *[interPt, ep]* have different i^{th} nearest neighbors at their endpoints. Re-start the processing from step (II.a) with each sub-segment and, that, for the **i^{th}** nearest neighbor **again**.

     b. Else // i^{th} nearest neighbors of sp and ep are identical

        i. If this i^{th} nearest neighbor is the k^{th} one ($i=k$)

1. Add the sub-segment to the result set.

ii. Else recursively, restart the processing from step (II.a) for the $(i+1)^{th}$ nearest neighbor.

Let's notice, that this search of the k nearest neighbors all along a query segment has a straightforward use for a whole trajectory composed of a set of contiguous segments (Trajectory queries).

4.4 Analytical Comparison

Table 1 shows an analytical comparison between R*-tree and VNR-tree:

Table 1. Analytical Comparison between VNR-tree and R*-tree.

	Construction Time	Storage Space	Search Complexity
R*-tree	O(NlogN)	O(N)	O(N+k)
VNR-tree	O(NM)	O(N)	O(logN +k)

- The construction process of the VNR-tree costs O(NM) where N is the number of the indexed sites and M is the number of edges in the Delaunay Triangulation of these sites. VNR-tree needs slightly more time to be constructed than the R*-tree does. But, an index is usually constructed once and used as long as the dataset it indexes doesn't change. This is suitable for our hypothesis of non-highly dynamic datasets (our data objects are stationary: hotels, gas stations, sights, etc.).
- VNR-tree has a little better worst case computational cost for the processing of the considered queries (nearest neighbors and distance range queries). The experimental study of the next section is going to evaluate factual search performances of both structures.

5. Experimental Evaluation

We evaluate performances of the VNR-tree (VNRT in figures' legends) by comparing it to the R*-tree. In this paper, performance evaluation is limited to the in-memory usage within an on-demand data access mode. For the VNR-tree, we use search algorithms described above. For the R*-tree, we use algorithms proposed in [11], in [13] and in [15] for nearest neighbors search, distance range search, and continuous nearest neighbors search, respectively.

Experimental setting: Experiments are carried out on a Microsoft Windows machine (Processor frequency: 1.6 GHz, Main memory: 1GB). All coding is done in the Java Language.

Datasets: Evaluation is done with several datasets:

- A real dataset composed of 21047 point sites representing the nodes of California City's road network [17].
- Several synthetic datasets (Table 2) generated by a random generator of points which provides a normal non uniform distribution of sites (like that of Figure 1). Datasets of Table 2 have all the same density (number of sites per surface unit). Let's notice that we do not specify the measure unit for space dimensions; it may

be the meter, the mile, the kilometer. Whatever was the measure unit, we use sufficiently large spaces to reflect realistic cases.

- In order to study the effect of increasing the dataset's density, we generate two variants of the dataset D20000. D20000_1 is a dataset of 20000 sites generated within the 2-d box [0-100,000]* [0-100,000]. D10000_2 is a denser dataset of 20000 sites generated within the 2-d box [0-70,000]* [0-70,000].

Table 2. Synthetic Datasets.

Dataset	Dataset size (# sites)	Dimensions of the search space
D5000	5,000	[0-70,000]*[0-70,000]
D10000	10,000	[0-100,000]*[0-100,000]
D15000	15,000	[0-123,000]*[0-123,000]
D20000	20,000	[0-142,000]*[0-142,000]
D25000	25,000	[0-158,000]*[0-158,000]

For each dataset, we construct: the R*-tree, the solution space (the VD) and the VNR-tree.

We evaluate the average response time (ART) for nearest neighbors queries as an average for 100,000 queries randomly generated over the whole search space.

R*-tree performance is evaluated for several values of a node size: 4ko, 1ko, 1/2ko, 1/4ko and 1/8ko (labeled in figures' legends RST-4K, RST-1k, RST-1/2k, RST-1/4k, RST-1/8k, respectively).

Experiments with the real dataset have shown the same behaviors as the synthetic datasets for all types of considered queries. Also, increasing dataset density didn't affect performances of both indexes and that for nearest neighbors search and continuous nearest neighbors search. Related figures are omitted for that reason and for space constraints.

