iSEE: Efficient Continuous K-Nearest-Neighbor Monitoring over Moving Objects

Wei Wu
National University of Singapore

Kian-Lee Tan
National University of Singapore

Abstract

In this paper, we propose iSEE, a set of algorithms for efficient processing of continuous k-nearest-neighbor (CKNN) queries over moving objects. iSEE utilizes a grid index and incrementally updates the queries' results based on moving objects' explicit location update messages. We have three innovations in iSEE: a Visit Order Builder (VOB) method that dynamically constructs a query's optimal visit order to the cells in the grid index with low cost, an Efficient Expand (EFEX) algorithm which avoids unnecessary and redundant searching when updating a query's result, and an efficient algorithm that quickly identifies the cells that should be updated after a query's result is changed. Experimental results show that iSEE achieves a 2X speedup, when compared with the state-of-the-art CPM scheme.

1 Introduction

K-nearest neighbor (k-NN) query is one of the fundamental kinds of queries in spatial-temporal databases. A k-NN query finds the k neighbors that are nearest to the query location. A continuous k-NN (CKNN) query over a set of moving objects is a k-NN query that runs continuously and updates the query result whenever the movement of the objects causes a change to the query's result. CKNN query has many interesting applications, for example, surrounding monitoring and location based services. Moreover, a spatial-temporal database server is expected to handle a huge number of moving objects as well as a large number of continuous queries. Thus, an efficient CKNN processing algorithm is very important.

In this paper, we look at the problem of processing CKNN queries (on a centralized server) based on the location update messages from the moving objects. The moving objects send location update messages to the server and the server keeps the CKNN queries' results up-to-date by processing these messages.

The processing of a CKNN query can be divided into two phases: initial processing and continuous update. Initial processing takes the CKNN query as a snapshot KNN query and finds its initial KNN result. Then in the continuous update phase, the query's result is updated whenever the moving objects' location update messages cause a change to its result. There are two situations in which a query's KNN result will be changed by objects' movement: 1) some objects move nearer to the query point than the query's current k nearest neighbors (kNN set); 2) some objects in the query's current kNN set moves further away from the query. We find that existing algorithms handle the first case gracefully, but do not handle the second case efficiently.

Grid index can be used to index both moving objects and CKNN queries so that for a given object's location update message we can quickly identify the queries that are affected by it. However, existing methods ignored optimizing the index update operation, which in fact is one of the most frequent operations in the system because each time a query's result is changed we need to update the index.

We propose iSEE, a set of algorithms addressing the above mentioned problems to achieve efficient CKNN monitoring over moving objects. We observe that the searching of neighbors in the near cells is redundant when handling the case where some existing nearest neighbors move away. We make updating a query's result more efficient by providing a method to identify and remove the redundant searching. We find that a query's affected cells (where the query should be indexed) form a continuous segment on the query's optimal visit order. Based on this finding, we provide an algorithm that identifies very efficiently the cells in the grid index that need to be updated after a query's result is changed. In iSEE, we also design an algorithm that dynamically constructs a query's optimal visit order to the cells. Using these algorithms, iSEE achieves much better performance than the CPM algorithm [6].

Querying moving objects has received much research interests and efforts. For range queries over moving objects, SINA [5] is a centralized approach; Q-index[7], MQM [3], and MobiEyes [4] are distributed processing schemes. For continuous KNN queries on moving objects, early works [1, 9, 8] made assumptions on the motion patterns of the objects and focused on predictive and approximate query an-

19th International Conference on Scientific and Statistical Database Management (SSDBM 2007)
0-7695-2868-6/07 $25.00 © 2007 IEEE

swering; recent works [13, 12, 6] relaxed these assumptions and proposed algorithms that process CKNN queries based on objects' location update messages and the grid index; disMKNN [10] is a distributed scheme for moving KNN monitoring.

In the next Section, we provide background and motivation of iSEE. Then a detailed description of iSEE is presented in Section 3. In Section 4, we present results of an experimental study. Section 5 concludes this paper.

2 Background and Motivation

2.1 CKNN Query and Grid Index

A continuous k-nearest-neighbor (CKNN) query can be represented as $q(qid, p, k, t)$ where qid is the ID of the query, p is the query location, k is the number of nearest-neighbors the query wants to find, and t is the time period during which the query should be continuously running. Since t only has influence on when the query should be removed from the system, and does not affect the processing strategy, to simplify the discussion, we do not involve t in the remaining discussion and algorithms descriptions.

Given a continuous k-NN query q on a set of moving objects O, the task is to ensure that the query's result set O', which is a subset of O, always satisfies the following conditions: $|O'| = k$ and

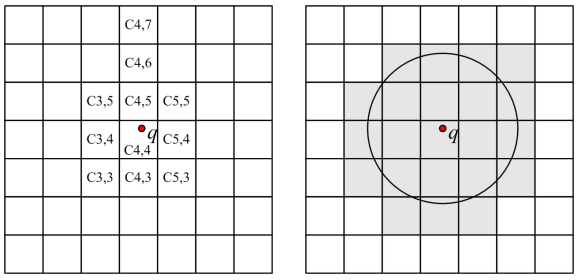
$$\forall\, o \in (O - O')\ dist(q, o) \geq Max\{dist(q, o')|o' \in O'\}$$

Here $dist(q, o)$ means the distance between the query q's location and the object o's location. The first condition ensures that query's result set contains k objects, and the second condition ensures that these k objects are the k nearest ones to q.

We call the distance between a query and its kth nearest neighbor the query's *critical distance*, and denote it as $CD(q)$. $CD(q)$ is the $Max\{dist(q, o')|o' \in O'\}$ in the above second condition. Now, the circle centered at q's location and with radius $CD(q)$ is the border that determines whether an object's movement may cause a change to the q's result set (KNNs). We call this circle the query's *critical circle*, and it is defined as $circle(q, CD(q))$. The critical circle is an imaginary circle, but it makes describing ideas and algorithms much easier.

