Monitoring Continuous *k*-Nearest Neighbor Queries using *c*-MBR

HaRim Jung° Sang-Won Kang MoonBae Song SeokJin Im Jongwan Kim Chong-Sun Hwang
Department of Computer Science and Engineering, Korea University
{harim°, swkang, mbsong, seokjin, wany, hwang}@disys.korea.ac.kr

Abstract

This paper addresses the problem of monitoring continuous *k*-nearest neighbor (*k*-NN) queries. Given a set of moving (or static) objects and a set of moving (or static) query points, monitoring continuous *k*-NN query retrieves and updates the closest *k* objects to a query point continually. In order to support location based services (LBSs) in highly dynamic environments, where objects and/or queries are frequently moving, monitoring continuous queries require real-time updated results when objects and/or queries change their locations. Thus, it is important to minimize time delay for maintaining up to date the results. In this paper, we present monitoring method to shorten time delay for updating continuous *k*-NN queries based on the notion of result region and the minimum bounding rectangle enclosing all objects in each cell, referred to as *c*-MBR, in the grid index structure. Simulations are conducted to show the efficiency of the proposed method.

1. Introduction

The advance in wireless communication and widespread mobile devices together with positioning technologies like global positioning systems (GPS) have enabled various location based services (LBSs). Such services entail frequent updates of query results due to highly dynamic nature of both objects and queries. One of the most fundamental issues for supporting LBSs is to monitor and provide real-time results to various location dependent queries such as range queries and *k*-nearest neighbor queries.

Recently, the problem of monitoring continuous location dependent queries has been investigated in [7,8,9,10]. The main focus of such problem is how to minimize time delay for keeping up to date the results under highly dynamic environments where both objects and queries are continuously moving with unpredictable manner.

This paper addresses the problem of monitoring continuous *k*-nearest neighbor (*k*-NN) queries. Given a set of moving (or static) objects and a set of moving (or static) query points, monitoring continuous *k*-NN query continually updates the closest *k* objects to each query point during the *running time of each query. Since continuous queries have long running* time, monitoring continuous queries focuses more on continuous maintenance of the results than initial result computation for each query. Thus, we are not concerned with initial *k*-NN evaluation and we assume either initial result of each query is available or initial result is evaluated by methods proposed in [8].

Since a large set of queries and objects update their location asynchronously, *it is inefficient to update the index and re-evaluate each query* results whenever any query and/or object moves [5]. Thus, we periodically update the index and re-evaluate all query results affected by the motion of objects at fixed time interval Δt. In order to provide real-time updated results for continuous *k*-NN query, it is critical to minimize Δt.

In this paper, we use the notion of result region and minimum bounding rectangle (MBR) to shorten CPU-time for maintaining up to date query results, thus reduce Δt.

The result region of the continuous *k*-NN query is the circular region centered at the query point q with radius *k*th_dist, where *k*th_dist is the distance between the q and *k*th-NN. The result region of q guarantees that only if location updates of moving objects occur in that region, the result of q can be affected. With the use of the result region, we do not need to consider location updates of moving objects outside the result region.

We use the in-memory index structure that takes the form of grid consisting of cells with size $\delta \times \delta$ to support fast maintenance of frequent location updates from moving objects. Each cell c in the grid G maintains the list of objects that fall therein and additional information on MBR, hereafter referred to as *c*-MBR, that completely encloses all objects within *c*. *c*-MBR enables us to reduce unnecessary visits of cells when re-evaluating query results. Based on the result region and c-MBR, we present Algorithms for index maintenance and query re-evaluation

The rest of this paper is organized as follows. Related Work for monitoring continuous *k*-NN queries are discussed in Section 2. In Section 3, we presents our technique covering index structure, incremental query update algorithm, and optimizations. Section 4 presents the performance evaluation. Finally we conclude the paper in Section 5.

2. Related Work

Answering *k*-NN queries is one of the most important problems in computer science especially in database community. Traditional algorithms for answering *k*-NN queries [2,3,4] assume a set of static objects indexed with R-tree-like structures [1] and efficiently evaluate query results to each static query by utilizing some pruning heuristics. Motivated by location based services (LBSs), various techniques that consider answering *k*-NN queries under dynamic environments, where objects and/or queries are constantly moving have been proposed [5,6]. These techniques utilize spatial or temporal validity information for answering continuous *k*-NN queries. These techniques do not assume that queries run continually over the database and focus on efficient query re-evaluation for a single snapshot query.