5.1. Nearest neighbors search (NNs search)

Two parameters are involved in the processing of a nearest neighbors query: the number of the requested nearest neighbors (k>=1) and the dataset size.

5.1.1. Effect of parameter k: Given a search space, we watch variation of the response time as a function of k. Figure 6 plots results for dataset D25000. Other datasets have curves of the same appearance.

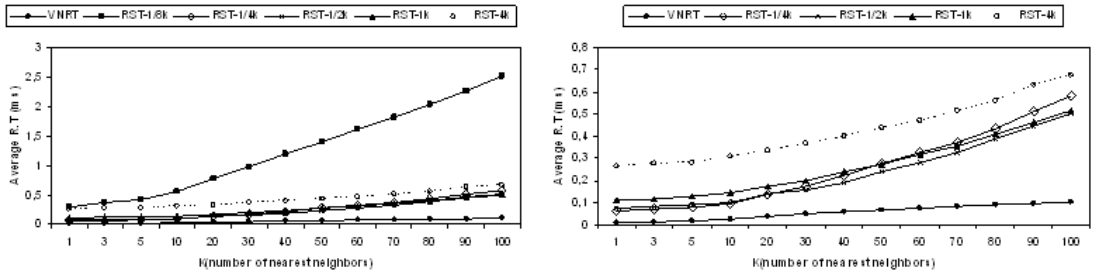

Figure 6. Response time =f(k) for D25000 (zoomed out through omitting RST-1/8k in the 2nd sub-figure).

The following observations are highlighted:

- Performance of the R*-tree improves as the node size decreases and this is to a certain threshold (1/4k). Sorted according to a decreasing response time, we have RST-4k>RST-1k>RST-1/2k. RST-1/2k has almost the same response time as the RST-1/4k. But, the RST-1/8 has an excessive response time which surpasses those provided by R*-trees of other node sizes. For a given dataset, when the node size of the R*-tree decreases, the number of entries per node decreases (the number of entries that the search algorithm has to test is reduced) and the total number of the index nodes increases (the index becomes bigger). But, also, when the node size of the R*-tree increases, the total number of index nodes decreases and the number of entries per node increases. This explains the existence of a threshold of the node size that provides a good response time.
- VNR-tree has an almost constant response time. Conversely, response time for R*-tree grows substantially faster as k increases. In fact, all search algorithms of VNR-tree visit a branch of the tree index (at most O(log N) internal nodes) and, then, perform a scanning of the query point's neighborhoods where they visit only pertinent leaf nodes. Meanwhile, the backtracked branch & bound search algorithms of the R*-tree have a number of visited internal nodes which increases with the query parameter (k for nearest neighbors search).
- For the considered range of dataset sizes (<=25000 sites), VNR-tree keeps outperforming RST-1/4k by a factor ranging from 4 (k=100), to 10 (k=1).

5.1.2. Effect of the dataset size for given values of parameter k: Given a value of k, we watch variation of the response time as a function of the dataset size. This shows scalability degrees of the indexing structures. We show results for two values of k: 1 and 20 (Figure 7).

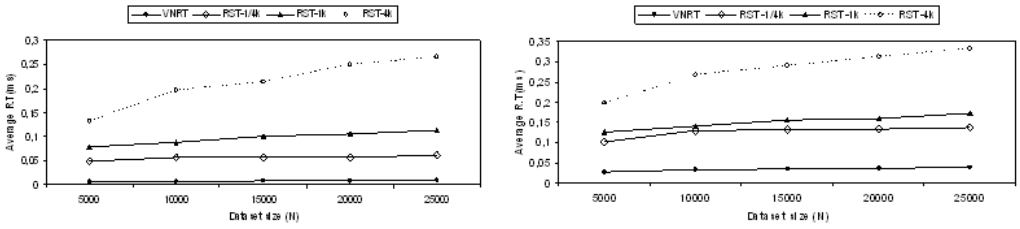

Figure 7. Response time =f(N) for k=1 and for for k=20, respectively.

