Processing Continuous *k* –Nearest Neighbor Queries in Location-Dependent Application

Wei Zhang, Jianzhong Li, and Haiwei Pan

School of Computer Science and Technology, Harbin Institute of Technology, Harbin, Heilongjiang, China

Summary
A *k* nearest neighbor (*k*-NN) query retrieves *k* objects in a given objects set which are closest to the query point *q*. Processing continuous *k*-nearest neighbor (*k*-NN) query over moving objects in location-dependent application requires that the frequent location updates of moving objects and intensive continuous *k*-NN queries must be efficiently processed at the same time. In this paper, we propose a grid cell based continuous *k*-NN query processing method (CkNN). It utilizes a main memory grid index to store the location of moving objects. Efficient *k*-NN search algorithm and incremental query processing algorithm are designed in CkNN. CkNN minimizes the cost of continuous *k*-NN query processing by reducing most unnecessary checking on queries / moving objects and reusing data obtained during query processing as moor as possible. The comprehensive experimental evaluation shows that CkNN outperforms state-of-the-art continuous *k*-NN query processing approach in all problem settings.

Key words:
Query Processing, Continuous Query, Location-dependent, Spatio-temporal

Introduction

As the development of positioning technology wireless communication, the widely applied location-dependent applications require new techniques to manage the information of moving objects. Recently, processing continuous *k*-NN queries over moving objects attracts considerable attentions. Besides computing the *k*-NN of queries after they are issued, the system needs to maintain the results of continuous queries up-to-date at each update cycle. The challenge of this problem is to efficiently handle frequent location updates as well as process intensive continuous *k*-NN queries. Early research in spatial databases focused on processing the *k*-NN query which retrieves *k* objects from the static dataset that are nearest to a static query point according to Euclidean distance. The existing algorithms in spatial database consider that the data are indexed by a spatial access method (e.g. R-Tree) and utilize some branch-and-bound approach to restrict the search space. In addition, several papers study variations of k-NN problems such as reverse *k*-NN [14] and constrained *k*-NN query [15]. However, all R-Tree based *k*-NN computing algorithms are designed for processing queries over static data, those traditional solutions in spatial database can not extend to the highly dynamic applications, e.g. frequent location update of moving objects.

Comparing with the traditional secondary memory based approaches, main memory based access methods is a better choice while processing continuous *k*-NN queries over constantly moving objects. Location-dependent application is characterized by a large number of objects and a large number of continuous queries (or users). Most users require the system answer their queries as soon as possible or even process their query in real time. However, the load of location-dependent service is heavier as it becomes more popular, and the response time increases, since more mobile objects are monitored and more continuous queries registered in the system. In [3], Kalashnikov et al. proposed that main memory grid index is an effective structure for processing continuous range queries over moving objects. This hash structure can efficiently supporting frequent location updates of moving objects, and the grid partition is also benefited to query processing algorithm. Nowadays, the price of main memory is much lower than ten years ago. It is common for a computing server which is equipped with several gigabyte main memories. Therefore, it is not only applicable but also necessary to research an efficient main memory continuous *k*-NN query processing algorithm for location-dependent application.

In the paper, we propose a grid cell level based continuous *k*-NN query processing algorithm, called CkNN for short. It utilizes a main memory grid index to store the moving objects. CkNN processes new registered queries by *k*-NN search algorithm. It searches the *k*-NN of queries according to the partition of grid cell level. During query processing, CkNN tries to minimize the cost of checking grid cells and moving objects. While processing static continuous *k*-NN queries, CkNN employ an incremental update and query processing algorithm. The incremental algorithm makes the most of information obtained in last query processing phase, and attempts to reuse the data produced in query processing as moor as possible. The comprehensive experimental evaluation shows that CkNN outperforms state-of-the-art continuous *k*-NN query processing approach in all problem settings.

The rest of the paper is organized as follows. Section 2 surveys related work on processing continuous *k*-NN query. Section 3 proposes the grid cell level based continuous *k*-NN query processing method. Section 4 presents the experimental evaluation of CkNN. Finally, section 5 concludes the works in the paper.

Manuscript received February 2, 2006.
Manuscript reviced February 28, 2006.

2. Related Work

Processing *k*-NN query over static objects has been well-studied in spatial database. The most widely applied approach is the branch-and-bound algorithm based on R-tree [8]. The algorithm traverses R-tree through a best-first search and maintains a priority list of *k*-NN candidates. Since traditional methods in spatial database are designed for processing *k*-NN queries over static data, those approaches can not efficiently support the highly dynamic location-dependent applications where the information of moving objects is updated frequently. Song and Roussopoulos [10] studied the problem of processing moving query over static data. Their approach attempts to reduce the cost of re-computing *k*-NN for moved queries by returning redundant objects with current *k*-NN results. If the moved query can be satisfied by recently received objects, *k*-NN computation is avoided. Tao and Papadias proposed a time-parameterized query which assumes objects move with linear and known velocities, returns validity period and next change of current results [11]. Based on the linear movement assumption, Kollios *et al.* [4] designed an algorithm for processing *k*-NN query over 1D (and 1.5D) moving objects. For two or higher dimensional, Benetis *et al.* [1] proposed the algorithm for processing predictive *k*-NN and reverse nearest neighbor query by employing TPR-Tree [9]. Above predictive *k*-NN processing algorithms require the velocity of moving object is available at query time. If the linear movement assumption does not hold, query results become invalid. CkNN does not make any assumption about the movement patterns of moving objects. It can process static or moving queries over moving objects.