In this paper, we assume that moving objects explicitly report their location updates to the server. Before receiving a new location update message from a moving object, the server assumes that the object remains at its last reported location. iSEE processes CKNN queries based on the location update messages from the moving objects.

To support efficient CKNN processing (on moving objects), the grid index has been adopted [12, 13]. A grid index divides the whole area into squares (or rectangles) with fixed size. In iSEE, both moving objects and queries are indexed in a grid.



(a) Grid (b) Affected Cells

Figure 1. Grid Index.

Moving objects are indexed in the cells of the grid according to their locations. Each object is mapped to one cell, and each cell has an objects list that contains the objects that fall in this cell. Indexing objects in the grid facilitates the search of nearest neighbors and avoids checking all the objects: objects in a nearer cell (relative to the query location) are nearer to the query than the objects in a cell that is far away to the query.

Queries are indexed so as to quickly identify the set of queries that may be influenced by an object's movement (in fact a location update message). Each query is indexed in the cells that it affects. A cell is affected by a query if the object movement involving[1] this cell may potentially change the query's result.

A query's *affected cells* are the cells that either intersect with or are covered by the query's critical circle. They are denoted as $AC(q)$. A query's affected cells form the minimum set of cells that cover its critical circle. Only the object movement that involve at least one of these cells have the potential to cause a change to the query's result set.

We define *min-distance* between a cell c and a query q as the minimum distance between c and q, and use $dist_{min}(q, c)$ to denote it. Whether a cell is affected by a query can be determined by comparing the min-distance between the cell and the query with the query's critical distance: the cell is affected by a query if (and only if) its min-distance to the query is shorter than the query's critical distance. Thus we can write $AC(q)$ as $AC(q) = \{c|dist_{min}(q, c) < CD(q)\}$. For example in Figure 1(b), the circle is query q's critical circle, and the gray cells are the ones either intersect with or are covered by the critical circle; only the objects whose movement involve the gray cells may potentially cause a change to the query's result.

We index a query by putting a reference to the query into each of its affected cells. Each cell has an *influenced*

[1] An object location update involves a cell if the object was in this cell before the update or the object is in this cell after the update.

queries list that contains all the queries that affect this cell. The influenced queries list of a cell c can be written as $\{q|dist_{min}(q,c) < CD(q)\}$.

After indexing the queries this way, it is very easy to identify for a given location update message the queries whose result may be changed by this message. Let $cell(p)$ denote the cell where location p is in the grid index. A location update message indicates that object o has moved from location p to p'. The cells that are involved in this movement are $cell(p)$ and $cell(p')$. Only the queries that affect $cell(p)$ and the queries that affect $cell(p')$ may be affected by this message. That is, for this message we only need to consider the queries in the following set: $\{q|cell(p) \in AC(q)\} \cup \{q|cell(p') \in AC(q)\}$.

2.2 Visit Order

A query q's *visit order* (called visit list in [6]) to the cells in the grid index is the ordering of cells given by the ascending order of the min-distances between the cells and the query. A query's visit order to the cells gives the order we should follow when searching for nearest neighbors. For example, let the grid shown in Figure 1 be a unit grid, and the location of q be (4.6, 4.8), then the visiting order of q to the cells should be $c_{4,4}, c_{4,5}, c_{5,4}, c_{5,5}, c_{3,4}, \ldots$, because we have $dist_{min}(q, c_{4,4}) = 0$, $dist_{min}(q, c_{4,5}) = 0.2$, $dist_{min}(q, c_{5,4}) = 0.4$, $dist_{min}(q, c_{5,5}) = 0.4472$, and $dist_{min}(q, c_{3,4}) = 0.6, \ldots$.

Following the visit order, when looking for a query's k nearest neighbors, we need to check the next cell for *nearer* neighbors (than the found ones) if and only if the min-distance between the cell and the query is shorter than the current critical distance[2] (i.e., $dist_{min}(q,c) < CD(q)$). For example, if after checking cells $c_{4,4}, c_{4,5}, c_{5,4}, c_{5,5}$ we have found more than k objects and among them the distance between q and its *k*th NN is smaller than 0.6, which is the minimum distance between q and the next cell in visit order, then we are sure that the k nearest neighbor for q we have found so far is its real k nearest neighbors.

An obvious way to construct the visit order for a query q is to compute all the min-distances between q and the cells, and then sort the min-distances to get the visit order. However, this method is not efficient. Instead, we propose an efficient method called VOB (visit order builder) that constructs the visit order dynamically (see Section 3.2).

2.3 CKNN Query Processing

When the server gets a new CKNN query, the query is first processed as a snapshot query, then its result is continuously updated based on the location update messages the server receives from the moving objects. These two steps are called initial process and continuous maintenance.

The objective of the initial process is to find the initial k nearest neighbors for this query. Once we have the visit order for a query, the initial processing of the query is very simple: check the cells for neighbors following the query's visit order, and terminate when we have found (at least) k neighbors and the min-distance between the next cell and the query is larger than the current *k*th distance.

In the continuous maintenance phase we update the query result based on the location update messages the server receives. An object o's location update message will affect a query q's result set in two situations. In the *move-out* situation, object o moves out from q's critical circle: object o was one of q's KNNs but it now moves away and is not necessarily one of q's KNNs anymore. In the *move-into* situation object o moves into q's critical circle: object o was not in q's result set but it now becomes nearer to q than q's current *k*th NN. Note for both cases, there is a special case where the current *k*th neighbor moves. If the *k*th neighbor moves farther from q, it is a move-out case; if the *k*th neighbor moves nearer to q, it is a move-into case. The key difference between move-out scenario and move-into scenario is that: in move-out scenario we have *fewer* than k objects in q's critical circle, while in move-into scenario we have *at least* k objects in q's critical circle.