The following notes are reported:

- The RST-4k is the less scalable indexing structure; its response time grows noticeably fast with the dataset size. This is because RST-4k has a great number of entries per node and that slows down the search. VNR-tree and RST-1/4k have almost constant response times with the increase of the size of the indexed dataset.
- VNR-tree substantially prevails for a quite good range of dataset sizes. This is to confirm efficiency of this index in nearest neighbors search. VNR-tree is constructed based on neighborships of the indexed sites in their VD allowing the clustering of neighboring sites at the level of leaf nodes. Such a feature is to better the response time.

- The case of k =1, is a standalone issue that corresponds to the nearest neighbor search: VNR-tree outperforms the RST-1k by a factor of about 12, it outperforms the RST-1/4k by a factor >= 7.

5.2. Distance Range Search (DR Search)

Two parameters are involved in the processing of a range query: the query range and the dataset.

5.2.1. Effect of the query ranges: Given a search space (a dataset of size N), we watch variation of the response time as a function of query ranges. We vary query ranges from [0-1000] to [0-15000]; this corresponds to a variation of the query range ratio to the search space side length from 0.001 to 0.1. We show results for dataset D25000 (Figure 8). The other datasets have same behaviors.

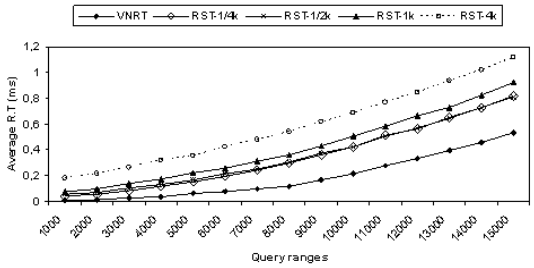

Figure 8. Response time =f(query ranges) for D25000.

The following observations are made:

- Like for nearest neighbors search, performance of the R*-tree improves as the node size decreases and that for a certain threshold (node size=1/4k). (Excessive values for RST-1/8k were omitted for an easy reading of Figure 8.)
- The VNR-tree substantially outperforms R*-tree. Nevertheless, this improvement decreases as the query range becomes larger since larger neighborhoods are to be scanned. Figures of the considered datasets show an improvement factor of VNR-tree over RST-1/4k ranging between 2 (range=[0-15000]) and 8 (range=[0-1000]). VNR-tree achieves an improvement factor between 3 and 7 within the range [0-6000]; it performs well within the local neighborhoods of the query point. VNR-tree adopts a non-overlapping partitioning scheme and non-backtracked search algorithms that allow fast search especially in the vicinity of the query point where non-extended area is to be scanned i.e. a limited number of leaf nodes are to be scanned. Neighborhood scanning is intuitively a natural strategy in identifying sites that are situated within the vicinity of a query point.

5.2.2. Effect of the dataset size for given query ranges: Given a query range, we watch variation of the response time as a function of the dataset size (N). We present results for two radii (3000 and 9000) in Figure 9. The following observations might be highlighted:

- VNR-tree and R*-tree with node sizes ranging between 1/4k and 4k have response times that grow slowly and almost uniformly with the increasing of dataset sizes. This affirms their scalability.

- VNR-tree substantially prevails for a quite good range of dataset sizes and query ranges.

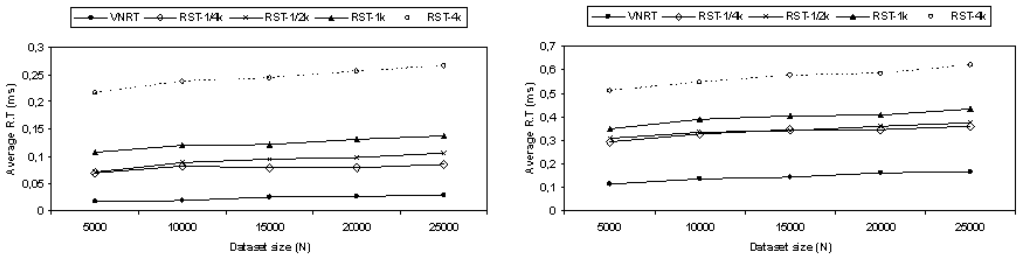

Figure 9. Response time=f(N) for query range [0-3000] and for query range [0-9000], respectively.

5.2.3. Effect of the dataset density: In this experimentation, we increase the dataset density for dataset D20000 by reducing the dimensions of the 2d-box on which the 20,000 sites are generated. Sorted according to increasing density, these datasets are: D20000, D20000_1, and D20000_2.