Processing continuous spatial queries over moving objects is first considered in [7], where static range query is indexed by an R-Tree based structure called Q-index and moving objects probe Q-index to invoke updating of influenced queries. *Mobieyes* [2] monitors continuous moving range queries over moving objects in distributed environment, while *SINA* [5] processes continuous range queries in a center server. Kalashnikov *et al.* [3] proposed that main memory grid index is more suitable for monitoring continues range query over moving objects than R-Tree based implementation. All aforementioned methods are focus on processing continuous range queries, and can not efficiently extend for *k*-NN query processing. Recently, Yu *et al.* [12], Xiong *et al.* [12], and Mouratidis *et al.* [6] proposed three approaches for processing continuous *k*-NN query based on grid index, hereafter referred to as YPK-CNN, SEA-CNN, and CPM, respectively. YPK-CNN and CPM employ main memory regular grid index, while SEA-CNN indexes objects in secondary memory with regular grid index. SEA-CNN only focuses on monitoring the changes in *k*-NN result of continuous queries (It assume the initial *k*-NN results are available). For any *k*-NN query point *q*, let *q.kNN-dist* denote the distance between *q* and its *k*th nearest neighbor. In SEA-CNN, the "*influence region*" is defined for every query *q*, which is centered at *q* with radius *q.kNN-dist*. The identifier of *q* is inserted into all cells overlapping *q*'s influence region. The query *q* is re-evaluated when any location updates of moving objects are related to cells which record *q*'s identifier. If any of the current *k*-NN of *q* moves out of *q*'s influence region, the radius of search region is enlarged to the distance to the previous *k*-NN which moved furthest from *q*, otherwise the radius of search region keeps *q.kNN-dist*. After the search region is determined, all moving objects in the search region are scanned for updating *q*'s new *k*-NN. In YPK-CNN, the registered continuous queries are re-evaluated every T time units (hereafter referred to as *result update cycle*). When a query *q* is evaluated for the first time, YPK-CNN employs a two-phase search to compute *k*-NN of *q*. In the first phase, the search algorithm start from the cell c_q containing *q*, and the square search region centered at c_q is iteratively enlarged until initial *k* candidates are found. In the second phase, the search region is enlarged to the square region centered at c_q with side length $(2 \times d) + \delta$, where *d* is the distance of furthest candidate object from *q*, and δ is the cell side length. All moving objects in cells which overlap with the square region are checked to determine the actual *k*-NN of *q*. The system architecture and index structure of CPM are same as that of YPK-CNN. When computing the initial *k*-NN of a query *q*, CPM partitions grid cells around *q* into conceptual rectangles according to their proximity to *q*. The conceptual rectangles are labeled by a direction and a level number. The direction is U, D, L, or R, stands for up, down, left, and right. The level number indicates how far the rectangle is from *q*. CPM sorts cells and conceptual rectangles based on their minimum distance to *q*. The sorted cells and rectangles are accessed in a best-first way to obtain the *k*-NN of *q*. For continuous query processing, CPM employs an incremental approach to monitor the changes in results of processed queries. The idea of the incremental approach is to book the query *q* into all cells intersecting *q*'s influence region (which is same as that defined in SEA-CNN), and try to compute new result based on all moving objects still in *q*'s influence region. If the algorithm fails in getting enough results, *k*-NN of *q* is recomputed. The aforementioned continuous *k*-NN monitoring method is most related work to this paper. Similar to these methods CkNN also assumes continuous queries are processed by the centralized server (in main memory). Since CPM outperforms YPK-CNN and SEA-CNN [6], we only compare CkNN with CPM. In the next section, we present CkNN in detail.

3. Continuous *K*-NN Query Processing in Main Memory Grid Index

The paper focuses on processing continuous *k*-NN query in main memory grid index. We assume objects move in 2-D space. The space is normalized to $[0, 1) \times [0, 1)$. The grid index divides the space into $n \times n$ non-overlap grid cells. The side length of each cell is $\delta = 1/n$. Section 3.1 gives the overview of system architecture and continuous query processing procedure. Section 3.2 designs a grid cell level based *k*-NN search algorithm. Section 3.3 proposes an incremental algorithm for continuous query processing.

3.1 System Architecture

The continuous *k*-NN query processing system includes three parts:

1. *Object table*. It is a hash table for storing moving objects' coordinates. The hashed values of objects identifiers are used to locate their coordinates. This implementation provides the most efficient access to coordinates of moving objects.
2. *Query table*. It is a hash table for storing the information of continuous *k*-NN queries. Similar to the object table, continuous queries are also located by their identifiers. The entry of a *k*-NN query *q* contains the identifier, the coordinate of *q*, the number of required nearest neighbors *k*, the maximum distance between *q* and its *k*th nearest neighbors *q.kNN-dist*, and the list of nearest neighbors *q.kNN-list*.
3. *Grid index*. The grid index is composed by grid cells. Each cell has two lists, object list and query list. Objects list stores the identifiers of objects which currently appear in the cell. Query list records the identifiers of queries whose influence region overlap the cell. The query list is used to support incremental query processing. In order to efficiently support intensive updates, the two lists are also implemented in hash table.

