Continuous Mutual Nearest Neighbour Processing on Moving Objects in Spatiotemporal Datasets

Shiva Ghorbani, Mohammad Hadi Mobini, and Behrouz Minaei-Bidgoli

***Abstract*—This paper proposed a new algorithm for answering a novel kind of nearest neighbour search, that is, continuous mutual nearest neighbour (CMNN) search. In this kind of query, by providing a set of objects O and a query object q, CMNN continuously returns the set of objects from O, which is among the k_1 nearest neighbours of q; meanwhile, q is one of their k_2 nearest neighbours. CMNN queries are important in many applications such as decision making, pattern recognition and although it is useful in service providing systems, such as police patrol, taxi drivers, mobile car repairs and so forth. In this paper, we have proposed the first work for handling CMNN queries efficiently, without any assumption on object movements. The most important feature of this work is incremental evaluation and scalability. Utilizing an incremental evaluation technique led to a significant decrease in processing time.**

***Index Terms*—Moving objects, nearest neighbor, query processing, spatio-temporal.**

I. Introduction

Nowadays, the abundant use of mobile devices and wireless networks has created new challenges for spatiotemporal applications. Examples of such applications include traffic monitoring and location-aware services. Due to the highly dynamic nature of moving objects, the results of queries in such environments are continuously changing; therefore, these applications require continuous evaluation for ensuring they provide the correct results. Objects and queries move freely and unpredictably in space. They send new location updates to system when they move from one location to another. The number of updates that the system receives is noticeably high and the queries that are issued last for extended time intervals. If some objects change their locations, the answer to a query may become invalid and will need to be computed once again.

A subject that has recently received much attention is the monitoring of continuous queries. “Place”Mokbel, Xiong [1] as proposed by Mokbel is a framework that continuously monitors ranged queries sent by moving objects. Continuous nearest neighbour search with a trajectory assumption for queries is proposed in [2]. Some research [3]-[5] has been conducted on continuous nearest neighbour monitoring without any assumption for objects or query movement. All of the above used a grid for indexing objects and queries. Other studies [6], [7] have used an R-tree- like structure for indexing objects and queries, or have applied it as a secondary index.

This paper studies a specific type of continuous nearest neighbour query on moving objects in spatiotemporal datasets, namely, continuous mutual nearest neighbour (CMNN) search. These types of queries are useful in applications that involve decision making, pattern recognition and data mining. CMNN is also useful in service providing systems, which are continuously, change their locations. These kinds of systems are utilizing in police patrol, cab, mobile car repairs, or any mobile service providers.

Continuous Mutual Nearest Neighbour (CMNN) search is a kind of query in which, by providing a set of objects O and a query object q, CMNN continuously returns the set of objects from O, which is among the k_1 nearest neighbours of q; meanwhile, have q as one of their k_2 nearest neighbours. Having a dataset D of objects, a query object q, and two parameters k_1, k_2, CMNN continuously returns objects $O \in D$ that $O \in NN_{k1}(q)$ and $q \in NN_{k2}(O)$. CMNN should find those objects which are the nearest neighbours of the query point; meanwhile, the query point is a nearest neighbour of answer objects. Despite of usefulness of CMNN, there is not any efficient method for handling this kind of queries. In this paper, we have proposed the first work for handling CMNN queries efficiently, without any assumption on object movements. This paper does not assume any prediction or trajectories regarding the movement of objects and queries. Objects and queries can move freely in the space; by considering a monitoring region for the CMNN query, it becomes possible to process these types of queries incrementally upon updates which are received from moving objects to the system.

CMNN has been used for producing hierarchical clustering trees [8]. The improvement of a k-means algorithm through the use of k-mutual nearest neighbour is addressed in [9]. Studies closely related to the present research include [10], [11]. In [10], an efficient method is proposed for answering mutual nearest neighbour search in spatial databases, where it is assumed that all objects and query points are stationary. This approach utilizes the TPL technique [12] with KNN search to answer mutual nearest neighbour queries. Processing mutual nearest neighbour searches for moving objects by assuming trajectories for objects and queries is addressed in [11]. The present research assumes that all objects and queries have trajectories, and answers the mutual nearest neighbour query with this restriction.

To our knowledge, there is no existing research on processing continuous mutual nearest neighbour searches on

Manuscript received December 21, 2015; revised March 15, 2016.

Shiva Ghorbani is with School of Computer Engineering, Iran University of Science and Technology, Tehran, Iran (e-mail: shiva_ghorbani@comp.iust.ac.ir).

Mohammad Hadi Mobini is with the Department of Computer Engineering, Sharif University of Technology, Tehran, Iran (e-mail: mobini@ce.sharif.edu).

Behrouz Minaei-Bidgoli is with the School of Computer Engineering, Iran University of Science and Technology, Tehran, Iran (e-mail: b_minaei@iust.ac.ir).

moving objects in spatiotemporal datasets without assuming any restriction on object movement or query movement. Therefore, in this paper, we propose a novel solution for solving these types of queries.

The rest of this paper is organized as follows. Section II reviews related researches. Section III presents the characteristics of CMNN and a naïve solution is proposed for answering these types of queries. Section IV presents our method for processing CMNN queries. Section V describes how updates are handled incrementally by our method. Section VI presents our experimental results and findings. In Section VII, we calculate and compare Big-O of the proposed algorithm, as well as a baseline algorithm. Finally, Section VIII concludes this paper.

II. Related Works

In this section, we briefly present nearest neighbour and reverse nearest neighbour search algorithms, as our work is based on these two techniques. There are many index structures available for both NN & RNN indexing, most of which are based on R-tree. R-tree and its derivations are very popular for indexing spatial datasets. However, in the present paper, we used a grid structure for indexing our objects and queries, as it was shown in [13] to be the best choice for managing moving objects in main memory. In the following we describe some NN and RNN algorithms.