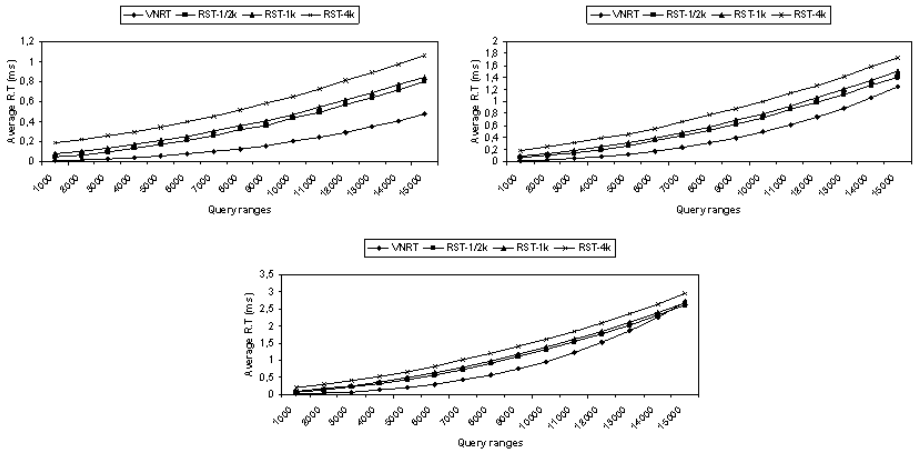

Figure 10. Response time =f(query ranges) for D20000, for D20000_1 ans for D20000_2, respectively.

Through Figure 10, the following impacts can be read:

- The VNR-tree keeps prevailing over R*-tree in the local vicinity of the query point. This improvement decreases as the dataset becomes denser since more answer sites are to be reported. Also, this improvement decreases as the query range becomes larger since wider neighborhoods are to be scanned (more leaf nodes are to be visited): VNR-tree is no longer outperforming RST-1/4k starting from the radius 15000 for the dataset D20000_2 (Figure 10). Generally, a global increasing of the dataset density or a partial one (that affects the distribution of a part of the dataset) diminishes performances of the VNR-tree for distance range queries.
- Compared to performances of VNR-tree for nearest neighbors search with a dense dataset, performances of VNR-tree for range search are sensitive to the increasing of the dataset density. This is due to the fact that, for a nearest

neighbors query, a step of the processing depends on the preceding steps (parameter k^{th} current nearest neighbor is updated at the visit of each leaf node), whereas a step of the processing of a range query depends on the query range upper bound (d2) solely. Hence, a range query is likely to fetch a larger part of the tree index than a nearest neighbors query does.

5.3. Continuous-nearest Neighbors Search (C-NNs search)

Three parameters are involved in the processing of a continuous nearest neighbors query: the number of nearest neighbors (k), number of sites of the dataset (N) and the query length ratio (QLR: the ratio of the length of the query line segment to the width of the search space).

Figure 11 shows the response time as a function of the QLR for dataset D25000 and (k=5). It affirms an interesting improvement of VNR-tree over R*-tree. The improvement factor between VNR-tree and RST-1/4k ranges between 3 (for QLR= 0.01) and 6 (for QLR= 0.25).

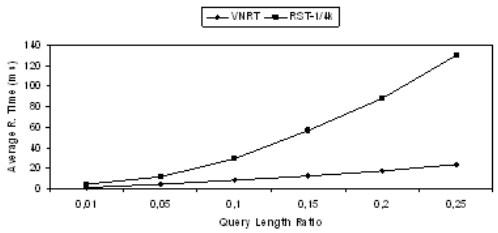

Figure 11. Response time =f(QLR) (Dataset D25000 and (k=5)).

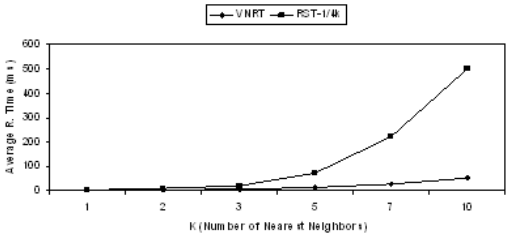

Figure 12. Response time =f(k) (Dataset D25000 and (QLR=17.5%)).