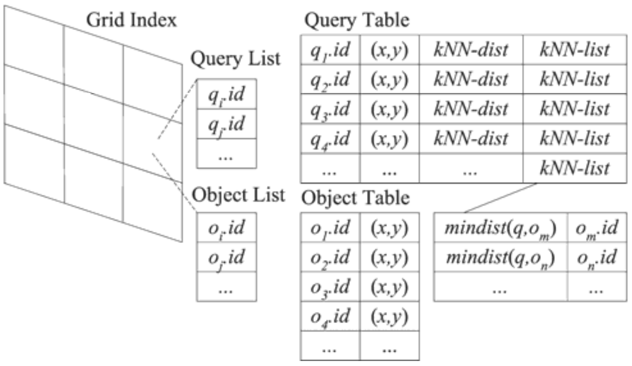

Fig. 1 System Structure

During each result update cycle, the system first updates objects table, query table and grid index, then processes registered continuous queries. While updating the coordinate of a moving object *O* in the object table, if *O* moves across the cells of grid index, *O*'s identifier are deleted from the object list of "old" cell and inserted in the object list of "new" cell. When a new continuous *k*-NN query *q* is registered in the system, the *k*-NN search algorithm is invoked to search the initial result of *q*. For all unmoved continuous queries, the incremental algorithm is applied to monitor the change of the query results. For all moved continuous queries, at first, their identifiers are removed form query lists in grid index. Then, their coordinates are updated in the query table. The *kNN-dist* and *kNN-list* are cleared as well. Finally, all moved queries are reevaluated by *k*-NN search algorithm.

3.2 *k*-NN Search Algorithm

The *k*-NN search algorithm is used to compute the query results from scratch for new registered queries and moved queries. As showed in Fig. 2, the algorithm divides grid cells around the query point *q* into multiple levels, e.g. the cell containing *q* is level 0 denoting as L_0; all cells around L_0 construct level 1 (L_1); the cells around L_1 is level 2 (L_2), and so on. When a cell level L_i is accessed during *k*-NN search, the full clockwise scan of all cells in L_i starts from the left bottom cell, i.e. cell $c_{0,0}$, $c_{1,1}$ and $c_{2,2}$. The basic ideal of the *k*-NN search algorithm is to construct initial *k*-NN candidates from cell levels around the query point at first, and then use objects in the levels around those *k*-NN candidates to refine the query result until all cells that possibly contains *k*-NN of query point have been checked. Meanwhile, the algorithm also tries to obtain the result by checking as few objects as possible.

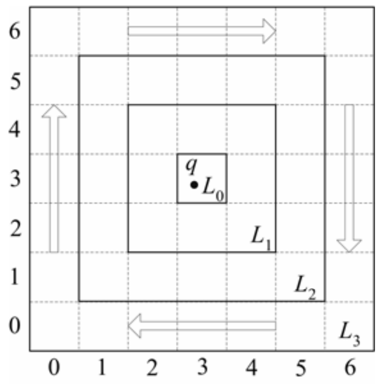

Fig. 2 Partition of Cell Levels

The *k*-NN search algorithm includes two phase. In the first phase, if the total number of retrieved objects and objects in current cell level is not greater than *k*, the algorithm directly retrieves objects from current cell levels. These objects (often less than *k* objects) are used to construct initial *k*-NN candidates. In the second phase, the algorithm builds and refines the final *k*-NN results according to objects in the cell levels whose minimum distance to the query is less than the *kNN-dist* of the query. In this phase, all cells in a cell level are sorted based on their minimum distance to the query. The algorithm visits these cells in a best-first manner. For a given query *q*, let *mindist*(*q*, *c*) represent the minimum distance between *q* and cell *c*, *mindist*(*q*, L_i) represent the minimum distance between *q* and cell level L_i, *q.kNN-dist* denotes the minimum distance between query point *q* and its *k*th nearest neighbor, and $N(L_i)$ denote the total number of objects in cell level L_i, respectively. In order to save the cost of sorting cells, only qualified cells are sorted. A qualified cell *c* must satisfy two criteria, (1) *c* contains at least one object, (2) *mindist*(*q*, *c*) < *q.kNN-dist*. Moreover, once the minimum distance of current sorted cells to *q* is greater than *q.kNN-dist*, all remaining sorted cells are discarded before sorting cells in next cell level. The algorithm is terminated when the minimum distance of next cell level to q is not less than *q.kNN-dist*.

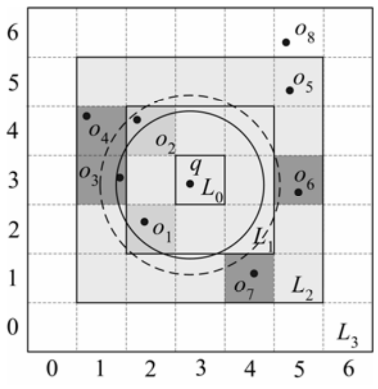

Fig. 3 Example of 2-NN Search in Cell Levels

We use Fig. 3 as example to explain the idea of cell level based k-NN search algorithm. The circle area center at the k-NN query q with the radius of *q.kNN-dist* is the influence region of q, such as the circle area enclosed by the dashed line or the real line in Fig. 3. The search procedure of 2-NN query q starts at L_0. Cell level L_0 is ignored for containing no objects. Then, the search range expands to L_1. Since $N(L_1) \leq 2$, all cells in L_1 are sequentially scanned and objects in them are directly retrieved to build the initial 2-NN candidates. While obtaining the initial candidates o_1 and o_2, *q.kNN-dist* is set to the distance between q and o_2, $dist(q, o_2)$, and the influence region of q is also determined (the area enclosed by the dashed line). Before accessing cells in L_2, the value of $mindist(q, L_2)$ is computed. The cells in L_2 is visited only when cell level L_2 overlaps current influence region of q, i.e. $mindist(q, L_2) <$ *q.kNN-dist*. While accessing L_2, all cells in it are inserted into a sorted heap *SH*. Accordingly, four cells in L_2 are en-heaped into *SH*, i.e. $SH = \{ < c_{1,3}, mindist(q, c_{1,3}) >, < c_{1,4}, mindist(q, c_{1,4}) >, < c_{4,1}, mindist(q, c_{4,1}) >, < c_{5,3}, mindist(q, c_{5,3}) > \}$. Although $c_{5,5}$ is not empty, it is not inserted into *SH* since $mindist(q, c_{5,5}) >$ *q.kNN-dist*. After $c_{1,3}$ is de-heaped, a better 2-NN o_3 is found and influence region is updated (the area enclosed by real line). Next, $c_{1,4}$ is de-heaped and o_4 is discarded. Since $mindist(q, c_{4,1})$ is greater than current *q.kNN-dist*, the rest entries in *SH* does not need to be visited and the heap is cleared. When the search range expands to L_3 and $mindist(q, L_3) <$ *q.kNN-dist*, the algorithm terminates and returns o_1 and o_3 as 2-NN of q. In Fig. 3, all grey cells are visited during k-NN search. The darker cells contain the initial k-NN candidates and the darkest cells are en-heaped in *SH*. Fig. 4 gives the pseudocode of k-NN search algorithm.