A. NN Query

A nearest neighbour query (NN) returns the nearest object O in dataset D that has minimum distance to the query. NN and KNN algorithms have been well-researched by, for example, [14], [15]. The primary approach of these algorithms is using a variant of R-tree structure [16]-[18] for indexing objects. R-tree uses a branch-and-bound technique and some distance metrics with which to prune search regions. Another extension of R-tree is proposed for indexing spatiotemporal databases, known as time parameterized R-tree (TPR-tree) [19]. TPR-tree is an extended version of R-tree that introduces velocity bounding rectangle (VBR) for non-leaf entries. In this method, objects are grouped together according to their positions and velocities, and then the rest of VBR method is handled like R-trees. Although these techniques are popular and well-established, they work well only for snapshot queries. For continuous queries where objects change location frequently and updating the query answers is necessary, these indexing techniques are not useful. Additionally, in the case of continuous queries, we need to update answers frequently; therefore, updating the tree too many times is not efficient [13]. As a result, we applied a grid as an indexing structure for moving objects.

Several approaches for evaluating nearest neighbour in high dimensional space have been proposed [20]-[22]. A Voronoi cell was utilized for answering NN queries, as proposed by Zhang [23]. A Voronoi cell is the area in which the NN of a query will remain the same. The algorithm therefore returns the NN according to a valid time result.

Constrained nearest neighbour search was proposed by Ferhatosmanoglu [24] to answer nearest neighbour queries with range constraints. This algorithm retrieves the NN in a specific and constrained area of space. Additional research has been conducted to answer queries related to constrained nearest neighbour (e.g., [25]).

Studying KNN queries for moving objects was first addressed in [26]. Following on, some studies have conducted on range and KNN queries for moving objects [27]-[29].

Continuous k-nearest-neighbour queries (CKNN) are addressed have been addressed in several studies [4]-[7]. Two main approaches in this context are sampling and trajectory. In the sampling approach, queries are evaluated multiple times. In each time interval or with each query movement, the query should be re-evaluated [4], [5], [7]. In the trajectory approach [2], [6] a trajectory is assumed for a moving query. The CKNN in this approach returns the nearest neighbour of every point on the query line segment. It assumes that objects are stationary and that only the query is moving. When the trajectory of a query changes, the query should be re-evaluated.

The following briefly explains some algorithms that have been proposed for answering continuous nearest neighbour searches for moving objects. In YPK_CNN [29], the query is evaluated in two steps. The query is placed in a cell, C_q. First, the query searches for *k* number of objects in C_q and the cells that are exactly in the next level (the cells that surround C_q). If *k* objects are found in this level, YPK_CNN searches the cells at the next two levels to ensure the correct result.

Proposed by Xiong [5], SEA_CNN is a framework for answering continuously concurrent KNN queries. It focused on scalability and incremental evaluation, and does not assume any module for evaluating the initialization of a newly issued query. Its main goal is monitoring changes that occur during the time that a query monitors the NNs. It incrementally changes the query answer using objects or query movements.

Mouratidis proposed CPM (conceptual partitioning method) [4] for answering continuous nearest neighbour queries efficiently. CPM disposes grid cells into conceptual rectangles that have a direction and level number. It visits the cells in ascending order of minimum distance between a cell and the position of *q* query. CPM initializes a heap by inserting C_q (the cell that contains the query point) and the zero level rectangles. Then, by de-heaping the entries, if the entry is a cell, the algorithm will analyse the objects inside it and update the results. If the entry is a rectangle, CPM en-heaps all the cells belonging to that rectangle, as well as in the next level rectangles with the same direction. The algorithm will continue until the heap is empty or until there is an object whose distance to the query is larger than the distance of KNN. CPM has some methods for handling query updates and multiple object updates.

Cheema proposed CircularTrip [3] in which, given a radius r and the location of a query *q*, it returns a set of cells that intersect with the circle of radius *r* and *q* as the focal point. CircularTrip improves efficiency and space requirements by minimizing the number of cells that need visiting. Like previous method CircularTrip is also using a heap to sort the cells that should be visited in the future.

B. RNN Query

A reverse nearest neighbour (RNN) query issued to the

system returns all objects that have a query as their nearest neighbour. A reverse k-nearest neighbour (RKNN) query is similar to RNN, the difference being that a specified query point q is one of the KNN of object O. Therefore, a RKNN returns the set of points that have q as one of their KNNs. Following on, we summarize some of research on RNN/RKNN, focusing specifically on continuous RNN (CRNN) and explaining it in detail.

Studies that have been conducted on this subject are divided into two main groups. The first is the algorithm-based on pre-computation approach; the second is algorithms without pre-computation. Pre-computation-based algorithms have two steps. First, for any object in the system, the distance between the object and its nearest neighbour should be computed in advance and form a vicinity circle around each object. The vicinity circle of each object has a radius equal to the distance between the object and its nearest neighbour and is centred at that object. After all vicinity circles have been formed, a given query point will be checked with all vicinity circles to see which vicinity circles are affected by the query point. Then, the algorithm returns those objects that have q in their own vicinity circles. Muthukrishnan [30] addressed RNN query using the pre-computation method. RdNN-tree [31] proposed by Yang *et al*. improved on the previous technique by introducing an index structure for answering these types of queries.

Some algorithms do not need pre-computation. Tao [12] proposed a method for reducing the search space by utilizing perpendicular bisectors. The idea of TPL is shown in Fig. 1. First, an object that is a NN of query q is chosen, called p. Then, a perpendicular bisector between query q and object p is divided into two halves H_q (the half plane including q) and H_p (half plane including p). The RNN will be p or another point in H_q, because the other points in H_p are closer to p than q. Like all RNN methods, TPL has a filter and a refinement step. The filter step checks for NN of the query point for drawing a perpendicular bisector and implementing filtering. In the refinement step, the NNs of the final step will be checked to see whether q is its nearest neighbour. If q is nearest neighbour of NNs according to the filtering step, that object will be an answer. TPL works very well in low dimensions.