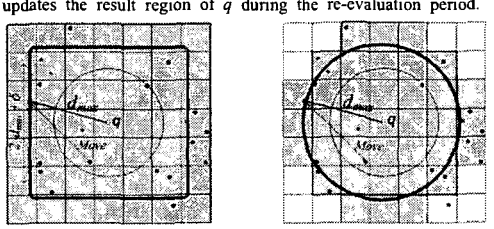
Recently, three methods for monitoring continuous *k*-NN queries were proposed in [8,9,10]. All the three methods focus on maintaining query results for multiple long running queries in a CPU-time-efficient way. They use a grid-like index structure for the fast maintenance. Furthermore, [9,10] take the similar approach by maintaining result region for each query.

Yu et al. [8] propose two main-memory indexing approaches with different scenarios for monitoring exact continuous *k*-NN queries over moving objects: object indexing and query indexing. With object indexing approach, overhaul query re-evaluation and incremental query re-evaluation are suggested. The overhaul query re-evaluation periodically re-builds the *index from the scratch and re-evaluates new k-NNs.* The incremental query re-evaluation, denoted as YPK-CNN, updates index incrementally and utilizes previous *k*-NNs for updating query results. In particular, to avoid unnecessary distance computation, the search scope for the new query result is determined. As illustrated in Figure $1_{(a)}$, the search scope is the approximated rectangular region centered at c_q with side length $2 \cdot d_{max} + \delta$ (c_q is the cell containing the query point q, d_{max} is the distance from query point q to the previous NN that *is currently furthest,* and δ *is the side length* of the cell *c*). The new *k*-NNs of q are found by accessing all cells that intersect with the search scope(shaded cells in Figure $1_{(a)}$). Query indexing approach indexes query points instead of objects. Authors also introduce hierarchical extension of object indexing approach for improving performance when objects are not uniformly distributed.

Xiong et al. [9] propose SEA-CNN. SEA-CNN assumes that objects are indexed with grid structure in secondary-memory and only focuses on incrementally maintaining the query results without considering the evaluation of initial *k*-NN evaluation. In order to monitor continuous *k*-NN queries incrementally, SEA-CNN performs the following three steps:

1. If any of the previous NNs of each query q moves within the result region or any of the outer objects moves into the result region, The search scope is set to the result region of q.
2. If any of the previous NNs of each query q moves out the result region, the search scope is updated to be the circular region centered at q with the radius d_{max}, where d_{max} is the distance from query point q to the previous NN that is currently furthest.
3. Finally, If each query q changes its location, SEA-CNN updates the search scope to be the circular region centered at q_{new} with the radius *sum of* d_{max} *in step 2 and the distance from* q_{new} to q_{old}, where q_{new} is the new location of q and q_{old} is the previous location of q.

After the above three steps, SEA-CNN retrieves the new query result of each query q by visiting all cells that intersect with the search scope and computing distance between q and every object o that is inside the cells. Then, updates the result region of q during the re-evaluation period.

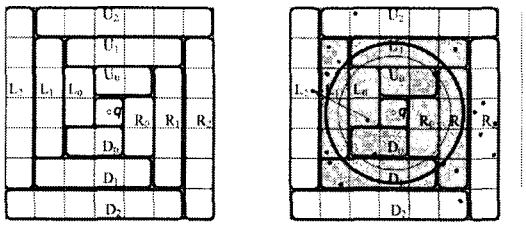


(a) Search scope of YPK-CNN (k=3) (b) Search scope of SEA-CNN (k=3)

Figure 1. Search scopes of YPK-CNN and SEA-CNN

CPM-CNN proposed in [10] conceptually partitions the grid G into directional rectangles(*DIR*s) for each query q. Figure $2_{(a)}$ illustrates the conceptually partitioned grid space around the cell c_q, where c_q is the cell containing the query point q. Each *DIR* is defined by (1)its direction(U, D, L, R) with relative position with respect to the c_q and (2)the level that

