Front. Comput. Sci., 2013, 7(1): 44–54
DOI 10.1007/s11704-012-2069-z

RESEARCH ARTICLE

VGQ-Vor: extending virtual grid quadtree with Voronoi diagram for mobile *k* nearest neighbor queries over mobile objects

Botao WANG (✉)[1], Jingwei QU[1], Xiaosong WANG[1], Guoren WANG[1], Masaru KITSUREGAWA[2]

1 Department of Information Science and Engineering, Northeastern University, Shenyang 110004, China
2 Institute of Industrial Science, the University of Tokyo, Tokyo 153-8505, Japan

Abstract Performing mobile *k* nearest neighbor (MkNN) queries whilst also being mobile is a challenging problem. All the mobile objects issuing queries and/or being queried are mobile. The performance of this kind of query relies heavily on the maintenance of the current locations of the objects. The index used for mobile objects must support efficient update operations and efficient query handling. This study aims to improve the performance of the MkNN queries while reducing update costs. Our approach is based on an observation that the frequency of one region changing between being occupied or not by mobile objects is much lower than the frequency of the position changes reported by the mobile objects. We first propose an virtual grid quadtree with Voronoi diagram (VGQ-Vor), which is a two-layer index structure that indexes regions occupied by mobile objects in a quadtree and builds a Voronoi diagram of the regions. Then we propose a moving *k* nearest neighbor (kNN) query algorithm on the VGQ-Vor and prove the correctness of the algorithm. The experimental results show that the VGQ-Vor outperforms the existing techniques (Bx-tree, Bdual-tree) by one to three orders of magnitude in most cases.

Keywords location based services, mobile *k* nearest neighbor query, mobile object index, Voronoi diagram

1 Introduction

Location based services (LBSs) provide personalized services to users of mobile devices based on their up-to-date locations. With the development of wireless communications and positioning technologies, LBSs are becoming increasingly important due to numerous emerging applications (such as, location-based search, vehicle tracking, traffic control, and accessing), where the services are provided in different forms of spatio-temporal queries. One of the common queries is *k* nearest neighbors (kNNs) query, which searches for the *k* data objects that lie closest to a given query point. Because both data object and query may be mobile, such queries rely heavily on the maintenance of the current locations of the mobile objects. The challenges in this type of query are efficient indexing and query processing techniques on the mobile objects.

In this paper, we focus on mobile *k* nearest neighbor (MkNN) queries, we provide a more general specification of this type of query whereby both source and target of a query may be mobile. One example is finding the three nearest companions when a tourist moves with a guided tour. Another example is finding the two nearest medics in the battlefield. Traditionally, research on kNN queries has focused on static objects [1–3], these approaches usually utilize index structures on static objects. Most of the work on static kNN query processing over moving objects is either static kNN queries over moving objects [4,5] or moving queries over static objects [6–10]. Some work has addressed the problem of MkNN [11–14]. At the same time, MkNN queries can be implemented based on mobile object indices, such as, TPR-tree [15], TPR*-tree [16], STRIPES [17], RUM-tree [18], Bx-tree [19], Bdual-tree [20], and ST^{2}B-tree [21], where different un-

Received March 6, 2012; accepted May 28, 2012

E-mail: wangbotao@ise.neu.edu.cn

derlying data structures (like R-tree and B+-tree) and strategies to reduce the cost of update operations are used. Many of them [8–10] utilize Voronoi diagrams [22] to enhance the kNN query processing because Voronoi diagrams are efficient data structures for exploring a local neighborhood in a geometric space. In these prior works, a single Voronoi diagram is built on static data objects.

The goal of this paper is to utilize the properties of Voronoi diagrams for efficient processing of MkNN queries. It is expensive to utilize Voronoi diagrams for MkNN queries directly. This is because the number of mobile objects is large and the frequency of update operations derived from position changes is high. Based on an observation that the frequency of region occupation change by a mobile object is much lower than the frequency of the position changes reported by the mobile objects, the main idea of our solution is to partition the space of LBS applications into grid regions and build a Voronoi diagram on the occupied regions rather than the individual objects. Further, we incorporate the Voronoi diagram into a virutal grid quadtree (VGQ) (Our previous work on VGQ is reported in [23]). The resulting data structure, called VGQ-Vor, is a VGQ index enriched by the Voronoi diagram of the grid regions stored in a VGQ.

The main contributions of this paper are: 1) we propose a data structure VGQ-Vor to support efficient MkNN queries; 2) we propose an MkNN query algorithm and prove the correctness of the algorithm; 3) we perform an experimental evaluation comparing our solution to the existing works.

Section 2 introduces related work. Section 3 describes Voronoi diagrams and VGQ in brief. Section 4 presents the architecture of VGQ-Vor, the MkNN query algorithm, and the correctness proof of the algorithm. Section 5 reports experimental evaluation. Finally, we provide conclusions in Section 6.

2 Related work

Much work has been done on mobile object indexing and kNN query processing. Research on kNN query can be broadly divided into two categories: 1) index built on static objects; 2) index built on mobile objects. Note that a mobile object may be either the source or target of a query, or even both. Our work belongs to the second category.

2.1 kNN queries over indexes of static objects

Some work has been done with static kNN queries on moving objects [4,5] and index is built on queries. In [4], the queries are indexed by a grid structure held in memory. In [5], the queries are indexed by a binary partitioning tree.

Many works focus on kNN queries from mobile devices on static objects with indices built on static points and road network data. In [6], methods are proposed to reduce the cost of each query operation by using information of previous queries and pre-fetched results that are stored in an R-tree. In [7], an algorithm is proposed to find the kNNs for all positions by searching an R-tree only once. In [24], best-first network expansion (BNE) algorithm is proposed for monitoring k-path nearest neighbor (kPNN) queries, where an expansion tree and a candidate set are utilized for efficient kPNN updates and results. In [25], uncertain trajectories hierarchy (UTH) is proposed to process spatio-temporal range queries for uncertain trajectories on road networks. In UTH, for each edge, time periods are stored in an R-tree within which objects are moving.