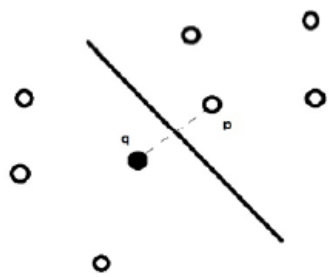

Fig. 1. TPL pruning technique.

Stanoiet proposed a novel method for answering RNN queries called SAA. In [32], Stanoiet *et al*. proved that the number of RNNs in a two dimensional space cannot exceed six. In his algorithm, the two-dimensional space around a query point is divided into six equal partitions (the degree of each partition is 60 degrees), i.e., S_0 to S_5. In the filter step, a nearest neighbour to q is found in each S_i, referred to as O_i. Other objects in this partition cannot be a RNN of q because they are closer to O_i than q. Thus, other objects in that partition are pruned by this method. In this step, six NN of q are defined. Then, in the refinement step, each O_i are verified by checking whether q is their nearest neighbour or not. Fig. 2 illustrates how the algorithm works.

As an illustrative example, consider Fig. 2, in which the space around query q is divided into six sub-regions, S0 to S5. In each sub-region Si, a nearest neighbour O_i is found. As is shown in Fig. 2, O_1 is the nearest neighbour of q in sub-region S_0; however, in the refinement step, this object is omitted from the RNN answer, because O_0 is the NN of O_1. Therefore, q cannot be the NN of O_1. SAA utilizes R-tree for indexing objects. Singh [33] proposed an algorithm for high dimensional space.

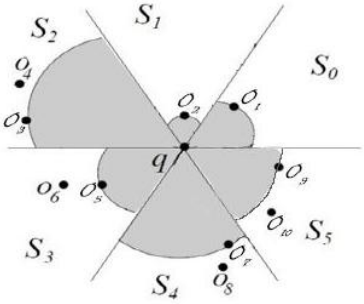

Fig. 2. Illustration of SAA.

Following on, numerous studies have been conducted based on SAA for answering RNN and RKNN for moving objects [34], [35]. Tian Xia [36] utilized SAA with a grid index to answer continuous RNN. Wei Wu [37] proposed an algorithm for answering continuous reverse k-nearest-neighbour monitoring. His approach was based on SAA with the use of a grid index structure. Rei Wu modified the refinement step by using a continuous range query monitoring technique. All of the works noted above employed monochromic RNN. Bichromatic RNN search was proposed by Stanoi [38]. A new method for answering this kind of query on moving objects proposed by Rahmati [39]. Another research on RNN by considering uncertain movement of objects has been performed by Emrich[40]. A recent study on reverse k nearest neighbour has been done by Yang [41].

III. Preliminaries

In this section, we first define MNN and CMNN queries, as well as the characteristics of these queries. Then, we propose a baseline algorithm for answering a CMNN search. This algorithm is used for comparing the performance of our algorithm to the baseline algorithm. In Section V, we show that our algorithm performs better than the baseline algorithm.

As is shown in [32], a query q can only have six reverse nearest neighbours; as such, the number of MNN query results cannot exceed six. A MNN query returns all objects that are in the six nearest neighbours of query q and that are in RNN query q. A CMNN is issued to the system once, but it repeatedly returns the answer by object movements and time passes, so it continuously returns the correct answer via query or the movement of objects. We can formally state that MNN (mutual nearest neighbour query) retrieves the set of objects S that $\forall P \in S$: 1) $P \in NN_{k1}(q)$ and 2) $q \in NN_{k2}(P)$. For the sake

of simplicity, we set k_2 to one, but we did not assume any restriction on k_1. As we know, NN and RNN queries are asymmetric.

Algorithm *Baseline* (q, K_1, K_2)
Input: q: query point, K_1: the number of NNs
Output: S_{result}: the set of MNN query answer

```
Do
    S_result = null
    S_c = perform K1-NN(q) by utilizing min-heap H1
    foreach point C ∈ S_c
        S_t = perform K2-NN(C) by utilizing min-heap H2
        If q ∈ S_t
        S_result = S_result ∪ {C}
    Return S_result
While(updates receive to system)
```

Fig. 3. Baseline algorithm.

This means that if object O_1 is the NN of O_2, it does not mean that object O_2 is also the NN of O_1. However, the MNN query is symmetric. If O_1 is a MNN_{k1k2} of O_2 we can conclude that O_2 is a MNN_{k2k1} of O_1. Another property that MNN has is that the result of the MNN query q is always a subset of k_1-NN(q) and meanwhile, is a subset of k_2-RNN(q). Therefore, due to the definition of MNN query and continuous query, by giving a query point q, k_1 and k_2 parameters, a CMNN query should continuously return the set of objects that are in k_1-NN(q) and k_2-RNN(q). With the movements of an object and time changes, the result of the query should be updated. In this section, we introduce a baseline algorithm for solving this problem. The baseline algorithm is introduced to compare it to our algorithm and to observe how our proposed algorithm is more effective.

A. Baseline Algorithm for CMNN Query

In this section, we propose a simple algorithm that is able to answer CMNN queries.

In this approach, we first perform a k_1-NN search on q to find k_1 objects that are nearer to q. These objects can be candidates for an MNN answer. In other words, the MNN answer is a subset of this candidate set. We call the candidate set S_c. In the next step, we check each candidate object in S_c to verify if it can be a MNN answer. This verification was done by performing a k_2-NN search on candidate object C and because the query should be answered continuously, with each update that the system received, we ran this algorithm once. Thus, it repeatedly sent a correct answer alongside the movements of an object. For performing a KNN search, we can use one of the existing methods, for example YPK algorithm. The baseline algorithm is shown in the Fig. 3.

B. Monitoring Region of CMNN

Continuous queries have monitoring regions. As these types of queries are continuously processed, we should consider a monitoring region for them to enable them to answer queries repeatedly. The monitoring region of nearest neighbour and range query is a simple circle or a rectangle. In a range query, the monitoring region is a regular shape and the distance of objects does not influence the query result. The monitoring region of a CNN is slightly different for a range query. It is usually a circle with the query as its centre and the radius of the circle depends on the distance of the object to the query point.