Figure 12 shows the response time as a function of the number of nearest neighbors (k) for dataset D25000 and (QLR=17.5%). It affirms an outstanding improvement of VNR-tree over R*-tree. The improvement factor between VNR-tree and RST-1/4k ranges between 2 (for k=1) and almost 10 (for k=10).

Figure 13 shows the response time as a function of the dataset size for (k=5) and (QLR=17.5%). It affirms that VNR-tree is more scalable: its response time grows noticeably more slowly with the increasing of the dataset size.

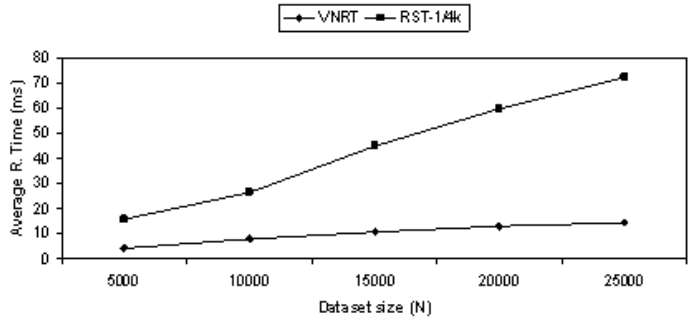

Figure 13. Response time =f(N) for (k=5) and (QLR=0.1).

6. Conclusion

This paper tackles the problem of enhancing the processing of location dependent queries (mainly, nearest neighbors, continuous nearest neighbors and distance range queries) based on indexing techniques. Most important existing indexes, like R-trees, adopt partitioning schemes with unavoidable overlapping and support backtracked search algorithms. Both features make search time heavy for an in-memory usage of the index. They also make such indexes impractical for a periodic broadcast data access mode within wireless environments like location based services.

We propose a new tree index, VNR-tree, constructed based on a non-overlapping binary partitioning scheme. VNR-tree supports, through the linking of its leaf nodes, novel backtracking-free search algorithms for the processing of several types of spatial queries based on the exploration of the mobile client's neighborhoods.

Experimental evaluation targets the use of VNR-tree as an in-memory index in the on-demand data access mode. It compares our proposal to R*-tree. Results confirm interesting performances for VNR-tree. For the processing of distance range queries, a prevailing of VNR-tree over R*-tree is guaranteed in the local neighborhoods of the query point. Increasing the dataset density tends to shrink the range of VNR-tree prevailing. For the processing of nearest neighbors queries, VNR- tree outperforms R*-tree substantially. This prevailing is insensitive to the dataset density.

In the broadcast data access mode, an important main feature of a good index is to be non-backtracked in order to fit the sequential access of the broadcast channel. So, evaluating the VNR-tree as an air-index is a straightforward extension of this work. Work remains for a variety of other directions. In fact, the proposed neighborhoods scanning algorithm over a Voronoi diagram tessellation seems to be very suitable to window queries, directional queries and, in general, all queries that need processing of spatial data situated within the neighborhoods of a query point.

References

[1] K. Gratsias, E. Frentzos, E. Dellis, and Y. Theodoridis, "Towards a Taxonomy of Location Based Services", Proceedings of the 5th International Workshop on Web and Wireless Geographical Information Systems, 2005.

[2] T. Imielinski, S. Viswanathan, and B.R. Badrinath, "Data on Air: Organization and Access", Journal of IEEE TKDE, 1997, vol. 9, pp. 353-372.

[3] S. Hwang, K. Kwon, S. Cha, and B. Lee, "Performance Evaluation of Main-Memory R-tree Variants", Proceedings of the 8th international symposium on Advances in Spatial and Temporal Databases (sstd'03), 2003, pp. 10-27.

[4] B. Zheng, W.C. Lee, and D.L.Lee, "Spatial Index on Air", Proceedings of the international conference on Pervasive Computing and Communications (PerCom 2003), 2003, pp. 297-304.

[5] N. Beckmann, H.P. Kriegel, R. Schneider, and B. Seeger, "The R*-tree: an Efficient and Robust Access Method for Points and Rectangles", Proceedings of the ACM SIGMOD International Conference, 1990, pp. 322-331.