Many techniques utilizing Voronoi diagrams have been proposed for kNN queries because it enables a very efficient search [22]. In [8], a first order Voronoi diagram is used in the processing of kNN queries in a spatial network. Here a large network is partitioned into small Voronoi regions and the distances are pre-computed. In [9], an incremental safe-region-based technique for moving kNN queries, called v*-diagram which exploits the current information of the query point and the search space in addition to data object. In [10], a new index structure, called Vor-tree, is proposed to use Voronoi diagrams with R-tree for efficient processing spatial nearest neighbor queries, there Voronoi cells are stored in R-tree.

All the above techniques are based on indices built on static queries or data, cannot be applied to MkNN queries directly.

2.2 kNN queries based on index built on mobile objects

In order to index mobile objects, many indices have been proposed, such as, TPR-tree [15], TPR*-tree [16], STRIPES [17], RUM-tree [26], Bx-tree [19], Bdual-tree [20], ST^{2}B-tree [21]. Different underlying data structures and strategies are used to reduce the cost of update operations on the indices. TPR-tree [15], TPR*-tree [16] use R-tree [27] structure with time-parameterized bounding boxes, there each bounding box has an associated velocity. TPR*-tree [16] outperforms the basic TPR-tree [15]. STRIPES [17] indexes predicted trajectories in a dual transformed space with a multi-dimensional PMR-quadtree as its underlying index. RUM-tree [26] is a memo-based approach to avoid disk accesses for purging old entries so as to minimize the cost of object updates on R-tree. Both Bx-tree [19] and Bdual-tree [20] use B+tree as their underlying index. In Bx-tree[19], moving

object position locations are represented as vectors that are timestamped based on their update time. The vectors are linearized with a space filling curve and the value is indexed by B+tree. Bdual-tree [20] extended Bx-tree by utilizing velocity information. In ST^2B-tree [21], the entire space is partitioned into regions of different different object density using a set of reference points and each region uses an individual grid file index with different cell sizes. Bx-tree is used for each region and a new set of reference points and new grid file index are chosen and rebuilt according to the latest data density. All the above indices support MkNN queries directly.

There also exist some works [11–14,28,29] to deal with MkNN queries. In [28], an analysis tool, the transformed minkowski sum (TMS) is introduced to determine the intersection of two moving objects of arbitrary shapes, which can be used to optimize range and kNN queries on mobile objects. Here mobile objects are indexed in TPR*-tree. In [14], a possibility-based vague kNN algorithm is proposed to process the query efficiently over the objects with uncertain velocity in road networks. The underlying index structure is build based on TPR-tree. In [29], a filter-and-refine strategy was proposed to find kNN on moving object trajectories which are indexed in a 3D-R-tree. In [11], two solutions (object-indexing and query-indexing) are proposed. In both methods, grid indices are used to index moving objects or moving queries to relax the assumption that the trajectories of the objects are fully predictable. In [13], an incremental monitoring algorithm is proposed to re-evaluate queries at a time when updates occur for road networks. The road network is index by PMR-quadtree and there is no update operation on the road network PMR-quadtree. In [12], an algorithm, called SEA-CNN is proposed to improve the performance of continuous kNN queries by incremental evaluation and shared execution, grid file is used as underlying index.

Orthogonal, but related to our work, a benchmark has been proposed for evaluating moving object index and kinds of mobile queries [30]. Our proposal is compared with the existing works using the data created in [30].

VGQ-Vor is different from the existing works in two aspects. First, Voronoi diagrams are used to support MkNN queries. Second, the underlying index is VGQ [23], where a mobile object index is built to index regions occupied by the mobile objects.

3 Preliminaries

In this section, we introduce Voronoi diagrams [22] and VGQ [23] in brief.

3.1 Voronoi diagram

A Voronoi diagram is a special kind of decomposition of a given space determined by the distances to a specified family of objects (subsets) in the space [22]. The Voronoi diagram of a given set of points $P=\{p_1, p_2, ..., p_n\}$ partitions the space into n regions. Each region corresponds a point p of P and each region consists of all points in the data space which are closer to p than any other points in P. Figure 1(a) shows an example Voronoi diagram for a set of points in a two-dimensional space.

Voronoi diagrams have many useful properties. Here, we review two of the basic geometric properties of Voronoi diagrams. These properties are relevant to the kNN query algorithm to be introduced in Section 4.

Property 1 The closet pair of points corresponds to two adjacent cells in the Voronoi diagram.

Property 2 The dual structure for a Voronoi diagram in a two-dimensional Euclidean space corresponds to a Delaunay triangulation of the same set of points.

Property 3 The average number of edges in the boundary of a Voronoi region is less than 6.

The Delaunay triangulation of a point set is a collection of edges. Figure 1(b) shows the Delaunay triangulation (in solid lines) corresponding to the Voronoi diagram (in dotted lines) shown in Fig. 1(a). The Delaunay triangulation satisfies the following properties relevant to the kNN query algorithm.

Property 4 The closest two points are connected by an edge of the Delaunay triangulation.

We utilize Property 1 and Property 4 in design and implementation of the kNN query algorithm. Property 3 is used to estimate the time and space needed to store the Voronoi diagram.

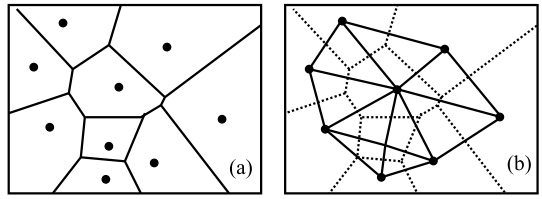

Fig. 1 Example of (a) Voronoi diagram and (b) Delauany triangulation

3.2 VGQ

We first presented VGQ in [23]. VGQ is an index of the regions occupied by mobile objects instead of the mobile ob-

jects themselves. It is designed based on the observation that the frequency of change between a region being occupied or empty is much lower than the frequency of position reports from the mobile objects. As an example, one can consider the traffic in a city; even though there are many cars moving at the same time, the regions covered by these cars do not change that dramatically. The number of updates to a region index will be much fewer than that of the mobile index. Consider a road intersection as a region with ten cars passing through. For that minute there will not be a status update as the junction has at least one car within its region. However those ten cars would register ten updates to a mobile update index as they individually change region. Consequently, the maintenance cost of the region index is lower than that of the mobile object index.