The move-into scenario is easy to handle. After an object moves into the critical circle, we have *k+1* objects in the circle, thus we should get rid of the existing *k*th neighbor (it now becomes the (*k+1*)th nearest neighbor). Since the *k*th nearest neighbor is very important, we also need to find the new *k*th NN in the updated result set. Observing that the move-into scenario always makes the critical distance smaller, in iSEE we call the procedure that handle move-into scenario the Shrink procedure (see Section 3.4).

For the move-out scenario, we now have only *k-1* objects in the (expanding) critical circle[3], we need to find a new *k*th nearest neighbor. Existing algorithms do not handle this scenario efficiently. Existing algorithms either suffer from unnecessary searching [13, 12] or redundant searching [6]. We propose an efficient algorithm called EFEX (Efficient Expand) for handling the move-out scenario (see Section 3.5)

2.4 Batch Processing

Batch processing is used to achieve higher message processing rate. Rather than update the (affected) queries' results after each location update message, batch processing updates the (affected) queries' results after investigating the overall impact this batch of messages has on the queries.

[2] Before having found at least *k* neighbors for the query, its critical distance is set to infinity.

[3] This also applies to the situation where the current *k*th moves away.

Updating an influenced query q's result after applying a batch of messages is done as follows. Let n_{in} be the number of objects that move into q's critical circle, and n_{out} be the number of objects that move out from q's critical circle. If $n_{in} > n_{out}$, we Shrink the critical circle and get rid of the farthest $(n_{in} - n_{out})$ neighbors from q's result set; if $n_{in} < n_{out}$, we Expand the critical circle to find $(n_{out} - n_{in})$ nearest neighbors for q.

It is important to note that batch processing should be used as a resort rather than a feature. A larger batch should be used only when the arriving rate of location update message is higher than the processing rate the system can achieve with a smaller batch size, because bigger batch size deteriorates the quality of monitoring: higher batch size means the queries' results are updated less frequently. This also shows that a faster CKNN processing algorithm is able to (and will) achieve better monitoring quality.

2.5 Update Index

The grid index needs to be updated when an object moves and when a query's affected cells change.

Updating the index upon the movement of objects is obvious and easy: receiving an object's new location, we check whether it is still in its cell. If yes, we do not need to update the index; otherwise, we remove the object from its (old) cell and put it into its new cell.

We may need to update the grid index after a query's result is updated. Recall that queries are indexed in the cells that they affect. A query's critical distance determines the cells that it affects. When a query's result is changed, its critical distance may change and therefore its affected cells change. After a Shrink, we need to remove the query from the cells that are not affected by it anymore. After an Expand, we need to add the query to the cells that are newly affected by the query.

In the CKNN monitoring system, the results of the queries change frequently due to the movement of the moving objects. Thus updating grid index is one of the most frequent operations in the system. Making this operation efficient is very important for improving the system's performance. Existing algorithms ignored this problem[4].

A natural method is to compare the query's affected cells with its affected cells before the result update, and then handle the Δ affected cells. The cost of this method is linear to the number of cells in the query's affected cells. In this paper, we propose a method called FIU (Fast Index Update) that is linear to the number of cells in the Δ affected cells, i.e., we achieve real incremental update of query index (see Section 3.6).

[4]Some did not mention how they update the indexing of queries, and the others use the obvious method described below.

3 iSEE: EFFICIENT CONTINUOUS KNN MONITORING

In this section we give the details of the data structures and algorithms we use in iSEE for continuous KNN monitoring on moving objects.

3.1 Data Structures

We use an Object Table (OT) to store each moving object's ID and current location. A hash index is built on object IDs so that accessing the objects is fast. In the following discussion, we use $o.id$ and $o.p$ to mean an object o's ID and location respectively.

For each query, the following information is maintained: query ID, query location, k value, result set, critical distance, visit order, affected pointer, priority queue. We use $q.id$, $q.p$, $q.k$, $q.rs$, $q.cd$, $q.vo$, $q.ap$ and $q.pq$ respectively to refer to the above elements of a query q. The affected pointer is used to facilitate index update. The priority queue is used for constructing visit order. A Query Table (QT) is used to store the queries. A hash index is built on queries' IDs so that accessing the queries is fast.

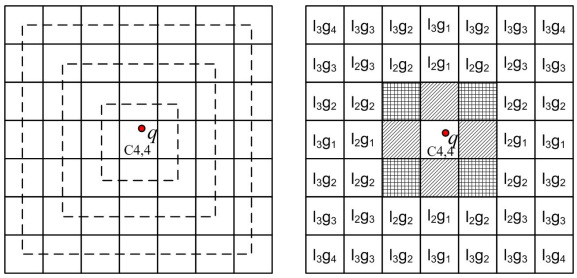
Figure 9(a) illustrates the structure of a query's visit order. Each visit order entry has the following information: the cell's ID (e.g., $C_{4,4}$, $C_{4,5}$), the min-distance from the cell to the query (e.g., 0 for $C_{4,4}$, 0.2 for $C_{4,5}$), the max-distance from the cell to the query (e.g., 1 for $C_{4,4}$, 1.342 for $C_{4,5}$). The visit order of a query is a list of visit order entries that are sorted based on their min-distances. The visit order is built and grown dynamically by our Visit Order Builder (VOB) algorithm.

The query's affected pointer ($q.ap$) points to the last entry (in the visit order) that is affected by the query. It is used to facilitate updating index after the query's result is changed.