Unlike the algorithm proposed in [6], our k-NN search algorithm avoids sorting cells which must be accessed for obtaining the initial k-NN candidates. Moreover, the criteria for filtering cells effectively reduce the cost of sorting cells in heap. This is also confirmed by the experimental evaluation.

```
Algorithm: k-NN Search (GI, QT, q)
Input: Grid Index GI, Query Table QT, k-NN Query q;
Output: q.kNN-list

1.  q.kNN-list = Ø; /*Initialize k-NN*/
2.  i = 0; /*Initialize cell level*/
3.  q.kNN-dist = ∞; /*Initialize kNN-dist */
    /*Construct initial k-NN candidates*/
4.  While ( q.kNN-list.size() + N(L_i) ≤ q.k) do
5.     Insert all objects in L_i into q.kNN-list;
6.        If (q.kNN-list.size() = = q.k) then
7.           Update q.kNN-dist;
8.     i = i + 1; /*Increase cell level*/
9.  End While
    /*Get mindist between q and current cell level*/
10. MinLev-dist = mindist(q, L_i);
11. While ((q.kNN-list.size() < q.k) or
             (MinLev-dist < q.kNN-dist)) do
12.    For all cells c_j ∈ L_i do
13.       If ((c_j not empty) and ((q.kNN-list.size() < k)
             or (mindist(q, c_j) < q.kNN-dist))) then
14.            En-heap entry < c_j, mindist(q, c_j)> in SH;
15.    While SH is not empty do
16.       De-heap next entry<c, mindist(q,c)>from SH;
17.          If (q.kNN-dist < mindist(q, c)) then
18.             For each object o in cell c do
19.                If (q.kNN-list.size() < k) then
20.                   Insert o into q.kNN-list;
21.                Else
22.                   Update q.kNN-list and q.kNN-dist;
23.             End For
24.          Else
25.             Empty SH
26.    End While
27.    i = i + 1 /*Increase cell level*/
28.    MinLev-dist = mindist(q, L_i);
29. End While
```

Fig. 4 k-NN Search Algorithm

3.3 Incremental k-NN Query Processing

The result of a new registered continuous k-NN query can be computed by the k-NN search algorithm. If the query point is static, i.e. it does not move, its k-NN can be maintained incrementally. The idea of incremental k-NN query processing is to exploit the information obtained from last query processing phase to reduce the cost of query processing.

The query lists in grid cells are required to support incremental processing. If a cell c overlaps the influence region of the query q, the answer of q may be influenced by the movement of objects in c. Accordingly, the identifier of q is inserted into the query list of c. When there is any object in c moves, the incremental k-NN update algorithm is triggered to check whether q's k-NN is changed. In Fig. 3, for example, all cells in L_0, L_1, cell $c_{1,2}$, $c_{1,3}$, and other three cells in L_2 overlap with q's influence region. Therefore, the identifier of q is inserted into the query lists of these cells. By recording the identifiers of queries in the query lists of these cells, incremental result update can be invoked when objects in those cells change their locations.

The incremental update algorithm is proposed to maintain the locations of moving objects in the grid index and the results of k-NN queries. The algorithm is performed while updating the locations of objects in the grid index. It includes three parts. Firstly, the location of objects in grid index is updated. Secondly, if the old position of an object is in a query's influence region, the query is checked for result update. Thirdly, if the new position of the object is in a query's influence region, the query is checked for result update, too. For example, when a moving object o updates its location from position $p_{old} \in$ cell c_{old} to $p_{new} \in$ cell c_{new}, all queries recorded in the query lists of these two cells are checked and updated as follow:

(1) For each query q in the query list of c_{old}, if o is currently in *q.kNN-list*, the algorithm removes o from *q.kNN-list*;
(2) For each query q in the query list of c_{new}, if o's new position p_{new} is in q's influence region, the algorithm inserts o into *q.tlist*. If the number of objects in *q.tlist* is more than k after o is inserted, the object with the farthest distance to q is deleted from *q.tlist*.

```
Algorithm: Incremental Update (GI, QT, $u_o$)
Input: Grid Index GI, Query Table QT,
Location update message $u_o = <o.id, p_{old}, p_{new}>$,
where $p_{old} \in c_{old}$ and $p_{new} \in c_{new}$;
Output: updated QT

1.  If ($c_{old} \neq c_{new}$) then
2.      Delete o.id from object list of cell $c_{old}$;
3.  For each q.id in query list of $c_{old}$ do
4.      If ($o \in$ q.kNN-list) then
5.          Delete o from q.kNN-list;
6.  If ($c_{old} \neq c_{new}$) then
7.      Insert o.id into object list of cell $c_{new}$;
8.  For each q.id in query list of $c_{new}$ do
9.      If ($dist(o, q) \leq$ q.kNN-dis) then
10.         Insert o into q.tlist;
11.             If (q.tlist.size() > k) then
12.                 Remove the farthest object in q.tlist;
```