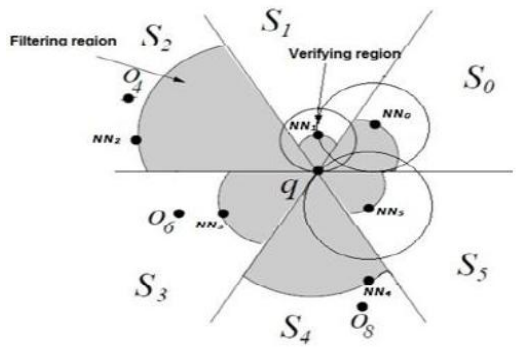

Fig. 4. The monitoring region of a CMNN (filtering & verifying region).

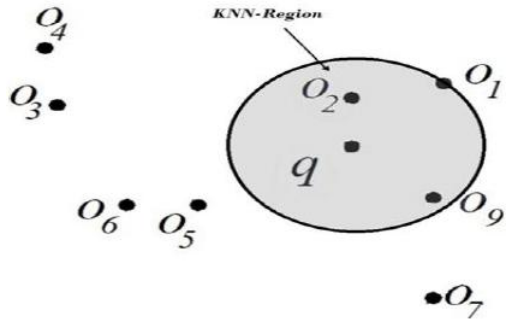

Fig. 5. The monitoring region of a CMNN (KNN region with k=3).

A CRNN query includes two separate monitoring regions, because it consists of two parts, i.e., filtering and verifying. Continuous filtering continuously computes and keeps candidates for RNN. In this step, the nearest neighbours of the query in each partition S_i, which can be a candidate for reverse nearest neighbours, are obtained. In the verifying step, these candidates are checked to verify whether they might be a RNN of the query. In CRNN, both monitoring regions for filtering and the verifying step should be computed for all of the six regions. We also need to monitor the filtering regions continuously for each partition, but the verifying region only monitors for the regions that are candidates for having an MMN query result in the initialization step. Later, by applying operating updates, we only processed those monitoring regions that were influenced by object updates. By utilizing the monitoring regions, we had the possibility for incremental query processing. Fig. 4 and Fig. 5 illustrate the monitoring regions of a CMNN query.

As is shown in Fig. 4, at first the nearest neighbours of the query point are defined in each region S_i, then a KNN search performs on q to find the exact k-nearest neighbours of q. This monitoring region is shown in Fig. 5 and is referred to as the KNN region.

The third part of the monitoring region in CMNN is the verifying region, which forms a circle around the objects that can potentially be an answer to the query. In the following sections, we show how the updates in filtering regions and the verifying region, as well as the result of the CMNN query, are handled and processed incrementally.

IV. CMNN Explanation

In this section we explain our algorithm, for which we used a grid to index for moving objects and queries. In each time interval, objects were mapped to the grid, based on their locations. We submitted queries to the system and it

continuously returned the MNN according to the movements of objects, which influenced the results. Each query point was stored in the system and the space around a query point is divided into six partitions (S_0 to S_5). For each partition Si, we kept (1) the nearest neighbour of that partition called NN_i; (2) $d(NN_i, q)$ which is the distance between NN_i and q; (3) RNN_i, the reverse nearest neighbour of q in S_i. RNN_i was an object nearer to NN_i than q. We computed NN_i for all six partitions at all times. RNN_i was computed only for those partitions where their NN_i was a subset of kNN(q) in the initialization step. If S_i did not contain any objects, the NN_i was set to null and its distance to q was set to infinity.

Algorithm *initMNN* (q, G, k)
Input: q: a query point, G: the Grid index, k: number of NNs
Output: MNNs of q
1. $S_M = S_c = S_{temp} = S_{result} = \emptyset$
2. $P_{reverse}$ = null
3. S_M = FindNearests-6ghesmati(q, G)
4. S_{temp} = FindNearests(q, G, K) // find exact KNN of q
5. $S_c = S_M \cap S_{temp}$
6. Foreach point c $\in S_c$ do
7. $P_{reverse}$ = CheckForReverse(c, q)
8. If($P_{reverse}$== q)
9. Add c to S_{result}
10. Return S_{result}

Fig. 6. The CMNN initialization.

A. CMNN Query Initialization

Initially, we set all NN_i to null and initialized all RNN_i as null, while the distances of NN_i and *q* was set to infinity. The algorithm for initializing CMNN is presented in the following; this algorithm computed the initial MNN result for each query issued to the system.

Algorithm *FindNearests-6ghesmati* (q, G)
Input: q: a query Point, G: the Grid index
Output: 6 nearest neighbors of query point
1. Initialize a queue Q accepting entries of the form (e, key)
2. Divide the space around the query point into six equal regions
3. Add the query cell to Q and set CurrentLevel to 0
4. *repeat*
   a. De-queue the first entry e (grid Cell) from Q
   b. Find out e is intersecting with which partitions S_is
   c. If e is exactly in one partition
      i. If there is a NN_i for that partition and the CurrentLevel exceeds ($Level_i$+2) *continue*
      ii. Else examine all points in e to check whether they can be a NN_i. Update the proper point P with the current NN_i and its level to q ($Level_i$) if it is necessary
   d. If e is in multiple regions S_i
      i. foreach S_i intersecting with e
         1. If there is a NN_i for that partition and the CurrentLevel exceeds ($Level_i$+2) *continue*
         2. Else examine all points that are in partition S_i to be a NN_i. Update the proper point P with current NN_i if it satisfies the conditions
   e. insert the next level grid cells into Q if the grid cell is in a partition that has not a NN_i yet if there isn't any cell with the same level of next level in the Q AND if the grid cell is in a partition that has not a NN_i yet OR the gridcell has level less than ($level_i$ +2)
5. *Until* Q is empty
6. Return nearests

Fig. 7. Algorithm to find nearests in each partition S_i.