Figure 2 shows the data structures used in VGQ. As shown in the lower right corner of the figure ("Virtual grid file"), the entire space is divided into grid cells. Each grid cell has a unique identifier (CID) and corresponds one region. Each region clusters a set of mobile objects and a mobile object can only be within one region. There only Regions 1, 6, 10, 11 and 14 are occupied by mobile objects and only these regions are inserted into the index shown in the upper right corner (We use a Quadtree index). The object hash table stores the information of mobile objects, such as the identifier (OID), CID, and its coordinate. The caddress hash table stores the information of the occupied regions, such as the CID of a region, its coordinate (CXY), and the set of all the mobile objects inside the region (PLEAF). A bidirectional link is built between each region item of the caddress hash table and its corresponding leaf node of the Quadtree. The query data is stored in the query hash table, such as the identifier of a query (QID) and its results (QVALUE).

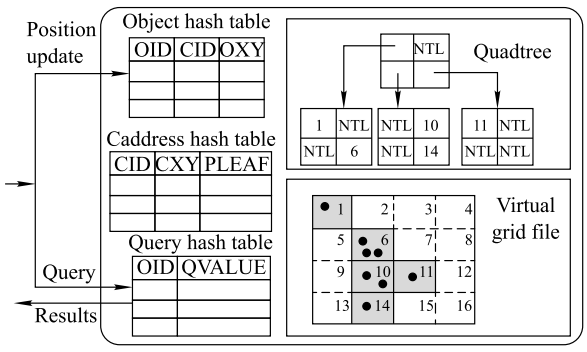

Fig. 2 Data structures used in VGQ

When a mobile object reports a position or issues a range query, the processes of update operation and query operation based on VGQ are described as follows:

- Update operation. There are three main steps. The first step is to get the previous and current regions occupied by the mobile object. The previous region is obtained by checking the object hash table and the caddress hash table, and the current region is obtained by mapping the coordinate of the mobile object to its corresponding grid cell (region). The second step is to delete the mobile object from the previous region and insert it into the current region. The final step is to maintain the Quadtree, if the occupancy status of the regions change. The previous region will be deleted from the Quadtree if the previous region becomes empty (not occupied by any mobile object) and the current region will be inserted into the Quadtree if the current region is not indexed by the Quadtree.
- Query operation. This is a two step process. The first setp is to obtain a set of regions that intersect the range by searching the Quadtree using a conventional quadtree range search algorithm. The second step is to refine the final results by range checking on the mobile objects inside the regions obtained in the previous step.

The grid file is not built physically, only the mapping relationship between mobile objects and regions is utilized, we call this the virtual grid file. Only the regions being occupied by mobile objects are indexed and the corresponding data structures are kept and maintained in the caddress hash table. Quadtree [31] is used as underlying index structure. Note however, that the underlying index structure is not limited to quadtree index, other spatial indices, like R-tree [27] and its variants, can also be used. No matter what kind of underlying index structure is used, the input of the index are regions occupied by mobile objects rather than mobile objects themselves, which is the key idea of VGQ. We select quadtree because its update operation is simpler than that of R-tree and its variants. For details of VGQ, please refer to [23].

4 VGQ-Vor and kNN query algorithm

In this section, we first introduce the architecture of VGQ-Vor, then propose a kNN algorithm based on VGQ-Vor (VGQ-Vor kNN), finally we give the correctness proof of the algorithm.

4.1 Architecture of VGQ-Vor

The main idea is to incorporate a Voronoi diagram into VGQ and build the Voronoi diagram from the regions occupied by

mobile objects instead of the mobile objects themselves. The resulting data structure, called VGQ-Vor, is a VGQ index enriched by a Voronoi diagram. Since the cost of building and maintaining a Voronoi diagram is high and the position changings of mobile objects are frequent, it is not efficient to build the Voronoi diagram directly from mobile objects. This is our motivation to build the Voronoi diagram on regions.

Figure 3(a) shows the architecture of VGQ-Vor. The Quadtree and virtual grid file are same as those shown in Fig. 2. Figure 3(b) shows an example of a Voronoi diagram built on the occupied regions. The occupancy status and identifiers are same as those shown in Fig. 2. The center points of the occupied regions are used to build the Voronoi diagram (in dotted lines). The corresponding Delaunay triangulation is drawn in solid lines.

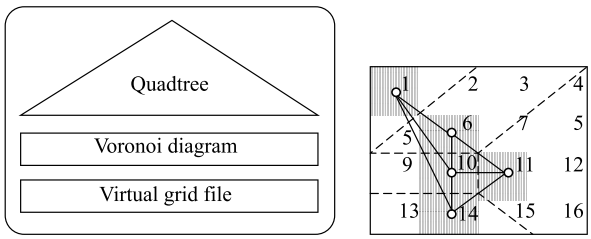

Fig. 3 (a) Archtecture of VGQ-Vor; (b) Voronoi diagram

When one empty region is occupied or one occupied region becomes empty, the maintenance algorithms are applied to VGQ-Vor. Given one occupied region, it is easy to find its kNN regions directly by following the connected edges of the Delaunay triangulation (Property 4 of Section 3 introduced on the Voronoi diagram). But for a mobile object inside a region, finding its kNN mobile objects can not be applied in the same way, because a region is not a point and there is a set of mobiles that are inside each region. The kNN mobile objects of a given point can be inside or outside the region where the point is located. So a new kNN algorithm based on VGQ-Vor is required. In next subsection, we introduce a kNN query algorithm based on VGQ-Vor.