indicates the number of *DIR*s between the *DIR* itself and c_q. CPM-CNN gives MINDIST ordering to each directional rectangle *DIR* or cell c and computes the initial k-NNs. In particular, CPM-CNN initializes a priority queue and enqueues the entries(corresponding either to c or *DIR*) with MINDIST(c, q) or MINDIST(*DIR*, q) as a key. Then, CPM-CNN dequeues the entries iteratively. If dequeued entry is a cell, CPM-CNN examines all objects inside the cell and updates the kNNs if necessary. If dequeued entry is a *DIR*, CPM-CNN enqueues all cells the *DIR* includes and the next level *DIR* with the same direction. The initial NN computation algorithm terminates when the next entry in the priority queue has the key greater than or equal to the distance of kth NN to q. In order to maintain the result region of q and to reduce search scope during the re-evaluation period, CPM-CNN maintains book-keeping information in the grid index G and additional structure. Figure $2_{(b)}$ illustrates the search scope of CPM-CNN. CPM-CNN enqueues all shaded cells(including *DIR*s) in to the priority queue and updates the result.



(a) Conceptual partitioning (b) Search scope of CPM-CNN (k=3)

Figure 2. Conceptual partitioning and search scope of CPM-CNN

The search scope of re-evaluation of all the aforementioned three methods can be approximated with collections of cells in the grid. Every cell c in the approximated search scope should be visited, even if it does not contain the result, in order to guarantee the correct result. If the number of visited cells is reduced, needless object-distance computations can be avoided. As a result, Δt also be shortened. To reduce the number of visited cells, we propose continuous k-NN monitoring method based on result region and c-MBR.

3. Monitoring Continuous k-NN Query using c-MBR

Like existing methods mentioned in Section 2 [8,9,10], we assume two-dimensional data space and use grid index for simple and fast maintenance under dynamic environment. The grid index G is two-dimensional array of cells with its each side length is δ Each cell ,denoted as c_{ij} at column i and row j(starting from the low-left corner), contains every object with its coordinates (obj_x, obj_y), where $\left\lfloor \frac{x}{\delta} \right\rfloor = i$ and $\left\lfloor \frac{y}{\delta} \right\rfloor = j$. Since what is important in monitoring continuous k-NN queries is to continually maintain k-NNs over time and to minimize time delay for re-evaluation, we do not deal with initial k-NN evaluation.

3.1 Preliminaries

In this section, we give definitions of c-MBR and distance measures that are necessary for the description of our algorithm for monitoring continuous k-NN queries. Data structure for our method is also presented.

Definition 1: Given a cell c_{ij} and objects $Obj=\{obj_i,\ 1\le i\le n\}\in c_{ij}$ with their coordinates (obj_x, obj_y), c-MBR$_{ij}$ is a two-dimensional minimum bounding rectangle specified by two endpoints of its major diagonal $min\_point=(x_{min}, y_{min})$ and $max\_point=(x_{max}, y_{max})$, where $x_{min}=min(\forall obj_x)$, $y_{min}=min(\forall obj_y)$, $x_{max}=max(\forall obj_x)$, $y_{max}=max(\forall obj_y)$ for $\forall(obj_x)$ and $\forall(obj_y)$ of *Obj*.

If c_{ij} has no object, we set (x_{min}, y_{min}) to (∞, ∞) and (x_{max}, y_{max}) to ($-\infty$, $-\infty$).

Definition 2: Minimum distance from a cell c_{ij} to a query point q with its coordinates (x, y), denoted as MINDIST(c_{ij}, q) is:

$$\text{MINDIST}(c_{ij},\ q) = \sqrt{|x-c_x|^2+|y-c_y|^2}$$

where $c_x = \begin{cases} i & \text{if } x<i \\ i+\delta & \text{if } x>i+\delta \\ x & \text{if } i\le x\le i+\delta \end{cases}$, $c_y = \begin{cases} j & \text{if } y<j \\ j+\delta & \text{if } y>j+\delta \\ y & \text{if } j\le y\le j+\delta \end{cases}$

Definition 3: Minimum distance from a c-MBR$_{ij}\subset c_{ij}$ to a query point q with its coordinates (x, y), denoted as MINDIST(c-MBR$_{ij}$, q) is:

$$\text{MINDIST}(c\text{-MBR}_{ij},\ q) = \sqrt{|x-cr_x|^2+|y-cr_y|^2}$$

where $cr_x = \begin{cases} x_{min} & \text{if } x<x_{min} \\ x_{max} & \text{if } x>x_{max} \\ x & \text{if } x_{min}\le x\le x_{max} \end{cases}$, $cr_y = \begin{cases} y_{min} & \text{if } y<y_{min} \\ y_{max} & \text{if } y>y_{max} \\ y & \text{if } y_{min}\le y\le y_{max} \end{cases}$

Above three definitions imply the following condition:

- MINDIST(c_{ij}, q)≤MINDIST(c-MBR$_{ij}$, q)≤MINDIST($Obj\in c_{ij}$, q)

3.2 Data Structure and Query Re-evaluation

Each cell c_{ij} in the grid G maintains its c-MBR$_{ij}$ and the list of objects within its extents. In addition, c_{ij} has the lists of queries whose result regions intersect with the c_{ij}.