A grid index is used to index both objects and queries. For each cell we maintain two lists: an objects list, and an affected queries list. In the objects list are the IDs of the objects whose locations fall in this cell. The grid index is updated when objects move or queries' affected cells change.

3.2 Visit Order Builder (VOB)

As shown in Section 2.2, when looking for nearest neighbors for a query, the optimal order to check the cells in the grid index is to always pick the next un-visited cell that has the smallest *min-distance* to the query point. We note that as long as a query location does not change, this query's visit order to the cells is fixed. However, building a visit order that contains all the cells in the grid index is very inefficient. A fine-grained grid index may have a huge number of cells, but a query will only visit a relatively small number of cells.



(a) Layers (b) Groups

Figure 2. VOB groups.

Moreover, storing a visit order containing all the cells is a waste of memory.

We design an efficient algorithm, VOB (visit order builder), for growing a query's visit order. Using VOB, visit order is grown based on necessity.

We observe that for each query, we can divide the cells in the grid index into different groups such that the cells in a group have similar min-distances to the query. We can then build the visit order for the query from the groups that are nearer to the query.

Let $cell(q)$ denote the cell where the query q is. First, the cells around $cell(q)$ are divided into different levels. The cells adjacent (around) to $cell(q)$ form the first level. The cells around level l form the $(l+1)$ level. For example, in Figure 2(a) the cells that are connected by a dotted line belong to the same level. We define $cell(q)$ as level 0, and level $l+1$ as the cells around level l.

Next, we divide the cells in each level l into $l+1$ groups based on their relative positions to $cell(q)$. In Figure 2(b), the cells in level 1 are divided into two groups, and the cells in level 2 are divided into 3 groups. In the figure, the cells belonging to the same group are filled with the same pattern. Each group has 4 or 8 cells depending on the position of the group. We note that cells in each group have similar min-distances to the query (please refer to [11] for more details).

More formally, suppose $c_{a,b}$ is the cell where q is, i.e. $cell(q)$, then the cells in level l are the cells $c_{i,j}$ such that

$$i = a \pm l, b - l \leq j \leq b + l$$
$$j = b \pm l, a - l \leq i \leq a + l$$

The cells in group g $(1 \leq g \leq l+1)$ of level l are the cells $c_{i,j}$ such that

$$i = a \pm (g-1), j = b \pm l$$
$$i = a \pm l, j = b \pm (g-1)$$

We use L_lG_g to refer to the group g of level l. Level l $(l \geq 1)$ has $8 * l$ cells. When $g = 1$ or $g = l+1$, L_lG_g has 4 cells; otherwise L_lG_g has 8 cells.

```
NextCell(q)
1.  e ← de-queue(q.pq)
2.  IF e is a cell
3.    RETURN e
4.  IF e is a group L_l G_g
5.    FOR each cell c in L_l G_g
6.      en-queue c with dist_min(q, c)
7.    en-queue L_{l+1} G_g with dist_min(q, L_{l+1} G_g)
8.    IF l=g
9.      en-queue L_l G_{g+1} with dist_min(q, L_l G_{g+1}
10.   RETURN NextCell(q)
```

Figure 3. Next cell to grow q's visit order

After dividing the cells into different levels and groups for query q, the groups have the following properties:

- Each cell belongs to one and only one group.
- The cells in the same group have similar min-distances to q. (This property does not hold for small value of l.)
- The min-distance between L_lG_g and q is smaller than the min-distance between $L_{l+1}G_g$ and q.
- The min-distance between L_lG_g and q is smaller than the min-distance between L_lG_{g+1} and q.

Here the min-distance between a group L_lG_g and q is the minimum among the min-distances between the cells in L_lG_g and q, defined as $Min\{dist_{min}(q,c)|c \in L_lG_g\}$, denoted as $dist_{min}(q, L_lG_g)$. Note both $dist_{min}(q,c)$ and $dist_{min}(q, L_lG_g)$ can be computed in constant time based on q's position in $cell(q)$ and the cell's (or group's) relative position to q.

We build the visit order of query q using a priority queue. The element we push into the priority queue is either cell or group. The min-distance between q and the element is used to order the elements in the priority queue. The rules we have for en-queue and de-queue are as follows. When a cell is de-queued, the cell is put to the end of the query's visit order. When a group L_lG_g is de-queued,

- the cells in the group (computed on-the-fly) are pushed into the priority queue;
- group $L_{l+1}G_g$ is pushed into the priority queue;
- if $g = l$, then L_lG_{g+1} is pushed to the priority queue.

In the beginning, $cell(q)$ and L_1G_1 are pushed into the priority queue. Then the procedure *NextCell* shown in Figure 3 is used to grow the query q's visit order.

VOB has two nice properties: 1) we grow the visit order (by calling *NextCell*) only when we need to visit more cells;

InitialKNN(q)

1. $q.rs \leftarrow \emptyset$;
   $q.cd \leftarrow +\infty$;
   $q.vo \leftarrow [\,]$; // empty list
   $q.pq \leftarrow anewpriorityqueue$;
2. en-queue $cell(q)$ and L_1G_1 to $q.pq$
3. $c \leftarrow NextCell(q)$
4. WHILE $q.cd > dist_min(q, c)$
5. add $< c, dist_{min}(q, c), dist_{max}(q, c) >$ to $q.vo$
6. IF $dist(q, 0) < q.cd$
7. add $< o, dist(q, o) >$ to $q.rs$
8. IF $|q.rs| \geq q.k$
9. $q.cd \leftarrow k$th distance in $q.rs$
10. IF $|q.rs| > q.k$
11. get rid of the $(k + 1)$th
12. $c \leftarrow NextCell(q)$
13. en-queue c to $q.pq$
14. put q to QT //query table
15. FOR EACH cell c in $q.vo$
16. put q to c's influenced queries list
17. $q.ap \leftarrow |q.vo| - 1$

Figure 4. Initial Process of q

2) the size of the priority queue is always kept small, thus the en-queue and de-queue operations are very cheap. The number of elements in the priority queue is small because the cells in the same group have similar min-distances to the query, thus most cells in the priority queue will be de-queued before the next group of cells are en-queued. These two properties ensure that the method is efficient in terms of both computational time and memory usage.