Fig. 5 Incremental Update Algorithm

By applying incremental update algorithm, all moved objects in current registered queries' influence region are checked. Meanwhile, if the total number of objects in a query's *kNN-list* and *tlist* is more than k, the updated k-NN of the query can be obtained by merging these two lists. The query does not need to be reevaluate after all location update messages are processed by the algorithm. The correctness of the incremental update algorithm can be proved as follow: all moved objects in c_{old} are checked against the queries recorded in c_{old} during step 3 ~ 6. If the moved objects are the results of the recorded queries at latest result update cycle, they are deleted from the queries' *kNN-list* at step 5. If any of those deleted objects still stay in the checked queries' influence region, it is inserted back to the queries' *tlist* at step 11. Moreover, for all new objects move into c_{new}, if their coordinates are in the influence regions of queries recorded in c_{new}, they are also inserted into those queries' *tlist*. If a query's *tlist* has more than k objects, it is safe to delete the farthest object at step 13 since there are at least k candidate objects which are not farther to q than the deleted object. Finally, since all objects which move into the influence regions of registered queries are checked, and the radiuses of the influence regions are not change during incremental update, objects in the *kNN-lists* and *tlists* must be the updated results. Fig. 5 presents the incremental update algorithm.

```
Algorithm: Continuous k-NN Process (GI, QT, OT)
Input: Grid Index GI, Query Table QT,
       Object Table OT;

1.  At each result update cycle do
2.      $U_q$ = Location updates set of moved queries;
3.      $U_o$ = Location updates set of moved objects;
4.      For each query in $U_q$ do
5.          If q is terminated
6.              Delete q from $U_q$ and QT;
7.          Else   /*q is moved or new query*/
8.              Update q's location in QT;
9.              Delete q from query list of cells in GI;
10.         For each update message $u_o$ in $U_o$ do
11.             Incremental Update (GI, QT, $u_o$);
12.         For each query q in $U_q$ do
13.             k-NN Search (GI, QT, q);
14.         For each query q in QT do
15.             Merge objects in q.tlist into q.kNN-list;
16.               If (q.kNN-list.size() < q.k) then
17.                   k-NN re-computation(GI, QT, q);
18.     According to the change of q.kNN-dist, update
        query list of cells in GI
```

Fig. 6 Continuous k-NN Process Algorithm

After all location update messages are processed by the incremental update algorithm, the results of queries are maintained by continuous k-NN process algorithm. The results of moved queries are cleared. The moved queries are reevaluated by the k-NN search algorithm. For each unmoved query q, firstly, objects in *q.tlist* are merged into *q.kNN-list*. If the total number of objects in *q.tlist* and *q.kNN-list* is more than k, only k nearest objects are kept in *q.kNN-list*, and *q.kNN-dist* is updated as well. Secondly, for those queries with less than k objects in *kNN-lists*, the k-NN re-computation algorithm is invoked to compute the results of the queries incrementally. The re-computation is based on current *kNN-dist* and *kNN-list* of corresponding query. Finally, the incremental query process algorithm updates the query lists of cells in the grid index according to updated results of continuous queries. Fig. 6 presents the pseudocode of continuous *k*-NN process algorithm.

The main idea of *k*-NN re-computation algorithm is similar to that of *k*-NN search in Fig. 4, except that it utilized current *kNN-dist* to reduce unnecessary checking of objects. For a give query *q*, it is easy to prove that although *q.kNN-list* currently has less than *k* objects, all these objects must be the part of *q*'s current *k*-NN. Furthermore, there are no other moved objects in the influence region of *q*, since the value of *q.kNN-dist* is not changed during executing incremental update algorithm. Therefore, If a cell or a cell level is fully contained in current influence region of *q* (i.e. the maximum distance between them and *q* is less than current *q.kNN-dist*), objects in them must already contain in *q.kNN-list* and do not need to be checked. The incremental search for query *q* starts from cell level L_0 and expand level by level. For a cell level L_i which is not fully contained in *q*'s influence region, suppose the incompletely covered cells record *m* objects, and *q.kNN-list* stores *l* objects. If $(m + l) \le k$, those *m* objects can be directly inserted into *q.kNN-list* and duplicated objects are deleted. The search range expands until above condition dose not hold. Then, the remaining search procedure is similar to that of *k*-NN search algorithm. Besides the cell filtering criteria in *k*-NN search algorithm, if a cell is fully covered by current query's influence region, it is not inserted into the heap, too. Fig. 7 presents the pseudocode of *k*-NN re-computation.

```
Algorithm: k-NN Re-computation (GI, QT, q)
Input: Grid Index GI, Query Table QT, k-NN Query q;
1.  i = 0;
2.  N(L_i)' = N(L_i) - number of objects in cells fully
    covered by current q's influence region;
3.  While ((q.kNN-list.size() + N(L_i)') ≤ k) do
4.     For all cell c in L_i do
5.        If (c ∩ q's influence region ≠ c) then
    /*check cells not fully in q's influence region*/
6.           For all object o in c do
7.              If (o ∉ q.kNN-list)
8.                 Insert o into q.kNN-list
9.                 If (q.kNN-list.size() = = q.k) then
10.                   Update q.kNN-dist;
11.    i = i + 1; /*Increase cell level*/
12.    N(L_i)' = N(L_i) - number of objects in cells fully
       covered by current q's influence region;
13. End while /*Obtain the initial k-NN objects*/
14. Similar to step 10 ~ 29 in algorithm k-NN Search
```