Query table QT stores the queries that are currently running with their ids, coordinates, $k$th_dist, and current kNNs. By using $k$th_dist, we determine the result region of each query q(i.e., the circular region centered at the query point q with radius $k$th_dist). Figure 3 illustrates the index structure.

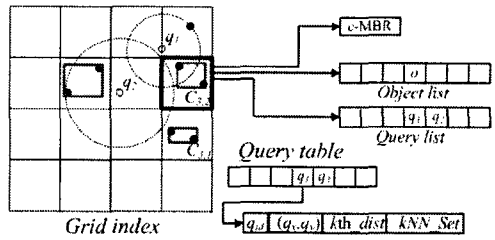


Figure 3. Grid index structure and query table

We assume that the initial result of each query q are available. By using the initial result of each q, we maintain the index and re-evaluate the result of q when location updates of objects occur. In particular, for each cell c_{ij} in G, we takes the following steps:

1. Check if any of the inner objects *obj* moves out of c_{ij}. If so, *obj* is deleted from the object list of c_{ij}. c-MBR$_{ij}$ is changed only if the previous coordinates (obj_x, obj_y) of *obj* was either *min_point* or *max_point* of c-MBR$_{ij}$. Each q in the query list of c_{ij} is checked whether q contains *obj* in its *kNN_set*. In case *obj* is in *kNN_set* of q and *obj* moves out from the result region of q, *obj* is deleted from *kNN_set* of q ,$k$th_dist is set to *dist*(q,*obj*) and q is marked as affected.
2. Check if any of the inner objects *obj* moves inside c_{ij}. If so, c-MBR$_{ij}$ is checked and updated if possible. Specifically, for current coordinates(obj_x, obj_y) of *obj*,
   - if $obj_x<x_{min}$ of c-MBR$_{ij}$ then x_{min} is set to obj_x.
   - if $obj_y<y_{min}$ of c-MBR$_{ij}$ then y_{min} is set to obj_y.
   - if $obj_x>x_{max}$ of c-MBR$_{ij}$ then x_{max} is set to obj_x.
   - if $obj_y>y_{max}$ of c-MBR$_{ij}$ then y_{max} is set to obj_y.

   Otherwise, c-MBR$_{ij}$ remains unchanged. Each q in the query list of c_{ij} also has to be checked. In particular, for each q in the query list of c_{ij}.
   - if q is marked as affected, q is ignored.
   - if *obj* is in *kNN_set* of q and *obj* moves out from the result region of q, *obj* is deleted from *kNN_set* of q ,$k$th_dist is set to *dist*(q,*obj*) and q is marked as affected.
   - if *obj* is not in *kNN_set* of q and *obj* moves into the result region of q, kth-NN of q is deleted from *kNN_set* and *obj* is inserted into *kNN_set*. Then the order of *kNN_set* and the result region of q are updated.
3. Check if any of the outer objects *obj* moves into c_{ij}. If so, *obj* is inserted into the object list of c_{ij}. c-MBR$_{ij}$ is checked and updated as done in step 2. Then, every q that is not marked as affected in the query list of c_{ij} is checked. If *obj* moves into the result region of q from step 2, the kth-NN of q is deleted from *kNN_set* and *obj* is inserted into *kNN_set*. Then the order of *kNN_set* the result region of q are updated.

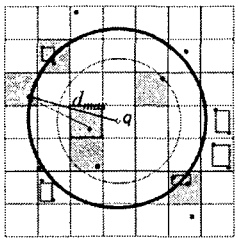
After above steps, We re-evaluate each affected query q_{aff}. We use priority queue and MINDIST(c-MBR$_{ij}$, q) metric for efficient re-evaluation. Figure 4 illustrates the re-evaluation algorithm to retrieve the new *kNN_set* of q_{aff}.