4.2 kNN query algorithm on VGQ-Vor

The algorithm is called VGQ-Vor kNN. The idea is to check mobile objects inside neighboring regions of the region occupied by the mobile object which issued the query. As introduced above, the neighbor regions can be quickly obtained according to Voronoi diagram (Delaunay triangulation) built on regions (see Fig. 3(b)) and the neighbor regions are scanned in ascending order of distances between these regions and the kNN query object. The algorithm gets k candidate results from the query region and neighboring regions first, and then starts the checking process. The process ends when no more mobile objects with distance closer to the kNN query object than any of the k candidate results can be found. Based on the properties of the Voronoi diagram, the search space of the kNN query can be pruned by just scanning the regions which are the neighboring regions of the region containing the query.

Before introducing VGQ-Vor kNN in detail, we introduce some procedures and functions to be used in VGQ-Vor kNN as follows:

- **FillPointListWithRegion**(region, pointList) is a procedure to insert the points included in region to pointList. region is a grid cell of the virtual grid file and pointList is a linked-list of mobile objects. Each mobile object is regarded as a point.
- **SortPointListByQuery**(pointList, query) is a procedure to sort pointList in ascending order of distance to the query object.
- **FillRegionListWithRegion**(region, regionList) is a procedure to fill regionList with neighboring regions of region on the Voronoi diagram. regionList is a linked-list of regions.
- **PointToRegionDistance**(point, region) is a function to calculate the distance between point and region. The distance between a point to a region is the shortest distance from the point to the boundary of the region.
- **SortRegionListByQuery**(regionList, query) is a procedure to sort regionList in ascending order of distance to the query object.
- **PointToPointDistance**(pointlist, k, query) is a function to calculate the distance between the kth point of pointList and the query object.
- **GetAndRemoveFirstRegion**(regionList) is a function to return and remove the first region from regionlist.

The related algorithms of the above procedures and functions are straightforward. The pseudo code of VGQ-Vor kNN is shown in Algorithm 1. There are two exits (Line 10 and Line 21) in the algorithm. The algorithm comprises two parts. The first part (Lines 1–10) is to find k candidate points. The second part (Lines 11–21) checks the k points obtained in the first part, and find the kNN points of query point q. In the two parts, the region containing q is scanned first, then the neighboring regions (perhaps not adjacent to the region) in the Voronoi diagram are scanned. The critical problem

is how to scan the relevant regions and terminate the scan. In the following, we explain the algorithm in detail based on the example shown in Fig. 4, where q is a kNN query point and seven points $\{p_1, p_2, p_3, p_4, p_5, p_6, p_7\}$ occupy six regions $\{1, 3, 4, 6, 10, 16\}$.

Algorithm 1 VGQ-Vor kNN(q, k)

```
Data: q: query point
      k: number of NN points
Result: pointList: set of result points
1  Set curRegion to be the region containing q;
2  Insert curRegion to regionList;
3  repeat
4      curRegion = GetAndRemoveFirst(regionList);
5      FillPointListWithRegion(curRegion, pointList);
6      FillRegionListWithRegion(curRegion, regionList);
7      SortRegionListByQuery(regionList, q);
8  until (pointList.length ⩾ k) or (regionList.length = 0)
9  if (pointList.length < k) then
10     Return pointList;
11 repeat
12     SortPointListbyQuery(pointList, q);
13     BoundDis = PointToPointDistance(pointlist, k, q);
14     curRegion = GetAndRemoveFirstRegion(regionList);
15     newDis = PointToRegionDistance(curRegion, q);
16     if (newDis ⩽ BoundDis) then
17         FillPointListWithRegion(curRegion, pointList);
18         FillRegionListWithRegion(curRegion, regionList);
19         SortRegionListByQuery(regionList, q);
20 until (newDis > BoundDis);
21 Return the first k points in pointList;
```

First we show an 8NN query example wherein the algorithm exits at Line 10. Line 1 sets curRegion to 6 and Line 2 sets regionList to $\{6\}$. Since Region 6 contains p_4, p_4 is added to pointList (Lines 4–5). The neighbor regions $\{1, 3, 10, 16\}$ of Region 6 in the Voronoi diagram are added to regionList (Line 6) and regionList is sorted to $\{3, 1, 10, 16\}$ (Line 7). Because pointList.length (1) is less than k at Line 8, the loop (Lines 3–8) is repeats. In the next iteration, curRegion is 3, at the end of this iteration, the content of pointList is $\{p_4, p_3, p_2\}$ and the content of regionList is $\{(1, 10, 4, 16\}$. Here, Region 4 is inserted into regionList (Line 6) and it is not adjacent to Region 6. In the same way, the loop (Lines 3–8) is iterated until regionList becomes empty (regionList.length = 0) because there are a total of 7 points and k is 8. The algorithm exits at Line 10.

Second we show a 4NN query example where the algorithm exists at Line 21. Since k is 4, the first loop (Lines 3–8) is repeated 3 times. The content of pointList is $\{p_4, p_3, p_2, p_1\}$ and the content of regionList is $\{10, 4, 16\}$. Since pointList.Length (4) is not less than k (4), the second loop (Lines 11–20) is executed. Line 12 sorts pointList into $\{p_3, p_2, p_4, p_1\}$. boundDis is the distance between q and p_1 (Line 13), curRegion is Region 10 (Line 14) and newDis is the distance between q and Region 10 (Line 15). Because the newDis is less than the boundDis at Line 16, Lines 17–19 are executed and we repeat the loop once more (Lines 12–20). At this point, the content of regionList is $\{4, 16\}$ and the content of pointList is $\{p_3, p_2, p_4, p_1, p_5\}$. After executing Line 12, the content of pointList is sorted to $\{p_3, p_2, p_4, p_5, p_1\}$, because the distance between q and p_5 is less than that between q and p_1. boundDis is the distance between q and p_5 (Line 13), curRegion is Region 4 (Line 14) and newDis is the distance between q and Region 4. In Fig. 4, newDis is larger than boundDis, so we terminate the loop (Lines 11–19). Line 21 returns the results of the 4NN query $\{p_3, p_2, p_4, p_5\}$.