The *initMNN* had two filtering steps and one refinement step, in which the objects that were found in the filtering steps were checked for being possible answers. Steps 3 and 4 in Fig. 6 are filtering steps. In these two steps, objects that were potential answers were determined and other objects were pruned. Therefore, the number of objects that had to be processed was small. Verification took place in step 7, in which the candidate objects were checked as being potential answers. The candidate objects were those objects that resulted from both filtering steps. All the objects in candidate sets were checked and those that were query results were returned at the end of algorithm.

Algorithm *checkForReverse* (q, candidate, G)
Input: q: a query point which is considered as nearest neighbor of candidate point, candidate: a NN of query which is considered as query point in this section, G: grid index
Output: R: an arbitrary point that is nearer to NN_i than q
1. Compute the level of q from candidate and name it as $Level_q$
2. Initialize a queue Q that accepts entries of the form (e, key) and then Add the candidate cell to Q
3. While Q is not empty
   a. De-queue the first entry e from Q
   b. Foreach point P in e except candidate, compare d(p, q) with d(candidate, q)
      i. If d(p, candidate) < d(candidate, q) return P as R
   c. Insert the next level gridcells into Q if there isn't any cell with the same level of next level in the Q AND next level < ($Level_q$ +2)
4. Return q As R

Fig. 8. Algorithm to find reverse nearest neighbors.

Algorithm *UpdateMNN* (U, G, Query)
Input: Update of object O that moves from O_{old} to O_{new}, Query: a query point q and the answer of last query result, G: the Grid index
Output: query result after updates
1. First we should check the filtering regions to see whether filtering parts would change or not. By using algorithm ***UpdatingFilterRegion*** (U,Q, G) return S_{temp1} as result
2. Check circle regions by utilizing ***UpdateVerifyingRegion*** (U, G, Q) and Update S_{temp1}
3. Check second filtering region(KNN region) by utilizing ***UpdateKNNFilteringRegion*** (U, Q, K, nearest) return S_{temp2}
4. $S_{temp3} = S_{temp1} \cap S_{temp2}$
5. Foreach item in S_{temp3}
   a. If(RNN_i == q) Add S_{temp} to S_{result}
6. Return S_{result}

Fig. 9. Algorithm to update query answer.

For each query *q* issued to the system, we first divided the space around *q* into six regions and found a nearest neighbour of *q* in each partition (the algorithm for this process is illustrated in Fig. 7). In the next step, we performed a kNN search on *q* to find the exact *k* nearest neighbours of *q*. Finally, we only checked those objects that were present in both filtering parts. In other words, we only checked objects that were the result of intersection in the two filtering steps. Alternatively, we might say that only the NN_is that were in kNN(*q*) were checked. Through this method, no matter how large *k* was, in the worst case scenario, we only needed to check six objects to see whether *q* was their NN.

The first filtering method is illustrated using an example in Fig. 4. The second filtering step, which finds the exact *k* nearest neighbours of the query, utilized YPK algorithm. When the above two filtering steps were applied, we found the objects that were returned by the first filtering step; meanwhile, these objects were also returned by the second filtering step. In other words, we were able to establish an intersection between the results of the two filtering steps and only checked those objects that were returned by these two filtering methods for being a MNN. The algorithm or checking objects to be a MNN is shown in Fig. 8. Using this algorithm, we checked the candidate objects to find out whether their nearest neighbour was *q* or not.

We created a circle centred at each candidate object and with radius d (*q*, candidate). If an arbitrary object was found between this candidate object and the query point, this candidate object was not considered to be a MNN answer. This algorithm returned the first object it found as a reverse nearest neighbour of *q*; it did not need to find the exact RNN of *q*. The first object that could break the condition was considered as the nearest neighbour of that candidate object.

V. Incremental Processing of CMNN

This section illustrates how to obtain CMNN query results incrementally. By updating the locations of objects, we incrementally updated the result sets; this does not require performing the entire algorithm at each time interval. We had two types of updates: (1) updates due to query movement; (2) updates that were the result of object movement. We treated the query movements by deleting the query and all the items that are related to query from the system. Thereupon we should submit the query *q along* with its new location and then perform the *InitMNN* algorithm on it.

Algorithm *UpdatingFilterRegion* (U, q, G)
Input: U: Update of object O that moves from O_{old} to O_{new}, q: query point, G: Grid index
1. *If* object O is one of the NN_is
   a. *If* the distance of new position of this object (O_{new}) to q, is less than the distance of its last position to q, and O_{new} is lied in the same region as O_{old}, replace O_{new} with NN_i then compute the reverse object for O_{new} by using **CheckForReverse** (O_{new}, q, levelOfO_{new}toQ)
   b. *Else* set NN_i as null for that specific region and then search for new NN_i in only that region. And compute the reverse object for O_{new} by using **CheckForReverse** (O_{new}, q, levelOfO_{new}toQ)
2. *If* object O moves in one of the six filtering regions
   a. Compute the region of O_{new}
   b. *If* there is a NN_i for that specific region and the distance of O_{new} to query point q is less than distance NN_i to q, replace the NN_i with O_{new} and compute the reverse object for new NN_i by using **CheckForReverse** (O_{new}, q, levelOfO_{new}toQ)
   c. *If* there isn't any NN_i for that region, consider O_{new} as NN_i, then compute the reverse object for O_{new} by using **CheckForReverse**(O_{new}, q, levelOfO_{new}toq)

Fig. 10. Algorithm to update filtering region.

The movement of objects may influence the answers of CMNN query in two ways. The result of filtering parts and verifying regions may have been changed by the movement of objects. The filtering step consisted of two different parts that are processed separately. We call the first filtering part filtering regions (Fig. 4) and the second filtering part the kNN region (Fig. 5). Each update received to the system was processed incrementally in the filtering region, kNN region and verifying region. Once all these updates were applied, an intersection between the kNN region and filtering region was applied to establish the final results. The pseudo-code for updating the results is shown in Fig. 9. In the following we explain how the updates are handled in these three sections.