Algorithm 1: re-evaluate the result of affected query
//Input = G: the grid index, q_{aff}: the affected query
procedure:
1. initialize the priority queue
2. **for** each cell c_{ij} that intersects with result region of q_{aff} **do**
3. **if** MINDIST(c-MBR$_{ij}$, q_{aff})≤q_{aff}.$k$th_dist **then**
4. enqueue every cell entry [c_{ij}, MINDIST(c-MBR$_{ij}$, q_{aff})]
5. **while** entry has key<q_{aff}.$k$th_dist and priority queue is not empty **do**
6. dequeue the next entry of the queue
7. **for** each *obj* c_{ij} in the cell entry **do**
8. **if** dist(obj, q_{aff})<q_{aff}.$k$th_dist **then**

9.	update q_{aff}.*kNN_set* and q_{aff}.kth_dist

Figure 4. Re-evaluation algorithm for affected query

As shown in Figure 5, when re-evaluating the affected queries, we enqueue only shaded cells into the priority queue. Thus, we reduce the visited cells as well as heap operation used for insertion/deletion operation of priority queue by using MINDIST(*c*-MBR$_{i,j}$, *q*) metric instead of MINDIST($c_{i,j}$, *q*) metric.



Search scope of *c*-MBR based monitoring (*k*=3)

Figure 5. Search scope of proposed method

3.3 Improved Query Re-evaluation

Although query re-evaluation algorithm discussed in Section 3.2 reduce the search scope of *q* and the number of visited cells, thus leads to shorten each re-evaluation time interval Δt, this can be improved by dealing with multiple location updates of objects [10]. Let Up_{obj} be the set of location updates of objects. Clearly, if the result region of *q* contains at least *k* objects after Up_{obj}, we can simply re-order the objects that fall in the result circle and form the new result of *q* without using re-evaluation algorithm.

For this purpose, each *q* maintains additional counter, sorted list for incoming objects and sorted list for outgoing objects during the re-evaluation. Only if the number of outgoing objects Num_{out} is greater than the sum of the number of incoming objects Num_{in} and the number of remaining objects Num_{rem}, we re-evaluate *q* by using Algorithm 1 presented in Section 3.2. However, in order to reduce the search scope, we set *q*.kth_dist to distance between the query point *q* and {Num_{out} − (Num_{in} + Num_{rem})}th element of *q.out_list*. Figure 6 illustrates the complete query re-evaluation algorithm. After algorithm termination, we update the result region of each *q*.

Algorithm 2: re-evaluate the result of queries
(Input = *G*: the grid index, *QT*: query table, *Up*: set of location updates)
procedure:
1. **for** each *q* in *QT* **do**
2. set *q.count*=0
3. initialize a sorted list *q.in_list* and *q.out_list* of size *k*
4. **for** each cell c_{ij} in *G* **do**
5. **if** c_{ij} has up_{obj} Up_{obj} **then**
6. **for** each up_{obj} Up_{obj} in *c* **do**
7. **for** each *q* in the *query_list* of *c* **do**
8. **if** *obj* *q.kNN_set* and *dist*(*q*,*obj*)≤*q*.kth_dist **then**
9. re-order *q.kNN_set* based on distance to *q*
10. update *q*.kth_dist if necessary
11. **else if** *obj* *q.kNN_set* and *dist*(*q*,*obj*)>*q*.kth_dist **then**
12. delete *obj* from *q.kNN_set*
13. insert *obj* into the *q.out_list*
14. *q.count* = *q.count* + 1
15. **else if** *obj* ∉ *q.kNN* and *dist*(*q*,*obj*)≤*q*.kth_dist **then**
16. *q.count* = *q.count* - 1
17. insert *obj* into the *q.list*
18. **for** each *q* in *QT* **do**
19. **if** *q.count*≤0 **then**
20. *q.kNN_set*= the best *k* objects in *q.list* *q.kNN_set*
21. update *q*.kth_dist and the result region of *q*
22. **else if** *q.count*>0 **then**
23. set *q*.kth_dist to (|Obj_{out}|-|Obj_{in}|)th element of *q.out_list*
24. call **Algorithm 1** in Figure 3.2

Figure 6. Re-evaluation algorithm for monitoring continuous *k*-NN queries

Finally, in case query *q* updates its location, we can either 1) maintain query results incrementally similar to the method proposed in [9] or 2)compute the new results of queries by using initial *k*-NN algorithm proposed in [8,10].