Fig. 7 *k*-NN Re-computation Algorithm

Compare to the *k*-NN re-computation algorithm of CPM proposed in [6], our approach is more efficient. Firstly, CPM consumes more memory for query processing. CPM assigns each query a visit list and a sorted heap separately, but CkNN only use one sorted heap to process all queries. For example, if the total memory unit for a visit list and a sorted heap is *m*, CkNN only need *m* memory unit to process 5000 queries, while CPM needs 5000 × *m* units! Secondly, CPM re-computes query results from scratch for all moved queries. Although CPM utilizes *visit list* as a cache of visited cells, all objects in those cells still needs to be re-checked. This wastes the computation resource, since most of those objects are already checked in incremental update algorithm. On the contrary, CkNN keeps the objects in *kNN-list* and reuses them. Meanwhile, since the *k*-NN search algorithm of CkNN (invoked form step 14) is more efficient, the overall running time of CkNN is less than that of CPM. Although CPM uses much more memory to build the cache for the query processing algorithm, the performance of CkNN still outperforms CPM. This is also confirmed by experiment evaluation in the next section.

4. Experimental Evaluation

In this section, we evaluate the performance of cell level based continuous *k*-NN process algorithm (CkNN). Since the CPM outperforms all other existing continuous *k*-NN process methods [6]. We only compare the performance of CkNN with CPM.

Table 1: The settings of parameters in experimental evaluation

Parameters	Default	Range
Population of moving objects N_o	100k	10, 50, 100(k)
Location update rate of moving objects at result update cycle U_o	50%	10, 30, 50(%)
Speed of objects / query	medium	Slow, medium, fast
Population of continuous *k*-NN queries N_q	5k	1, 5, 10(k)
Number of nearest neighbors *k*	32	1, 16, 32, 64
Location update rate of queries at result update cycle U_q	30%	10, 30, 50(%)

The data sets in the experiment are generated by the same spatio-temporal data generator used in [6]. The spatio-temporal data is generated in the road map of Oldenburg (a city in Germany), which is same as the map used in [6]. The output of the generator is a set of mobile objects (e.g., cars, trucks, or people) moving on the road map. Moving objects are denoted by their coordinates in the map at successive time stamps. The trip of each object starts from network nodes in the map. After an object reaches a random destination through the shortest path, it then disappears. The speed of the moving objects is classified to three types in the generator, slow, medium, and fast. The slow speed equals 1/250 of the sum of the space extents per time stamp. Medium and fast speeds are 5 and 25 times larger of slow speed, respectively. The continuous *k*-NN queries are generated by the same pattern as mobile objects. They are also objects moving on the same road map, but they will not disappear in the space during the simulation. At each result update cycle, all queries are evaluated by CkNN and CPM. The length of simulation is 100 time stamps. In each experiment, only one parameter is changed, while other parameters keep their default values. All experiments are performed on a Pentium 3.2 GHz CPU with 1GByte memory. Table 1 lists the ranges and default values of parameters used throughout experiments. The location update rate in table 1 is the percentage of moved objects (or queries) at each result update cycle.

In the experiment, we evaluate the effects of five factors on the performance of continuous *k*-NN query processing method. The five factors are the most important issues when evaluate the efficiency and effectiveness of the algorithm for processing continuous *k*-NN queries over moving objects. They are granularity of grid index, number of nearest neighbors, scalability, the location update rate of moving objects and queries, and the speed of objects and queries. Granularity determines the approximation of grid cell. The influence region can be better approximate by smaller cells. However, it costs query processing algorithm more time to check the grid cells. Therefore, continuous query processing algorithm can be benefit from appropriate granularity. For a given granularity, the more nearest neighbors are required by the query, the more objects and cells are needed to be checked. A good algorithm must be less sensitive to the increase in number of NNs. Generally, the enlarged scalability in the workload increases the query processing cost. An efficient algorithm should cost less overall running time on processing the workload. If the location update rate of objects increases, it takes more time for grid index to update the location of objects in the index, and more objects are checked during incremental update of query results. If the location update rate of queries increases (i.e. more queries moved), more queries need to be reevaluated. As a result, the cost of query processing increases. Similarly, if the speed of moving objects/ queries increases, it means that the environment becomes more dynamic. The performance of an effective algorithm must be better under this dynamic situation.

Effect of Granularity

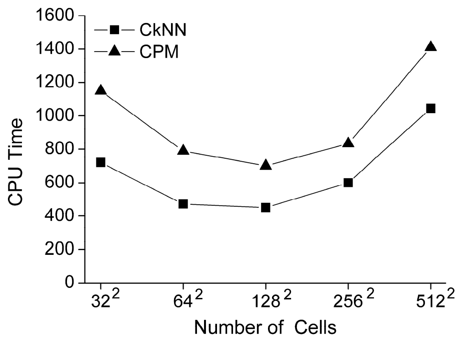

Fig. 8 Overall CPU time versus various grid granularity

At first, we generate the workload with default parameters in table 1. Fig. 8 presents the overall running time of CkNN and CPM under various grid granularity ranging between 32 × 32 and 512 × 512. Fig. 8 illustrates that CkNN outperforms CPM for all grid sizes. As explained in section 3, CPM incurs unnecessary sorting of cells and redundant checking of moving objects. It is showed that the 128 × 128 grid is benefit to both of the query processing algorithms. Therefore, we use this granularity to perform all remaining experiments.