A. Updates in Filtering Region

Updates in filtering regions were handled similar to handling updates explained in [36], though there were some differences. Updates in the filtering region were handled incrementally, without the need for processing all parts of the six regions from the start. In the worst case scenario, with each update that the system received, the algorithm needed to search for nearest neighbour in only one region.

Algorithm *UpdateVerifiynRegions* (U, G)
Input: U: the Update of object O that moves from O_{old} to O_{new}
1. for all six regions (S_0 to S_5)
   a. If there is a NN_i in that region (S_i), check to see whether O_{new} is closer to NN_i than its RNN or not
      i. If there isn't any RNN_i for that region, compare d(NN_i , O_{new}) with d(NN_i, q)
         a. if d(NN_i, O_{new}) < d(NN_i, q) then consider O_{new} as RNN_i in S_i
      ii. Else If d(O_{new}, NN_i) < d(NN_i, RNN_i) then replace RNN_i with O_{new}
2. If O_{old} was not null
   a. for all six regions (S_0 to S_5)
      i. If there is a NN_i in that region (S_i) AND RNN_i equals to O_{old} **CheckForReverse**(q, NNi)

Fig. 11. Algorithm to update verifying region.

Algorithm *UpdateKNNFilteringRegion* (U, q, K, Nearests)
Input: U: an Update of object movement from O_{old} to O_{new}, q: query point, K: number of nearest neighbors of q that should be checked, Nearests: k of objects that are nearer to q
1. If object O is a member of the Nearests
   a. If d(O_{new}, q) > kth Nearests // leave the search region
      i. Apply the KNN search algorithm from the scratch to find K nearest objects to query point q
   b. If d(O_{new}, q) < kth Nearests do nothing
2. If object O doesn't exist in Nearests and d(O_{new}, q) < kth Nearests
   a. Remove kth item from Nearests and then insert object O in Nearests

Fig. 12. Algorithm to update kNN region.

When an update occurred in the filtering region, there were two aspects that needed to be observed: (1) whether one of the NN_is had moved from its previous location; (2) some objects had moved within a filtering region.

Step 1 of the algorithm illustrates the first instance (a NN_i changes its previous location). When this happens, the NN_i moved closer to the query point; in the region located prior to an update, the updated object (O_{new}) had been considered as a NN_i. If the NN_i moved away from the query point or toward a different region, we needed to search for a new NN_i in that $region_i$ (the region of O_{old}).

The second situation showed the case where the O_{new} of any object may have influenced a filtering region. This could cause a new object to enter the system.

In all cases, after updating the filtering region and its new NN_i, the algorithm had to compute the verifying region accordingly. Therefore, the CheckForReverse algorithm (previously explained) was used to compute new regions and verify them.

B. Updates in Verifying Region

In this part we show how an update will impress verifying regions. Verified regions will change when an object exits from it or when an object enters it. Therefore, when an update is received in the system, the algorithm should check its O_{old} (its previous location) to see whether it exits from some verified regions, as well as check whether its new location impresses on verified regions and if so, which ones. The details of the algorithm are shown in Fig. 11.

C. Updates in KNN Region

The kNN region was first calculated in the initialization section. During time intervals, the results of this part may change due to the movement of objects. This part describes how these updates are managed incrementally in the kNN monitoring region.

When an object update is received in the system, three

cases exist that can influence changes in the kNN region: 1) object O enters the kNN region; 2) Object O leaves the kNN region; 3) object O moves inside the kNN region.

The algorithm handles updates as follows: if object O was not in the kNN, the result and distance of O and *q* is smaller than distance *k*th NN to *q*. The algorithm omits the *k*th NN from the answer and adds the new object O to the result, and sort the objects according to their distance. Step 2 of the algorithm controlled this case. Step 1 of the algorithm illustrates cases 2 and 3.

VI. Experiments

In this section, the efficiency of our proposed CMNN query processing algorithms are evaluated and compared with the baseline algorithm. The datasets used for observing performance was generated by the GSTD framework using random data and Gaussian data distribution. The metric that we observed in this study was processing time. All algorithms were implemented in C++ and the experiments were conducted on a PC with an AMD Athlon 3Ghz CPU and 4 GB primary memory running Windows 7 Ultimate. The effects of the following parameters were studied: number of moving objects, number of movements and the value of k and size of the grid cell. Datasets parameters have been shown in TABLE I.

TABLE I: Dataset Parameters

Parameter	Default	Range
Number of objects	10k	1,10,20,30,40,50,60,70,80,90,100(k)
Number of movements	5k	1,5,10,20,30,40,50,100(k)
Value of K	5	1,2,4,6,8,10,12,14,16,18,20
Grid cell size	0.2	0.08, 0.1, 0.2, 0.4, 0.8, 1, 1.5, 2, 2.5, 3

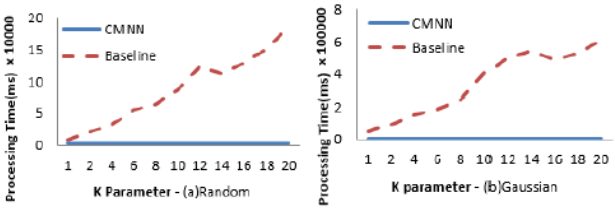

Fig. 13. Effect of *k* parameter on processing time.

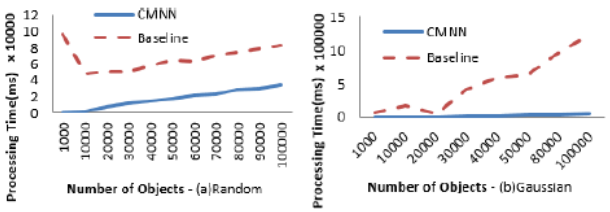

Fig. 14. Effect of number of objects on processing time.