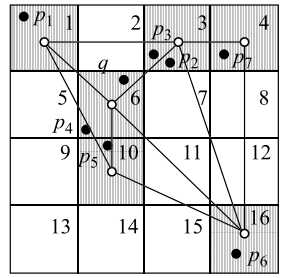

Fig. 4 Example of a kNN query based on VGQ-Vor

The update operation of a Voronoi diagram is more costly than that of a quadtree. So the the time complexity of the update operation on VGQ-Vor is the same order as that of a Voronoi diagram. Because the average number of neighbors of one point in a Voronoi diagram is less than 6 (Property 3 in Section 3.1). For a VGQ-Vor kNN query, the best time complexity is $O(k)$, the average time complexity is $O(6k)$ and the worst time complexity is $O(n)$, n is the number of regions. In order to store the Voronoi diagram, all the edges of the Delauany triangulation are stored, and it needs extra space. According to Property 3, the space complexity is $O(n)$.

4.3 Correctness proof of Vor-VGQ kNN algorithm

Based on the properties of Voronoi diagrams, the search space of a kNN query can be pruned by simple scanning the regions which are neighbors regions of the region containing q or the regions having been scanned. As shown in Fig. 4, the neighbors do not need to be adjacent to the region containing q, it depends on the value of k and data distribution. The sequence of scanning regions is in ascending order of their distances to q. Next, we prove that the VGQ-Vor kNN algorithm finds kNNs correctly.

Theorem 1 Let P be a set of points, R be a set of points obtained by VGQ-Vor kNN(q, k) on P, boundDis be the distance from q to the kth point in R. For a point $\acute{p} \in P$ with distance to q $\acute{p}$Dis, if $\acute{p}$Dis < boundDis, then $\acute{p}$ must be included in R.

Proof Suppose there exists a point $\acute{p} \in P$, $\acute{p} \notin R$ and $\acute{p}$Dis < boundDis. If $\acute{p}$ is not be found by the VGQ-Vor kNN algorithm, it means the distance between q and the region containing $\acute{p}$ ($\acute{p}$RegionDis) is larger than or equal to boundDis. Because $\acute{p}$ is inside the region and $\acute{p}Dis \geqslant \acute{p}$RegionDis is true, so $\acute{p}$Dis $\geqslant$ boundDis is true. This is contradictory to the supposition $\acute{p}$Dis < boundDis. Meaning that $\acute{p}$ cannot be missed by VGQ-Vor kNN. □

5 Evaluation

5.1 Environment setup

VGQ-Vor algorithm is evaluated in a simulated environment and compared with existed works (Bx-tree [19], Bdual-tree [20]). As introduced in Section 2, a lot of work have been done on this topic. A benchmark was proposed in [30] for evaluating moving object indexes comparing the representative mobile indexes (TPR-tree [15], RUM-tree [26], Bx-tree [19], STRIPES [17], Bdual-tree [20]) which support mobile queries on mobile objects. Here we compare VQG-Vor with Bx-tree and Bual-tree, because they have better performance than the others. In our evaluation, two data generators are used, the Spade [30] and Brinkhoff data generators.

All algorithms are implemented in C++ and compiled with GNU GCC. The source code of the Spade data generator, Bx-tree, and Bdual-tree can be downloaded from http://www.comp.nus.edu.sg/~spade/releases.html. The source code of Brinkhoff data generator is downloaded from http://iapg.jade-hs.de/personen/brinkhoff/generator/. The hardware platform is an IBM X3500 server with 2 Quad Core 1 333 MHz CPUs and 16 GB of memory under linux (Red Hat 4.1.2–42).

Table 1 summarizes the parameters used in the evaluation which may have a potential impact on our performance. In the experiments, all parameters use the default values unless otherwise specified.

Table 1 Specifications of parameters

Parameter	Value range	Default value
Number of mobile objects	40k, 60k, 80k, 100k, 150k, 200k	100k
k	5, 10, 15, 20, 25, 30, 40, 50, 60, 70, 80, 90, 100	10
Update/query ratio	1, 10, 100, 1 000, 10 000	100
Regions (row×column)	100×100, 150×150, 200×200, 250×250, 300×300, 350×350	250×250

5.2 Evaluation results

The evaluation has two parts. In the first part, we evaluate kNN queries on VGQ-Vor with different parameters specified in Table 1 with data generated by the Brinkhoff data generator. In the second parts, we compare VGQ-Vor with existing works with data generated by the Spade data generator.

5.2.1 Evaluation of VGQ-Vor and kNN algorithm

Figure 5 shows the average query time with different numbers of mobile objects. The number of mobile objects ranges from 40 000 to 200 000. The average query time shows a trend of light growth with regard to the number of mobile objects. When the number of mobile objects increases, the number of mobile objects inside a region becomes larger. So it takes more time to check the mobile objects inside regions.

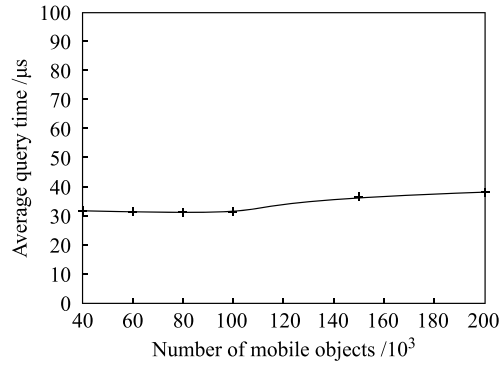

Fig. 5 Average query time with different number of mobile objects

Figure 6 shows the average update time with different numbers of mobile objects. The average update time decreases with the increment of mobile objects. The reason is that the more objects there are, the less the frequency of occupancy status changes. Since the average query time increases lightly and the average update time decreases with regard to the number of mobile objects, we feel Vor-VGQ is suitable to

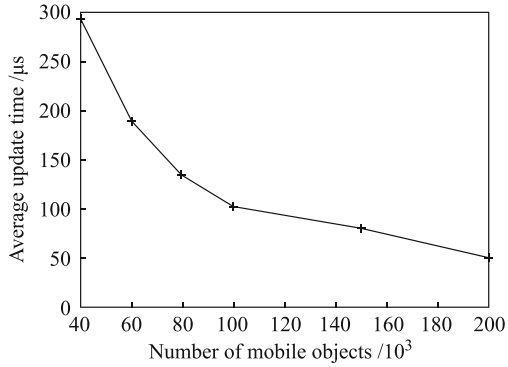

Fig. 6 Average update time with different number of mobile objects

deal with mobile applications within a reasonable scale, for example 40k–200k.