In [11], we compared VOB and Conceptual Partitioning (the visit order constructing method used in [6]). Our results showed that VOB is not only 5%-10% more efficient than Conceptual Partitioning, it consumes significantly less memory. Interested readers are referred to [11] for details.

3.3 Initial KNN

The task of the initial processing of a CKNN query is to find the query's initial KNN set. Figure 4 shows the *InitialKNN* procedure. We follow the query's visit order to check the cells for nearest neighbors. During this course we also build up the the query's initial visit order (a query's visit order may be grown in handling move-out scenario). We visit the next cell only if the min-distance between the query and that cell is shorter than the found critical distance.

At the end of *InitialKNN(q)* we put q to the cells that are affected by q (lines 15-16), and initialize q's affected pointer to the end of q's visit order (we will talk more about this in Section 3.6). After the *InitialKNN(q)* all the cells in q's visit order are affected by q. For all the cells c in q's visit order we have $q.cd > dist_{min}(q, c)$, because: a cell is visited in *InitialKNN(q)* only if its min-distance is shorter than the current critical distance; and all the objects in the cell have a longer distance to the query than the cell's min-distance.

Shrink (q)

1. Retain the nearest k objects in $q.rs$
2. $q.cd \leftarrow k$th distance in $q.rs$
3. **UpdateIndex**(q)

Figure 5. Shrink: handle move-into scenario

3.4 Shrink

The Shrink procedure is used to update a query's result when the number of objects that move into the query's critical circle is larger than the number of objects that move out from the query's critical circle. In this case, the *k* nearest ones are retained as the query's new result set, and a new critical distance is found. Another case for invoking Shrink is when the number of move-in objects is the same as the number of move-out objects, but the query's *k*th NN moves inwards. In this case, the new *k*th distance is set as the query's new critical distance. The procedure of $Shrink$ is shown in Figure 5.

After a Shrink, the query's new critical distance become smaller, thus the query's affected cells may be changed, and we need to update the index. This is done by the $UpdateIndex$ procedure.

3.5 Efficient Expand (EFEX)

In a move-out scenario, the number (n_{out}) of objects that move out from a query's critical circle is larger than the number (n_{in}) of objects that move into the query's critical circle, and the query has fewer than k objects in its current critical circle. Our task is to find another $(n_{out} - n_{in})$ nearest neighbors for the query.

The important observations we have for handling the move-out scenarios are: 1) the $(n_{out} - n_{in})$ objects we are looking for are not in the cells that are *totally covered* by the query's current (old) critical circle; 2) when looking for the $(n_{out} - n_{in})$ nearest neighbors, the principle of visit order still applies– follow the visit order to check the cells and terminate when the next cell's min-distance is larger than the current *k*th distance.

Whether a cell is covered by a query's critical circle is checked by comparing the current (old) critical distance and the max-distance between the cell and the query. We use *max-distance* and notation $dist_{max}(q, c)$ to refer to the

```
Expand (q)
 1.  cd_old ← q.cd
 2.  q.cd ← +∞
 3.  voe ← the first entry in q.vo
 4.  WHILE q.cd > voe.dist_min
 5.    IF cd_old < voe.dist_max
 6.      FOR EACH o in the objects list of cell voe.c
 7.        IF o is not in q.rs
     Lines 7-12 of InitialKNN(q)
14.    IF voe is not the end of q.vo
15.      voe ← the next entry in q.vo
16.    ELSE
17.      BREAK
18.  IF voe is the end of q.vo
     Lines 3-14 of InitialKNN(q)
31   UpdateIndex(q)
```

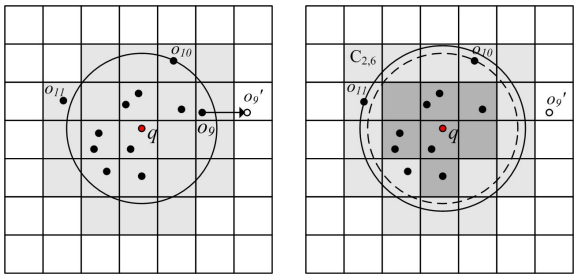
Figure 6. Shrink: handle move-into scenario

maximum distance between query q and cell c. The max-distance between a cell and the query is computed[5] when the cell is put to the query's visit order, and the information is stored in the query's visit order.

Based on these observations, our Efficient Expand (EFEX) algorithm goes as follows: follow the query's visit order and only check the cells that are not totally covered by old critical circle, put the newly found neighbors to the query's result set and terminate when the next cell's min-distance is larger than the *k*th distance in the query's result set. We call this algorithm Expand because after handling the move-out scenario, the query has a larger critical circle, thus basically we expand the query's critical circle during the process. It is efficient because we eliminate redundant searching (by skipping the covered cells) and unnecessary searching (by following the visit order). The algorithmic description of EFEX is presented in Figure 6.

Note that the query's visit order only contains the cells that we have visited before (either in the initial process of the query or in other Efficient Expand of this query), because the query's visit order is grown when necessary. Thus it is possible that after checking all the cells in the query's visit order (line 18), we have not found enough objects or the min-distance of the next cell in the query's priority queue is shorter than the current *k*th distance. In these two circumstances we use VOB to grow the query's visit order until the termination condition is satisfied. After updating the query's result, we need to update the grid index (line 31), because the query's critical distance has changed and therefore its affected cells may change.