Effects of number of nearest neighbors

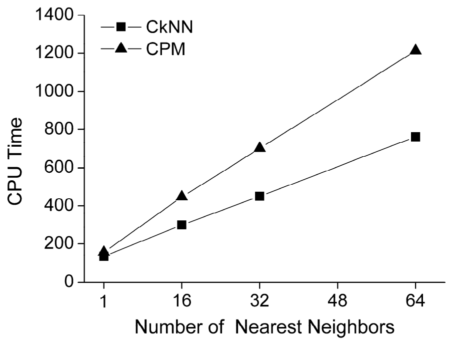

Fig. 9 Overall CPU time versus number of NNs

Fig. 9 measures the overall running time of CkNN and CPM on processing continuous *k*-NN queries which require various number of nearest neighbors. It shows that the overall running time as a linear function of *k* (the numbers of NNs), with other parameters keep default values. CkNN outperforms CPM as *k* increases. This is because: (1) CPM sorts all cells before checking objects in them, while CkNN directly retrieves objects from cells to build initial *k*-NN candidate and sorts fewer filtered cells to refine the query results; (2) the incremental update algorithm of CkNN is more efficient than that of CPM, since CPM checks more objects during this procedure; (3) the performance of *k*-NN re-computation algorithm of CkNN outperforms that of CPM, since CkNN reuses the retrieved objects in *kNN-lists* of queries.

Effects of scalability

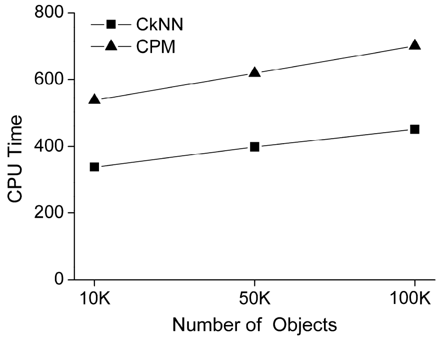

Fig. 10 Overall CPU time versus number of moving objects N_o

Fig. 10 and Fig. 11 illustrate the scalability of the proposed method. We set the generator into constant mode to produce constant objects population N_o and N_q in the simulation. Fig. 10 and Fig. 11 present the effects of

N_o and N_q on the overall running time, respectively. They show that (1) the cost of CkNN and CPM increases linearly to both N_o and N_q; (2) the performance of the two methods are more sensitive to the change of N_q; (3) the performance of CkNN is better than that of CPM under all conditions.

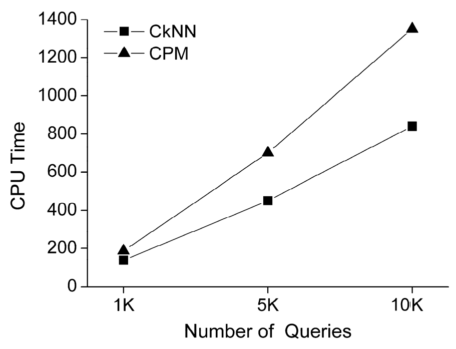

Fig. 11 Overall CPU time versus number of continuous queries N_q

Effect of location update rate

Fig. 12 and Fig. 13 compare the overall running time of CkNN and CPM in processing continuous queries with various location update rate of objects and queries. The cost of query processing by CkNN and CPM increases as location update rate U_o and U_q increase. This illustrates the performance of continuous *k*-NN query processing over moving objects decreases when the agility of objects increases. Meanwhile, it is showed that the performance of CkNN still outperforms CPM under all settings.

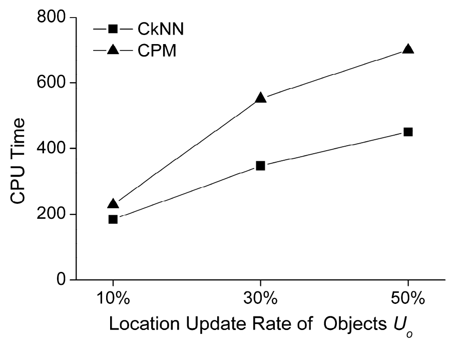

Fig. 12 Overall CPU time versus location update rate of objects

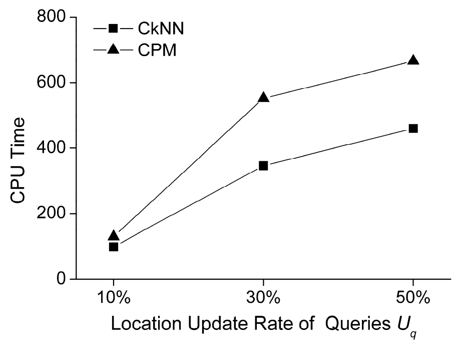

Fig. 13 Overall CPU time versus location update rate of queries

Effects of object/query speed

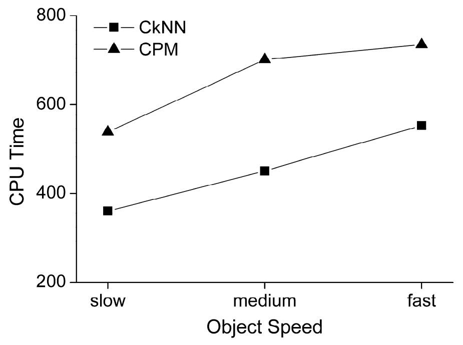

Fig. 14 Overall CPU time versus object speed

Fig. 14 and Fig. 15 compare the overall running time of CkNN and CPM with respect to the object or query speed. Two query processing methods degenerate when objects move fast. The faster objects speed is, the farther they move. This makes more *k*-NN results become invalidate during each result update cycle, and more objects are checked by incremental update algorithm and k-NN re-computation algorithm. Therefore, the query processing cost increases. On the contrary, as showed in Fig. 15, the performance of CkNN and CPM is not influenced by query speed, since these two methods compute *k*-NN of moved queries from scratch. The figures illustrate that CkNN outperforms CPM under all object/query speed.