Figure 7 shows the average query time with different k values. The k value ranges from 5 to 30. The average time becomes larger with the increment of k value, because more regions needs to be scanned. The reason that the query time increases slowly with the increment of k is that the number of regions to be accessed does not increase linearly with k. In our case, one region contains about 1.5 points. When k is 30, the results are kept in at least 20 regions. Because each region has fewer than six neighbors on average, so VGQ-Vor scans about 120 regions on average, which contains about 360 points. It is very possible that the kNN query results can be found with these points. When k is smaller than 360 in a reasonable range, for example (20–50) shown in Fig. 13, the query time does not change very much.

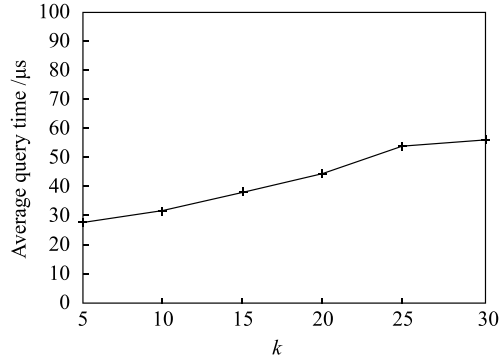

Fig. 7 Average query time with different k values

Figure 8 shows the throughput of VGQ-Vor with different ratios of update/query. The ratio of number of update operations to the number of query operations ranges from 1 to 10 000. The value 1 means there is one position update for each mobile query. According to the results shown in Fig. 5 and Fig. 6, the performance of query operation is better than that of update operation. The difference depends on the number of mobile objects and changing frequency of occupancy state of a region. The higher the ratio is, the lower the cost benefit of query operation is. Here, when the ratio is larger than 1 000, the performance of update is dominant. Since a query mobile object is retrieved by other mobile objects, so a query mobile object triggers both query operation and update operation, that is the reason that the throughput increases with the ratio.

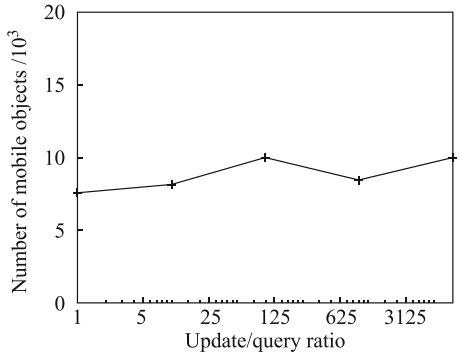

Fig. 8 Throughput of VGQ-Vor with different ratios of update/query

Figure 9 shows the average query time with different space partitions. Here the number of regions ranges from 100×100 to 350×350. The average query time reduces as the number of regions becomes large. The reason is that when the number of mobiles objects remains constant, the larger the number of regions is, the fewer mobile objects are in each region. It means the Vor-VGQ kNN checks fewer mobile objects for each region. So the query time becomes shorter.

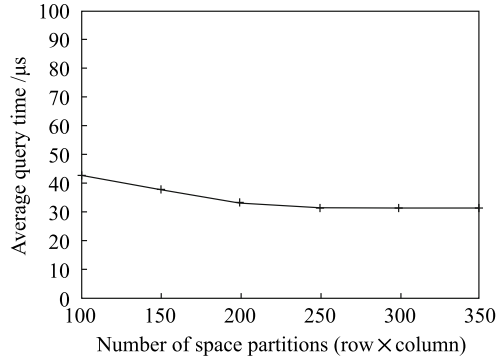

Fig. 9 Average query time with different different space partitions

Figure 10 shows the average update time with different number of space partitions. In contrast to the average query time, the average update time increases with the increases in the number of regions. When the number of mobile objects is fixed, the larger the number of regions is, the higher the frequency of occupancy state changing is. The extreme case is that each region corresponds to a single point, in this case, each movement of a mobile object causes an occupancy change. This is why the average update time becomes large when the number of regions becomes large.

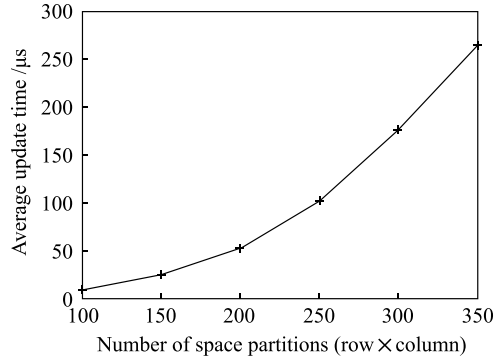

Fig. 10 Average update time with different space partitions

5.2.2 Comparison with existing work

We compare VGQ-Vor with Bx-tree [19] and Bdual-tree [20]. Bdual-tree is an extension of Bx-tree, which utilizes velocity information to obtain better performance than Bdual-tree. Here Spade data generator [30] is used to create test data. There are two reasons to use Spade data generator instead of Brinkoff generator [32]. One is that we want to test VGQ-Vor with different data sets. The second reason is that the data format requirements of Bx-tree and Bdual-tree are special, the Brinkoff data generator cannot be used directly. In [30], Bx-tree and Bdual-tree are compared with data created by the Spade data generator. In most cases, Bx-tree and Bdual-tree outperform the other index structures.

Figure 11 and Fig. 12 show the average query time and update time with different number of mobile objects. Just as in [30], Bdual-tree outperforms Bx-tree on query operation, and Bx-tree outperforms Bdual-tree on update operation. For both query operation and update operation, VGQ-Vor outperforms Bx-tree and Bdual-tree in most of cases. The main reason is the strategy to build mobile object index. VGQ-Vor indexes regions occupied by the mobile objects, so the number of update operations can be reduced and the mobile objects can even move from one region to another region. In Fig. 12, the average update time increases with the number of mobiles objects. It seems to be contrary to the time reported in Fig. 6, the reason is that in the test of Fig. 12, the number of queries is fixed (same source codes and setting as [30]), the ratio of the number of update operations to the number of query operations is not fixed. The ratio increases from 40:1 to 200:1, this means the number of position changes increases, and therefore so do the number of update operations on the Voronoi diagram.