[5]Both min-distance and max-distance between a cell and a query can be computed efficiently, because one of the cell's corner points has the min-distance, and one of the cell's corner points has the max-distance. Min-distance and max-distance are stored in visit order because the two distances do not change.



(a) Before Expand (b) After Expand

Figure 7. Example of Efficient Expand.

Figure 7 shows an example of Efficient Expand. Here, q is a continuous 10-NN query. Its result before updating is shown in Figure 7(a). The cells in gray color are the ones that are affected by q. Object o_9, which was in q's critical circle moves out to the position marked as o_9'. This move-out scenario triggers an Efficient Expand on q. Figure 7(b) shows how the Expand works. The cells in darker gray color are the ones that are covered by q's old critical circle (which is depicted using dash line), so they are not checked during the expansion. The other cells in brighter gray color are checked for the new 10th nearest neighbor of q. These cells are checked according to their order on q's visit order. Object o_{11} is found to be q's new 10th nearest neighbor.

3.6 Efficient Index Update (EIU)

Updating index is one of the most frequent operations in iSEE. As discussed in Section 2.5, updating the index as a result of object movement is straightforward. As such, we shall focus on handling updates due to changes in the query result. Recall each time a query's result is updated (either in Shrink or Efficient Expand) we need to update the grid index, because the query is indexed in the cells it affects, and each time the query's result is updated its affected cell may change. To update the index, we must figure out the Δ affected cells after a query's result is changed.

We optimize this operation by utilizing the query's visit order. Recall a query's visit order is an ordering of the cells according to the ascending order of the cells' min-distances to the query. Let $c_1, c_2, \ldots, c_i, c_{i+1}, \ldots$ be the cells on a query q's visit order where c_i is the cell referred by the *i*th visit order entry; they have the property $dist_{min}(q, c_i) \leq dist_{min}(q, c_{i+1})$. Also recall that a cell c is affected by q if and only if $dist_{min}(q, c) < CD(q)$. Thus we find that in q's visit order we can find one cell c_a (here c_a means the cell referred by the ath entry on the visit order) such that:

for all $i \leq a$ we have $dist_{min}(q, c_i) < CD(q)$, and for all $i > a$ we have $dist_{min}(q, c_i) \geq CD(q)$.

This means that in a query's visit order there exists a cell c_a that divides the visit order into two parts: all the cells before c_a, and c_a, are affected by the query, and all the cells after c_a are not affected by the query. We call such a cell c_a the *dividing cell* of the query's visit order. This leads to an interesting finding about a query's affected cells: they are *continuous* on the query's visit order. This property enables us to find the Δ affected cells very efficiently after a query's result is changed.

Let c_a be the dividing cell on q's visit order before q's result is changed. Let cd_{new} be the new critical distance of q, and $c_{a'}$ be the new dividing cell on q's visit order, after q's result is updated. (Note that a and a' are the index numbers.) We have:

- $a' < a$, if $cd_{new} \leq dist_{min}(q, c_a)$;
- $a' = a$, if $dist_{min}(q, c_a) < cd_{new} \leq dist_{min}(q, c_{a+1})$;
- $a' > a$, if $cd_{new} > dist_{min}(q, c_{a+1})$.

More interestingly, if $a' < a$ then the cells in range $(a', a]$ on q's visit order are the cells that are no longer affected by q; if $a' = a$, we do not need to update grid index; if $a' > a$, then the cells in range $(a, a']$ are the cells that are newly affected by the query. Both $(a', a]$ and $(a, a']$ are a continuous segment on q's visit order.

This finding leads to the following method for updating a query's index in the grid after its result is changed. We maintain a pointer that points to q's dividing cell on its visit order. This pointer is called the *affected pointer* ($q.ap$). After a Shrink, we move the pointer to the left along q's visit order until the current pointed cell's min-distance to q is less than the new critical distance, and we remove q from (the influenced queries lists of) the cells that the pointer passed by. After an Efficient Expand, we move the pointer to right as long as the right next cell's min-distance is less than the new critical distance, and we add q to (the influenced queries lists of) the cells that the pointer passed by.

Using this method, a query's *affected pointer* always points to the query's dividing cell on the visit order, and the Δ affected cells are found very efficiently. The cost of this algorithm is proportional to the number of cells in the Δ affected cells. The algorithmic description of this algorithm is shown in Figure 8.

The position of a query's affected pointer is initialized after the query's initial process. At the end of initial process, we put the pointer to the last entry on the query's visit order (line 17 of procedure $InitialKNN$). Figure 9 shows an example of using this algorithm to find Δ affected cells after a query's result is updated (depicted in Figure 7). Before the result update (expand), the query's critical distance is 1.98,

```
UpdateIndex (q)   //update index after q's result is updated
1.  voe ← the visit order entry q.ap points to
2.  IF q.cd ≤ voe.dist_min // Shrink case
3.    WHILE (q.cd ≤ voe.dist_min)
4.      remove q from the iql of cell voe.c
5.      q.ap − −
6.      voe ← the visit order entry q.ap points to
7.  ELSE
8.    WHILE (|q.vo| > (q.ap + 1)) //not at end of q.vo
9.      voe ← the visit order entry q.ap + 1 points to
10.     IF voe.dist_min < q.cd //Expand case
11.       add q to the iql of cell voe.c
12.       q.ap + +
13.     ELSE
14.       BREAK
```