[6] M. Berg, O. Cheong, M. Kreveld, and M. Overmars, Computational Geometry: Algorithms and Applications.2nd Ed. Springer-Verlag.

[7] A. Guttman, "R-trees: A dynamic Index Structure for Spatial Searching", Proceedings of the ACM SIGMOD Conference on Management of Data, 1984, pp. 47-54.

[8] M.Y. Eltabakh, R. Eltarras and, W.G. Aref, "Space-Partitioning Trees in PostgreSQL: Realization and Performance", Proceedings of the ICDE '06, 2006, pp. 526-531.

[9] R.K.V. Kothuri, S. Ravada, and D. Abugov. "Quadtree and R-tree Indexes in Oracle Spatial: a Comparison Using GIS Data", Proceedings of the ACM SIGMOD'02, 2002, pp. 546 – 557.

[10] Y. Manolopoulos, A. Nanopoulos, A.N. Papadpoulos and Y. Theodoridis, "R-trees have grown everywhere", Available from: http://www.rtreeportal.org/pubs/MNPT03.pdf (2005, accessed 2011).

[11] J.K.P. Kuan and, P. Lewis, "Fast k Nearest Neighbors Search for R-tree Family", Proceedings of the First International Conf. on Information, Communications and Signal Processing, 1997, pp. 924-928.

[12] A. Papadopoulos and, Y. Manolopoulos. "Performance of Nearest Neighbor Queries in R-trees", Proceedings of the 6th ICDT, 1997, pp. 394-408.

[13] G.R. Hjaltason and, H. Samet, "Distance Browsing in Spatial Databases", Journal of ACM TODS, 1999, vol. 24, pp. 265-318.

[14] M. Sharifzadeh and, C. Shahabi, "VoR-tree: R-trees with Voronoi Diagrams for Efficient Processing of Spatial Nearest Neighbors Queries", Journal of PVLDB 2010, vol. 3, pp. 1231-1241.

[15] Y. Tao, D. Papadias and Q. Shen, "Continuous nearest neighbor search", Proceedings of the 28th international conference on Very Large Data Bases (VLDB '02), 2002, pp. 287-298.

[16] B. Zheng, J. Xu, W.C. Lee and D.L. Lee, "Grid-Partition Index: A Hybrid Approach to Nearest-Neighbor Queries in Wireless Location-Based Services", VLDB Journal 2006, vol. 15, pp. 21-39.

[17] F. Li, "Real Datasets for Spatial Databases: Road Networks and Points of Interest", Available from: http://www.cs.fsu.edu/~lifeifei/SpatialDataset.htm (2005, accessed 2011).

Authors

Hassenet Slimani is an assistant at the University of Jendouba (Tunisia). She received her Engineer degree and Master degree in computer science from the Faculty of Sciences (University of Tunis-Elmanar/Tunisia) in 2003 and 2005, respectively. Currently, she is pursuing a phD in query processing within mobile environments.

Faïza Najjar is an assistant professor at the National School of Computer Science and Engineering in Manouba (Tunisia). She received her PhD in computer science from the University of Tunis ElManar, Tunisia, in June 1999. In 2003, she conducted a post-doctorate work on mobile computing and databases in the department of computer science and engineering at SMU (Dallas-TX). Najjar's current research interests are in pervasive and mobile computing and especially location based services

Yahya Slimani studied at the Computer Science Institute of Alger's (Algeria) from 1968 to 1973. He received the B.Sc.(Eng.), Dr Eng and Ph.D degrees from the Computer Science Institute of Alger's (Algeria), University of Lille (France) and University of Oran (Algeria), in 1973, 1986 and 1993, respectively. He is currently Full Professor at the Department of Computer Science of Faculty of Sciences of Tunis. His research activities concern Datamining, parallelism, distributed systems and Grid Computing. Dr. Yahya Slimani has published more than 100 papers from 1986 to 2009. He contributed to Parallel and Distributed Computing Handbook, Mc Graw-Hill, 1996. He is currently Scientific Expert for the European Union. He joined the Editorial Boards of the Information International Journal in 2000, the J.UCS Journal and others journals.