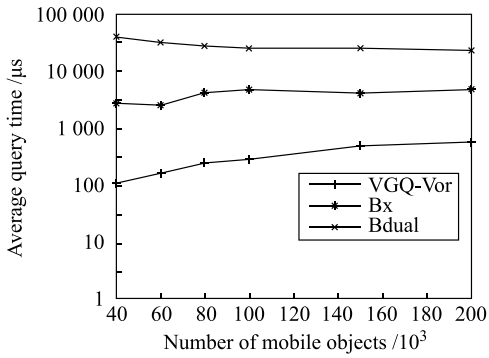

Fig. 11 Average query time with different number of mobile objects

Figure 13 shows the average query time with regards to different k values for kNN queries. Here, the k ranges from 1 to 100. Similar to Fig. 11, Bdual-tree outperforms Bx-tree,

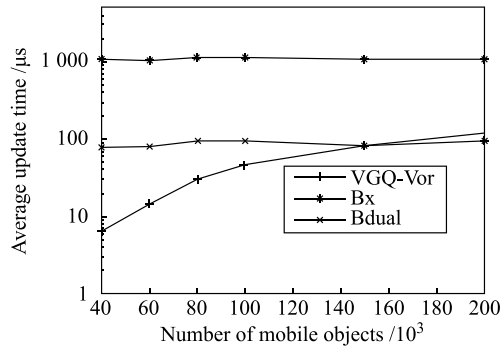

Fig. 12 Average update time with different number of mobile objects

and VGQ-Vor outperforms Bdual-tree and Bx-tree.

Figure 14 shows the throughput results. The ratio of the number of update operations to the number of query operations ranges from 1 to 10 000. Here, the changing trends of Bx-tree and Bdual-tree are same as those evaluated in [30]. When the ratio is 1, it means the number of update operations is the same as that of query operation, VGQ-Vor outperforms Bx-tree and Bdual-tree by three orders of magnitude in this case. The reason why the performance of Bx-tree and Bdual-tree increases with the increment of the ratio is that they use B-tree. The increment of the ratio means the update operation becomes dominant. Because the cost of the update operation is much less than that of a range query operation on B-tree,

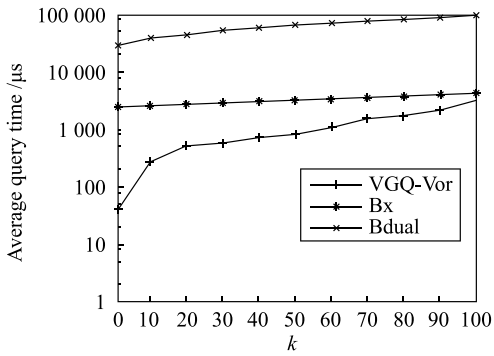

Fig. 13 Average query time with regard to different k values for kNN queries

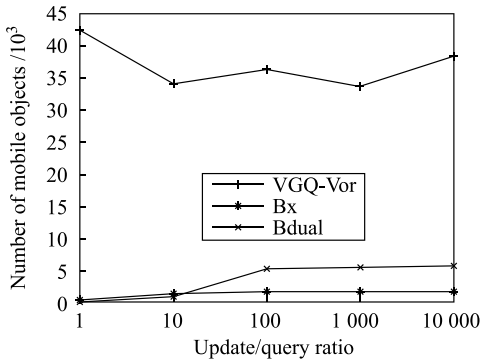

Fig. 14 Throughput with different ratios of update/query

so the throughput increases when the ratio increases. The throughput of VGQ-Vor mainly depends on the performance of the update operation of Voronoi diagram.

6 Conclusions and future work

Spatial indexes for mobile objects are complicated because the locations of mobile objects are changing frequently. For the problem of mobile kNN queries, we have proposed an index structure called VGQ-Vor, which incorporates Voronoi diagrams into a VGQ, designed a kNN query algorithm VGQ-Vor kNN, and provided a correctness proof of the algorithm. The main characteristics of VGQ-Vor are that a Voronoi diagram is built on the regions occupied by mobile objects instead of the mobile object themselves in order to reduce the number of update operations on the index structure and improve the efficiency of mobile kNN queries. We evaluated VGQ-Vor and the kNN algorithm in two data sets and compared Bx-tree and Bdual-tree with VGQ-Vor. The results show that the query performance of VGQ-Vor has good scalability with regards to the number of objects and the update performance of VGQ-Vor decreases with the increment of the number of mobile objects if the ratio of number of update operations to the number query operations is fixed. In the most cases, VGQ-Vor outperforms Bx-tree and Bdual-tree by one to three orders of magnitude.

In the future, we will focus on: 1) finding the best space partition to improve the throughput based on the cost tradeoff between query operation and update operation; 2) to improve the performance of update operation by optimizing maintainence algorithms on Voronoi diagram.

Acknowledgements This paper was supported by the National Natural Science Foundation of China (Grant Nos. 61173030, 60803026, 61073063).