Figure 8. UpdateIndex: update q's affected cells

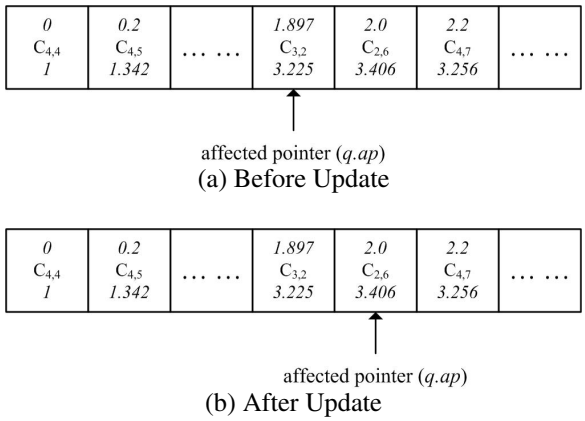


Figure 9. Example of Index Update.

and its affected pointer points to entry of cell $c_{3,2}$. Cell $c_{3,2}$ and the cells before it on the visit order are the ones affected by the query, and the cells after it (from the entry of $c_{2,6}$) are not affected by the query. After the expand, the query's critical distance becomes 2.1. We move the affected pointer rightwards and stop at the entry of $c_{2,6}$, because it is the last one whose min-distance is less than the new critical distance (2.1). Since $c_{2,6}$ is the only cell we passed during the movement, it is the only cell that is newly affected by the query, i.e. Δ affected cells. We update grid index by adding the query to $c_{2,6}$'s influenced queries list.

4 Experimental Evaluation

In this Section, we experimentally study the performance of iSEE and compare it with the state-of-the-art algo-

Table 1. System Parameters

Parameter	Default	Range
No. of objects (no)	50 K	10,20,50,70,100,150 (K)
No. of queries (nq)	2 K	1,2,5,7,10 (K)
k	20	5, 10, 20, 40, 80, 160
No. of cells (nc)	100^2	$32^2, 64^2, 128^2, 256^2, 512^2$
Object speed (v)	medium	slow, medium, fast
Batch size (bs)	100	1, 10, 100, 1000, 10000

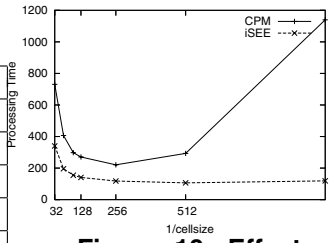


Figure 10. Effect of Grid granularity

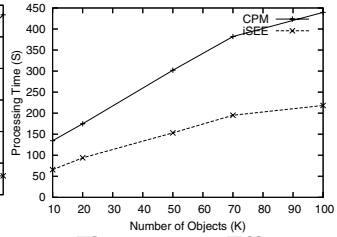


Figure 11. Effect of objects population

rithm(s). In [6], it is shown that CPM outperforms Object-Indexing[13] and SEA-CNN[12] for all problem settings, so it suffices for us to compare iSEE only with CPM.

4.1 System Model

We implemented iSEE and CPM[6] and ran them on the queries and moving objects datasets generated using the Network-based Generator[2] with the Oldenburg map. Queries are randomly generated locations with specified k values. The moving objects datasets are used as the location update message streams.

The system parameters under investigation are summarized in Table 1. Here slow, medium, and fast are the default speed values used in the moving objects generator[2]. Grid indexes of different granularity are used to study the effect of cell size on the algorithms' performance. Batch size is the number of messages in each batch when the location update messages are processed.

In each set of experiments, we vary one system parameter, with the objective to study the impact of this parameter on system (algorithm) performance. We use the processing time as the performance metric. As we shall see, iSEE is about 2 times faster than CPM, and iSEE scales better.

4.2 Effect of Grid Index Granularity

The effect of grid index granularity on the performance of the algorithms in show in Figure 10[7]. The value on the x-coordinate is $1/\delta$ where δ is cell's side size, and the number of cells is $(1/\delta)^2$. We can see that iSEE is about 2 times faster than CPM when $1/\delta$ is less than 512, or by comparing their best performance. With very fine-grained grid index the performance difference between iSEE and CPM is huge. When the cell size gets smaller, the number of cells a query affects becomes larger (grows with speed $(1/\delta)^2$), thus the cost of checking the cells (even most cells are empty) and updating index increases. This is why CPM's performance deteriorates when cell gets too small. In iSEE, we avoid redundant searching and update index efficiently, therefore iSEE performs very well with fine-grained grid index.

4.3 Effect of number of objects

Figure 11 shows that when the population of the objects get bigger, the time the algorithms need to processing the messages gets longer. It is as expected because we get more messages when the number of objects increases. iSEE outperforms CPM and scales better. It is interesting to see that when the number of objects gets larger and larger, the increase speed ($\frac{\Delta t}{\Delta no}$) of processing time slows down. The reason behind this is that when the population of objects becomes larger, the number of objects in each cell increases, thus the number of cells that will be affected by a query gets smaller.

4.4 Effect of k

The processing time increases with k (the number of NNs wanted), but iSEE increases slower, as shown in Figure 12. The almost linear increasing of processing time is quite natural because when k increases we need to find more neighbors and it is more likely a query's result will change. When k increases, the gap between iSEE and CPM gets larger because the query's critical circle increases with k, thus more computation is saved by iSEE with bigger k.

4.5 Effect of Batch Size

As we pointed out, the batch size affects the quality of monitoring, because large batch size hides the number of

[6]We strictly follow the description of CPM in [6], but make the following optimization to its Update Handling procedure. In CPM's original Update Handling pseudo code, each time a batch of messages is processed, the whole query table (QT) is scanned twice. It is inefficient and not necessary, and will incur a lot of overhead when the batch size is small. In our implementation, we collect the set of queries that are affected by the messages during the course of processing them (as we do in iSEE's *HandleObjectUpdates* procedure).