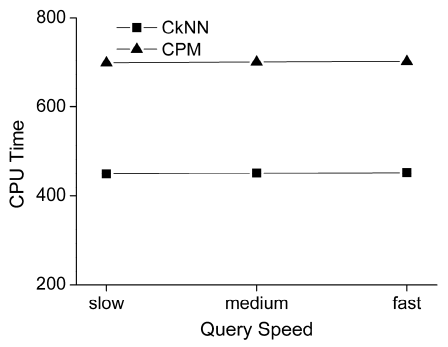

Fig. 15 Overall CPU time versus query speed

5. Conclusions

The paper studies the problem of processing continuous *k*-NN queries in location-dependent application. The challenge of this problem is how to efficiently process continuous *k*-NN queries and location update of moving objects at the same time. We utilize the main memory grid index to store moving objects. Our contribution is an efficient cell level based continuous *k*-NN query processing algorithm, CkNN for short. The algorithm computes the results of new queries and moved queries from scratch by the *k*-NN search algorithm. For existing queries which do not change their locations, CkNN employs the incremental update algorithm to keep the query results up-to-date. For moved queries, *k*-NN search algorithm is applied to reevaluate them. The experimental evaluation demonstrates that CkNN outperforms the current state-of-the-art algorithm CPM in all experiment settings. In the future, we plan to extend our approach to process variations of continuous *k*-NN query, such as reverse *k*-NN query [14].

Acknowledgments

The research was supported in part by the National Natural Science Foundation of China under Grant No.60473075, the Key Project of the Natural Science Foundation of Heilongjiang Province under Grant No.ZJG03-05.

References

[1] Rimantas Benetis, Christian S. Jensen, Gytis Karciauskas, and Simonas Saltenis. Nearest neighbor and reverse nearest neighbor queries for moving objects. In *IDEAS*, pages 44-53, 2002.
[2] Bugra Gedik and Ling Liu. Mobieyes: Distributed processing of continuously moving queries on moving objects in a mobile system. In *EDBT*, pages 67-87, 2004.
[3] Dmitri V. Kalashnikov, Sunil Prabhakar, and Susanne E. Hambrusch. Main memory evaluation of monitoring queries over moving objects. *Distributed and Parallel Databases*, 15(2):117-135, 2004.
[4] George Kollios, Dimitrios Gunopulos, and Vassilis J. Tsotras. Nearest neighbor queries in a mobile environment. In *STDBM*, pages 119-134. Springer, 1999.
[5] Mohamed F. Mokbel, Xiaopeng Xiong, and Walid G. Aref. Sina: Scalable incremental processing of continuous queries in spatio-temporal databases. In *SIGMOD Conference*, pages 623-634, 2004.
[6] Kyriakos Mouratidis, Dimitris Papadias, and Marios Hadjieleftheriou. Conceptual partitioning: An efficient method for continuous nearest neighbor monitoring. In *SIGMOD Conference*, pages 634-645, 2005.
[7] Sunil Prabhakar, Yuni Xia, Dmitri V. Kalashnikov, Walid G. Aref, and Susanne E. Hambrusch. Query indexing and velocity constrained indexing: Scalable techniques for continuous queries on moving objects. *IEEE Transaction on Computers*, 51(10):1124-1140, 2002.
[8] Nick Roussopoulos, Stephen Kelley, and Frédéic Vincent. Nearest neighbor queries. In *SIGMOD Conference*, pages 71-79, 1995.
[9] Simonas Saltenis, Christian S. Jensen, Scott T. Leutenegger, and Mario A. Lopez. Indexing the positions of continuously moving objects. In *SIGMOD*, pages 331-342, 2000.
[10] Zhexuan Song and Nick Roussopoulos. K-nearest neighbor search for moving query point. In *SSTD*, pages 79-96, 2001.
[11] Yufei Tao and Dimitris Papadias. Time-parameterized queries in spatio-temporal databases. In *SIGMOD Conference*, pages 334-345, 2002.
[12] Xiaopeng Xiong, Mohamed F. Mokbel, and Walid G. Aref. Sea-cnn: Scalable processing of continuous k-nearest neighbor queries in spatio-temporal databases. In *ICDE*, pages 643-654, 2005.
[13] Xiaohui Yu, Ken Q. Pu, and Nick Koudas. Monitoring k-nearest neighbor queries over moving objects. In *ICDE*, pages 631-642. IEEE Computer Society, 2005.
[14] Stanoi, I., Riedewald, M., Agrawal, D., Abbadi, A. Discovery of Influence Sets in Frequently Updated Databases. VLDB, 2001.
[15] Ferhatosmanoglu, H., Stanoi, I., Agrawal, D., Abbadi, A. Constrained Nearest Neighbor Queries. SSTD, 2001.

Wei Zhang received the BS degree in computer science, and MS degree in aerospace engineering from Harbin Institute of Technology, China. Currently, he is a PhD candidate in the school of computer science and Technology at Harbin Institute of Technology. His research interests include spatio-temporal databases and mobile computing.

Jianzhong Li is a professor in the school of computer science and technology at Harbin Institute of Technology. His research interests include database, data mining, data warehouse, parallel computing, spatio-temporal databases and mobile computing.

Haiwei Pan received my B.S. and M.S. degrees from Heilongjiang University in 1997 and 2000, respectively. Currently, he is a Ph.D. candidate in school of computer science and technology at Harbin Institute of Technology. His research interests include data mining, medical image mining and text mining.