References

1. Roussopoulos N, Kelley S, Vincent F. Nearest neighbor queries. ACM SIGMOD Record, 1995, 24(2): 71–79
2. Seidl T, Kriegel H. Optimal multi-step k-nearest neighbor search. ACM SIGMOD Record, 1998, 27(2): 154–165
3. Chaudhuri S, Gravano L. Evaluating top-k selection queries. In: Proceedings of the 25th International Conference on Very Large Data Bases. 1999, 397–410
4. Kalashnikov D, Prabhakar S, Hambrusch S. Main memory evaluation of monitoring queries over moving objects. Distributed and Parallel Databases, 2004, 15(2): 117–135
5. Cai Y, Hua K, Cao G. Processing range-monitoring queries on heterogeneous mobile objects. In: Proceedings of the 2004 IEEE International Conference on Mobile Data Management. 2004, 27–38
6. Song Z, Roussopoulos N. k-nearest neighbor search for moving query point. Advances in Spatial and Temporal Databases, 2001, 79–96
7. Tao Y, Papadias D, Shen Q. Continuous nearest neighbor search. In: Proceedings of the 28th International Conference on Very Large Data Bases. 2002, 287–298
8. Kolahdouzan M, Shahabi C. Voronoi-based k nearest neighbor search for spatial network databases. In: Proceedings of the 30th International Conference on Very Large Data Bases. 2004, 840–851
9. Nutanong S, Zhang R, Tanin E, Kulik L. The v*-diagram: a query-dependent approach to moving knn queries. Proceedings of the VLDB Endowment, 2008, 1(1): 1095–1106
10. Sharifzadeh M, Shahabi C. Vor-tree: R-trees with voronoi diagrams for efficient processing of spatial nearest neighbor queries. Proceedings of the VLDB Endowment, 2010, 3(1–2): 1231–1242
11. Yu X, Pu K, Koudas N. Monitoring k-nearest neighbor queries over moving objects. In: Proceedings of the 21st International Conference on Data Engineering. ICDE'05. 2005, 631–642
12. Xiong X, Mokbel M, Aref W. Sea-cnn: scalable processing of continuous k-nearest neighbor queries in spatio-temporal databases. In: Proceedings of the 21st International Conference on Data Engineering. ICDE'05. 2005, 643–654
13. Mouratidis K, Yiu M, Papadias D, Mamoulis N. Continuous nearest neighbor monitoring in road networks. In: Proceedings of the 32nd International Conference on Very Large Data Bases. 2006, 43–54
14. Fan P, Li G, Yuan L, Li Y. Vague continuous k-nearest neighbor queries over moving objects with uncertain velocity in road networks. Information Systems, 2012, 37(1): 13–32
15. Šaltenis S, Jensen C, Leutenegger S, Lopez M. Indexing the positions of continuously moving objects. ACM SIGMOD Record, 2000, 29(2): 331–342
16. Tao Y, Papadias D, Sun J. The TPR*-tree: an optimized spatio-temporal access method for predictive queries. In: Proceedings of the 29th International Conference on Very Large Data Bases. 2003, 790–801
17. Patel J, Chen Y, Chakka V. STRIPES: an efficient index for predicted trajectories. In: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data. 2004, 635–646
18. Silva Y N, Xiong X P, Aref W G. The RUM-tree: supporting frequent updates in R-trees using memos. The VLDB Journal, 2009, 18(3): 719–738
19. Jensen C, Lin D, Ooi B. Query and update efficient B+-tree based indexing of moving objects. In: Proceedings of the 30th International Conference on Very Large Data Bases. 2004, 768–779
20. Tao Y, Xiao X. Primal or dual: which promises faster spatiotemporal search? The VLDB Journal, 2008, 17(5): 1253–1270
21. Chen S, Ooi B, Tan K, Nascimento M. ST^{2}B-tree: a self-tunable spatio-temporal b$^+$-tree index for moving objects. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data. 2008, 29–42
22. Okabe A, Boots B, Sugihara K, Chiu S N. Spatial Tessellations: Concepts and Applications of Voronoi Diagrams. Wiley, 2000
23. Wang B, Chen H, Ma J, Kitsuregawa M, Wang G. Design and implementation of mobile object index based on covered area. Journal of Frontiers of Computer Science and Technology, 2010, 4(1): 64–72

24. Chen Z, Shen H, Zhou X, Yu J. Monitoring path nearest neighbor in road networks. In: Proceedings of the 35th SIGMOD International Conference on Management of Data. 2009, 591–602
25. Zheng K, Trajcevski G, Zhou X, Scheuermann P. Probabilistic range queries for uncertain trajectories on road networks. In: Proceedings of the 14th International Conference on Extending Database Technology. 2011, 283–294
26. Xiong X, Aref W. R-trees with update memos. In: Proceedings of the 22nd International Conference on Data Engineering. ICDE'06. 2006, 22–22
27. Guttman A. R-trees: a dynamic index structure for spatial searching. Readings in Database Systems, 1988, 599–609
28. Zhang R, Jagadish H, Dai B, Ramamohanarao K. Optimized algorithms for predictive range and kNN queries on moving objects. Information Systems, 2010, 35(8): 911–932
29. Güting R, Behr T, Xu J. Efficient *k*-nearest neighbor search on moving object trajectories. The VLDB Journal, 2010, 19(5): 687–714
30. Chen S, Jensen C, Lin D. A benchmark for evaluating moving object indexes. Proceedings of the VLDB Endowment, 2008, 1(2): 1574–1585
31. Finkel R, Bentley J. Quad trees a data structure for retrieval on composite keys. Acta Informatica, 1974, 4(1): 1–9
32. Brinkhoff T. A framework for generating network-based moving objects. GeoInformatica, 2002, 6(2): 153–180

Botao Wang received his PhD in Computer Science in 2000 from Kyushu University. Currently, he is a professor in the Department of Information Science and Engineering, Northeastern University. His research interests include spatial-temporal databases, publish/subscribe systems, data streaming and mobile data management.

Jingwei Qu received his BS in Computer Science from Northeastern University, in 2009. He received his MS in Computer Systems from Northeastern University, in 2011. His research focuses on mobile data management.

Xiaosong Wang received her BS in Computer Science from Anhui Construction Industry Institute, in 2009. She is currently an MS candidate in Computer Application Technology at Northeastern University. Her research focuses on mobile data management.

Guoren Wang received his BS, MS, and PhD from Northeastern University in 1988, 1991, and 1996 respectively, all in Computer Science. He is currently a full professor and doctoral supervisor in the Department of Information Science and Engineering, Northeastern University. His research interests include XML data management, data streaming analysis, high-dimensional indexing, and P2P data management.

Masaru Kitsuregawa received his PhD in Information Engineering in 1983 from the University of Tokyo. Now he is a professor and director of Center for Information at the Institute of Industrial Science, University of Tokyo. His research interests include parallel processing and database system.