[7]We notice that the curve of CPM shown here is a little different from the one(s) shown in [6]. We guess the reason is that in [6]'s experiments, a very large batch size is used. This also explains why the Update Handling in the CPM simply scans through the whole query table to find the queries that are affected by the messages.

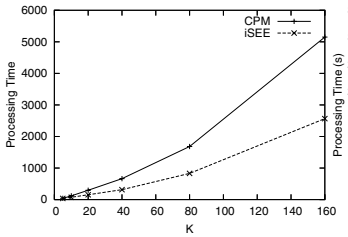


Figure 12. Effect of k

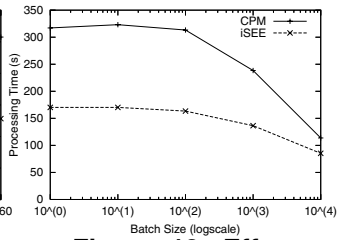


Figure 13. Effect of Batch Size

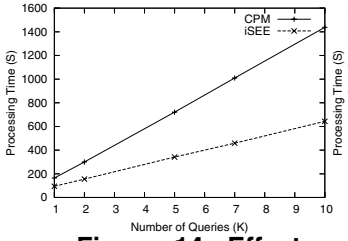


Figure 14. Effect of number of queries

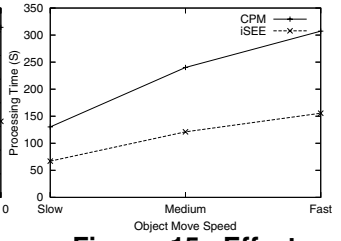


Figure 15. Effect of objects' moving speed

times of result change. In real system, the time the algorithm has to process the messages is fixed: the duration of the monitoring. Thus, if the algorithm can process the messages in the time limit with smaller batch size, the algorithm delivers better quality of monitoring. Note this set of algorithm is conducted by running the algorithms on the messages with fixed batch sizes. Thus the less time an algorithm uses, the better quality of monitoring it is able to deliver. Figure 13 shows the difference between iSEE and CPM's capabilities of supporting small batch sizes.

4.6 Effect of Number of Queries

Figure 14 shows that the processing time increases linearly with the number of queries in the system. This is natural, because the average number of queries each cell has increases linearly with the number of queries, and then the average number of queries each message will affect increases linearly.

4.7 Effect of Moving Speed

The effect of moving objects' speed on the algorithms' performance is depicted in Figure 15. When the objects move faster, it is more likely that the queries' results will be changed, thus we see increases of processing time for both iSEE and CPM with higher speed.

5 CONCLUSIONS

In this paper, we proposed iSEE, a set of algorithms to process CKNN over moving objects efficiently. Specifically, 1) we proposed the VOB algorithm that dynamically constructs a query's visit order to the cells in the grid index; 2) we identified redundant searching that can be eliminated when updating a query's result, and proposed the EFEX algorithm that avoids redundant searching; 3) we pointed out that updating index is one of the most frequent operation in a CKNN monitoring system, and optimized this operation. Experimental results show that iSEE outperforms the CPM scheme.

References

[1] R. Benetis, C. S. Jensen, G. Karciauskas, and S. Saltenis. Nearest neighbor and reverse nearest neighbor queries for moving objects. In *IDEAS*, 2002.

[2] T. Brinkhoff. A framework for generating network-based moving objects. *Geoinformatica*, 6(2):153–180, 2002.

[3] Y. Cai, K. A. Hua, and G. Cao. Processing range-monitoring queries on heterogeneous mobile objects. In *MDM*, 2004.

[4] B. Gedik and L. Liu. Mobieyes: Distributed processing of continuously moving queries on moving objects in a mobile system. In *EDBT*, Heraklion, Crete, Greece, 2004.

[5] M. F. Mokbel, X. Xiong, and W. G. Aref. Sina: scalable incremental processing of continuous queries in spatio-temporal databases. In *SIGMOD*, pages 623–634, Paris, France, 2004. ACM Press.

[6] K. Mouratidis, D. Papadias, and M. Hadjieleftheriou. Conceptual partitioning: an efficient method for continuous nearest neighbor monitoring. In *SIGMOD*, pages 634–645, Baltimore, Maryland, 2005. ACM Press.

[7] S. Prabhakar, Y. Xia, D. V. Kalashnikov, W. G. Aref, and S. E. Hambrusch. Query indexing and velocity constrained indexing: Scalable techniques for continuous queries on moving objects. *IEEE Trans. Comput.*, 51(10):1124–1140, 2002.

[8] K. Raptopoulou, A. Papadopoulos, and Y. Manolopoulos. Fast nearest-neighbor query processing in moving-object databases. *GeoInformatica*, 7(2):113–137, 2003.

[9] Y. Tao and D. Papadias. Time-parameterized queries in spatio-temporal databases. In *SIGMOD*. 2002.

[10] W. Wu, W. Guo, and K.-L. Tan. Distributed processing of moving k-nearest-neighbor query on moving objects. In *ICDE*, 2007.

[11] W. Wu and K.-L. Tan. isee: Efficient continuous k-nearest-neighbor monitoring over moving objects. Technical report, http://www.comp.nus.edu.sg/ g0404403/papers/TR-2006-isee.pdf, 2006.

[12] X. Xiong, M. F. Mokbel, and W. G. Aref. Sea-cnn: Scalable processing of continuous k-nearest neighbor queries in spatio-temporal databases. In *ICDE*, pages 643–654. IEEE Computer Society, 2005.

[13] X. Yu, K. Q. Pu, and N. Koudas. Monitoring k-nearest neighbor queries over moving objects. In *ICDE*, 2005.