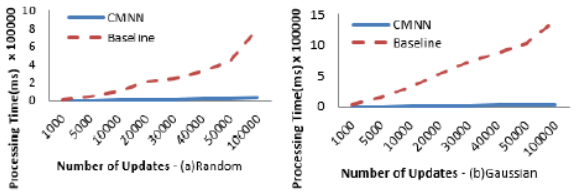

Fig. 15. Effect of number of movements on processing time.

Effect of *k*: when we increased parameter *k*, the processing time did not change significantly in the CMNN algorithm. As verifying this step of the CMNN algorithm does not depend on *k* and in the worst case scenario, on the initializing step, the CMNN algorithm only checked six objects for having the query as their NN. With each update, only those objects that were in one of the NN_is and also in the kNN of the query were checked to see whether they had a query as their nearest neighbour.

Effect of number of objects: when the number of objects was bigger, both the baseline and CMNN developed higher processing times. We differentiated the number of objects and the number of updates from one another; by increasing the number of objects, only the initialization time of the CMNN algorithm changed. As is shown in Fig. 14, the CMNN algorithm performed better than its baseline counterpart.

Effect of the number of movements (updates): Fig. 15 depicts the effect of the number of movements of an object on processing time. Increasing the number of updates increased the processing time of both Baseline and CMNN, but not to the same extensive degree. This was because all the updates in our algorithm were processed incrementally. Fig. 15 shows the effect of the number of movements on processing time.

Effect of size of grid cell: we changed the size of grid cells to uncover the effect of grid cell on processing time. The results are depicted in Fig. 16.

By changing the size of grid cells, the number of objects in each grid cell also changed; thus, changing grid cell size affected processing time. If the number of objects in each grid cell was numerous, the time needed for processing each grid cell increased.

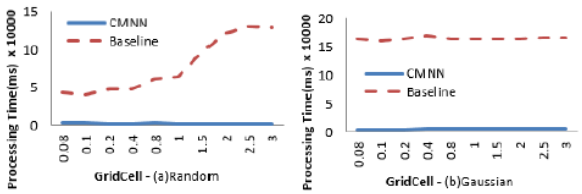

Fig. 16. Effect of size of grid cell on processing time.

VII. Conclusion

In this paper, we studied the problem of continuous mutual nearest neighbour monitoring on moving objects. We also proposed a basic algorithm for solving the problem of continuous mutual nearest neighbour monitoring. We then compared our algorithm to the Baseline algorithm. We showed that CMNN query processing can be divided into continuous filter, continuous KNN and continuous verification. We handled the updates (which are received to the system by object movements) incrementally. Our proposed method was suitable for answering queries involving data streams. We did not assume any restrictions on data input and objects were processed separately. In other words, unlike snapshot queries that process all objects within a time interval, in our proposed algorithm, the query is executed when an object update is received into the system. The experiment results illustrated the performance of our algorithm in terms of its efficiency.

References

[1] M. F. Mokbel, X. Xiong, W. G. Aref, and S. E. Hambrusch, "PLACE: A query processor for handling real-time spatio-temporal data

streams," in *Proc. the Thirtieth International Conference on very Large Data Bases*, vol. 30, 2004, VLDB Endowment.

[2] Y. Gao, B. Zheng, G. Chen, Q. Li, and X. Guo, "Continuous visible nearest neighbor query processing in spatial databases," *The VLDB Journal*, vol. 20, no. 3, pp. 371-396, 2011.

[3] M. A. Cheema, Y. Yuan, and X. Lin, "Circulartrip: An effective algorithm for continuous kNN queries," *Advances in Databases: Concepts, Systems and Applications*, 2007, Springer, pp. 863-869.

[4] K. Mouratidis, D. Papadias, and M. Hadjieleftheriou, "Conceptual partitioning: An efficient method for continuous nearest neighbor monitoring," in *Proc. the 2005 ACM SIGMOD International Conference on Management of Data*, 2005.

[5] X. Xiong, M. F. Mokbel, and W. G. Aref, "Sea-cnn: Scalable processing of continuous k-nearest neighbor queries in spatio-temporal databases," in *Proc. 21st International Conference on Data Engineering*, 2005.

[6] Y. Tao, D. Papadias, and Q. Shen, "Continuous nearest neighbor search," in *Proc. the 28th International Conference on very Large Data Bases*, 2002, VLDB Endowment.

[7] H. Xiao, Q. Li, and Q. Sheng, "Continuous K-nearest neighbor queries for moving objects," *Advances in Computation and Intelligence*, 2007, Springer, pp. 444-453.

[8] A. K. Jain, R. P. W. Duin, and J. Mao, "Statistical pattern recognition: A review," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2000, vol. 22, no. 1, pp. 4-37.

[9] C. Ding and X. He, "K-nearest-neighbor consistency in data clustering: Incorporating local information into global optimization," in *Proc. the 2004 ACM Symposium on Applied Computing*, 2004, ACM.

[10] Y. Gao, B. Zheng, G. Chen, and Q. Li, "On efficient mutual nearest neighbor query processing in spatial databases," *Data & Knowledge Engineering*, 2009, vol. 68, no. 8, pp. 705-727.

[11] Y. Gao, B. Zheng, G. Chen, Q. Li, C. Chen, and G. Chen, "Efficient mutual nearest neighbor query processing for moving object trajectories," *Information Sciences*, 2010, vol. 180, no. 11, pp. 2176-2195.

[12] Y. Tao, D. Papadias, and X. Lian, "Reverse kNN search in arbitrary dimensionality," in *Proc. the Thirtieth International Conference on very Large Data Bases*, vol. 30, 2004.

[13] D. Šidlauskas, S. Šaltenis, and C. W. Christiansen, "Trees or grids?: Indexing moving objects in main memory," in *Proc. the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, 2009, ACM.

[14] A. Papadopoulos and Y. Manolopoulos, "Performance of nearest neighbor queries in R-trees," *Database Theory*, 1997, Springer, pp. 394-408.

[15] N. Roussopoulos, S. Kelley, and F. Vincent, "Nearest neighbor queries," *ACM Sigmod Record*, 1995, ACM.