4 Performance Evaluation

In this section, we investigate the effectiveness of proposed method with respect to 1)the number of objects, 2)the number of queries. All the experiments are performed on Intel Pentium IV 2.4GHz with 512GB RAM.

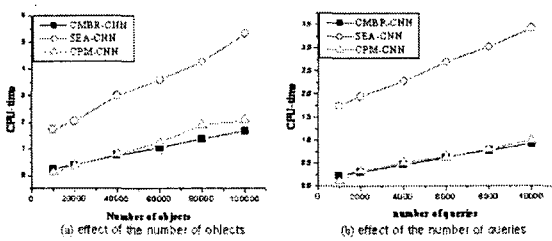
We use data set that randomly move and assume that objects and queries are not disappeared during the experiments. We set *k* to 10 in all the experiments. The granularity of the grid index G is set to 64 ×64. We compare the CPU-time of our method with SEA-CNN [9] and CPM-CNN [10]. As Table 1 shows, the number of objects is varied from 10(K) to 100(K) and the number of queries is varied from 1(K) to 10(K) in the experiments.

Table 1. Parameter settings

Parameter	Range
Number of objects	10, 20, 40, 60, 80, 100 (K)
Number of queries	1, 2, 4, 6, 8, 10 (K)

As shown in Figure 7$_{(a)}$, CPU-time of all the three method is increased linearly to the number of objects. However, our method and CPM-CNN are less sensitive to the number of objects. Although the performance of CPM-CNN is better than our method with small number of objects, When the number of objects is increased, our method shows better performance.

Similar to the effect of the number of objects, Although CPU-time of all the three method is increased linearly to the number of queries as illustrated in Figure 7$_{(b)}$, our method shows the best performance as the number of queries is increased.



(a) effect of the number of objects (b) effect of the number of queries

Figure 7. The effects of the number of objects/queries on CPU-time

5 Conclusion and Future work

In this paper, we investigates the problem of monitoring continuous *k*-nearest neighbor(*k*-NN) queries. Based on the notion of result region and *c*-MBR, we present *c*-MBR based in-memory grid structure and query re-evaluation algorithm. With the use of MINDIST(*c*-MBR, *q*), we reduce unnecessary computation for updating query result.

As a future work, we will devise lazy update *c*-MBR scheme to reduce the cost for updating *c*-MBR. Since no efficient incremental monitoring method for the movement of queries, we will devise efficient incremental monitoring method with respect to the motion of queries. We will also extend our method to handle other variations of *k*-NN queries such as reverse *k*-NN.

References

[1] A.Guttman, "R-trees: a dynamic index structure for spatial searching", In SIGMOD, 1984.
[2] N.Roussopoulos, S.Kelley, and F. Vincent, "Nearest neighbor search", In SIGMOD, 1995.
[3] K. L. Cheung and A. W. C. Fu, "Enhanced nearest neighbor search on the R-tree", In SIGMOD, 1998.
[4] Gisli R. Hjaltason, Hanan Samet, "Distance Browsing in Spatial Databases", ACM Transactions on Database Systems, 1999.
[5] Y. Tao and D. Papadias, "Time-parameterized qieries in spatio-temporal databases", In SIGMOD, 2002.
[6] J. Zhang, M. Zhu, D. Papadias, Y. Tao, and D. Lee, "Location-based spatial queries", In SIGMOD, 2003.
[7] Dmitri V. Kalashnikov, Sunil Prabhakar, Susanne E. Hambrusch, "Main Memory Evaluation of Monitoring Queries Over Moving Objects", Distrib. Parallel Databases, 15(2):117-135, 2004
[8] Yu, X., Pu, K., Koudas, N, "Monitoring K-Nearest Neighbor Queries Over Moving Objects", In ICDE, 2005.
[9] Xiong, X., Mokbel, M., Aref, W, "SEA-CNN: Scalable Processing of Continuous K-Nearest Neighbor Queries in Spatio-temporal Databases", In ICDE, 2005.
[10] K. Mouratidis, M. Hadjieleftheriou,and D. Papadias, "Conceptual Partitioning: An Efficient Method for Continuous Nearest Neighbor Monitoring", In SIGMOD, 2005.