[16] N. Beckmann, H. P. Kriegel, R. Schneider, and B. Seeger, "The R*-tree: An efficient and robust access method for points and rectangles," vol. 19, 1990.

[17] A. Guttman, "R-trees: A dynamic index structure for spatial searching," vol. 14, 1984, ACM.

[18] T. Sellis, N. Roussopoulos, and C. Faloutsos, "The R+-tree: A dynamic index for multi-dimensional objects," 1987.

[19] S. Šaltenis, "Indexing the positions of continuously moving objects," *Encyclopedia of GIS*, 2008, Springer, pp. 538-543.

[20] S. Berchtold, C. Böhm, D. A. Keim, and H. P. Kriegel, "A cost model for nearest neighbor search in high-dimensional data space," in *Proc. the sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1997, ACM.

[21] P. Indyk, "Nearest neighbors in high-dimensional spaces," 2004.

[22] E. Kushilevitz, R. Ostrovsky, and Y. Rabani, "Efficient search for approximate nearest neighbor in high dimensional spaces," *SIAM Journal on Computing*, 2000, vol. 30, no. 2, pp. 457-474.

[23] J. Zhang, M. Zhu, D. Papadias, and Y. Tao, "Location-based spatial queries," in *Proc. the 2003 ACM SIGMOD International Conference on Management of Data*, 2003, ACM.

[24] H. Ferhatosmanoglu, I. Stanoi, and D. Agrawal, "Constrained nearest neighbor queries," *Advances in Spatial and Temporal Databases*, 2001, Springer, pp. 257-276.

[25] M. Hasan, MA. Cheema, W. Qu, and X. Lin, "Efficient algorithms to monitor continuous constrained k nearest neighbor queries," *Database Systems for Advanced Applications*, 2010, Springer.

[26] G. Kollios, D. Gunopulos, and V. J. Tsotras, "Nearest neighbor queries in a mobile environment," *Spatio-Temporal Database Management*, 1999, Springer.

[27] D. V. Kalashnikov, S. Prabhakar, S. E. Hambrusch, and W. G. Aref, "Efficient evaluation of continuous range queries on moving objects," *DEXA*, 2002, Springer.

[28] K. L. Wu, S. K. Chen, and P. S. Yu, "Incremental processing of continual range queries over moving objects," *IEEE Transactions on Knowledge and Data Engineering*, 2006, vol. 18, no. 11, pp. 1560-1575.

[29] X. Yu, K. Q. Pu, and N. Koudas, "Monitoring k-nearest neighbor queries over moving objects," in *Proc. 21st International Conference on Data Engineering*, 2005.

[30] F. Korn and S. Muthukrishnan, "Influence sets based on reverse nearest neighbor queries," *ACM SIGMOD Record*, 2000.

[31] C. Yang and K. I. Lin, "An index structure for efficient reverse nearest neighbor queries," in *Proc. 17th International Conference on Data Engineering*, 2001.

[32] I. Stanoi, D. Agrawal, and A. E. Abbadi, "Reverse nearest neighbor queries for dynamic databases," *ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, 2000.

[33] A. Singh, H. Ferhatosmanoglu, and A. Ş. Tosun, *High Dimensional Reverse nearest Neighbor Queries*, 2009.

[34] R. Benetis, C. S. Jensen, and G. Karčiauskas, "Nearest neighbor and reverse nearest neighbor queries for moving objects," in *Proc. Database Engineering and Applications Symposium*, IEEE.

[35] R. Benetis, C. S. Jensen, G. Karĉiauskas, and S. Ŝaltenis, "Nearest and reverse nearest neighbor queries for moving objects," *The VLDB Journal*, 2006, vol. 15, no. 3, pp. 229-249.

[36] T. Xia and D. Zhang, "Continuous reverse nearest neighbor monitoring," in *Proc. the 22nd International Conference on Data Engineering*, 2006.

[37] W. Wu, F. Yang, C. Y. Chan, and K. L. Tan, "Continuous reverse k-nearest-neighbor monitoring," in *Proc. 9th International Conference on Mobile Data Management*, 2008. IEEE.

[38] I. Stanoi, M. Riedewald, D. Agrawal, and A. E. Abbadi, "Discovery of influence sets in frequently updated databases," *VLDB*, 2001.

[39] Z. Rahmati, V. King, and S. Whitesides, "(Reverse) k-nearest neighbors for moving objects," in *Proc. the Seventh International Conference on Motion in Games*, 2014, ACM.

[40] T. Emrich, HP. Kriegel, and N. Mamoulis, "Reverse-nearest neighbor queries on uncertain moving object trajectories," *Database Systems for Advanced Applications*, 2014, Springer.

[41] S. Yang, M. A. Cheema, X. Lin, and W. Wang, "Reverse k nearest neighbors query processing: experiments and analysis," in *Proc. the VLDB Endowment*, vol. 8, no. 5, pp. 605-616, 2015.

Shiva Ghorbani was born in Iran, in 1986. She received the B.S. degree in computer engineering from the central branch of Azad University of Tehran, Iran, in 2008, the M.Sc. degree in software engineering from the Iran University of Science and Technology, in 2013.

Mohammad Hadi Mobini was born in Iran, in 1986. He received the B.S. degree in computer engineering from the Shahid Beheshti University of Tehran, Iran, in 2010, the M.Sc. degree in software engineering from the Sharif University of Technology, in 2012.

Behrooz Minaei-Bidgoli received the B.S. degree in mathematics for computer science from Qom University of Tehran, Iran, the M.Sc. degree in computer engineering from the Iran University of Science and Technology and the Ph.D. degree in computer science and engineering from the Michigan State University.

He is currently an assistant professor in the School of Computer Engineering in Iran University of Science and Technology.

He received scholarships for the School of Computer Science and Engineering University of Michigan, U.S. in 2001. He was second place graduated of the University of Science and Technology, Tehran, Iran in 1997. He ranked first graduating undergraduates from the University of Qom, Iran in 1990.