A Unified Algorithm for Continuous Monitoring of Spatial Queries

Mahady Hasan, Muhammad Aamir Cheema, Xuemin Lin, Wenjie Zhang

The University of New South Wales, Australia
{mahadyh,macheema,lxue,zhangw}@cse.unsw.edu.au

Abstract. Continuous monitoring of spatial queries has gained significant research attention in the past few years. Although numerous algorithms have been proposed to solve specific queries, there does not exist a unified algorithm that solves a broad class of spatial queries. In this paper, we first define a *versatile top-k query* and show that various important spatial queries can be modeled to a versatile top-k query by defining a suitable scoring function. Then, we propose an efficient algorithm to continuously monitor the versatile top-k queries. To show the effectiveness of our proposed approach, we model various inherently different spatial queries to the versatile top-k query and conduct experiments to show the efficiency of our unified algorithm. The extensive experimental results demonstrate that our unified algorithm is several times faster than the existing best known algorithms for monitoring constrained k nearest neighbors queries, furthest k neighbors queries and aggregate k nearest neighbors queries.

1 Introduction

With the availability of inexpensive mobile devices, position locators and cheap wireless networks, location based services are gaining increasing popularity. Some examples of the location based services include fleet management, geo-social networking (also called location-based networking), traffic monitoring, location-based games, location based advertisement and strategic planning etc. Due to the popularity of these services, various applications have been developed that require continuous monitoring of spatial queries. For example, a person driving a car may issue k nearest neighbors (kNN) query to continuously monitor k closest restaurants. Similarly, a taxi driver might issue a query to continuously monitor the passengers that are within 5 Km of his location. Cabspotting[1] and Zhiing[2] are two examples of such applications.

Driven by such applications, continuous monitoring of spatial queries has received significant research attention in the past few years. For example, several algorithms have been proposed to answer kNN queries [14, 22, 20], range queries [8, 12, 18, 3], constrained kNN queries [7, 9] and aggregate kNN queries [14]. Although various algorithms have been proposed to solve each of these spatial

[1] http://cabspotting.org/faq.html
[2] http://www.zhiing.com/how.php

queries, to the best of our knowledge, there does not exist a unified algorithm that solves all the above mentioned queries. In this paper, we propose a unified algorithm to monitor a broad class of spatial queries including the above mentioned spatial queries.

In Section 3.1, we first define a *versatile top-k query* and then show that various spatial queries can be modeled to the versatile top-k query (Section 3.2). Given a set of objects, a versatile top-k query reports k objects with the lowest scores. The score of each object is computed by using a *versatile scoring function* $vsf()$ (the properties of a versatile scoring function are formally defined in Section 3). Various spatial queries can be modeled to the problem of a versatile top-k query by choosing a suitable scoring function. For example, a kNN query can be modeled to a versatile top-k query by choosing a scoring function such that $vsf(o) = dist(o, q)$ where $dist(o, q)$ returns the Euclidean distance between the object o and a reference point q (called query point in spatial queries).

We present an efficient algorithm to continuously monitor versatile top-k queries. Our unified algorithm can efficiently monitor any spatial query that can be modeled to versatile top-k queries by defining a suitable scoring function. To monitor any of these spatial queries, the only change we need to make in the implementation is to add a new scoring function for that specific spatial query. The unified algorithm then uses this scoring function to monitor the spatial query.

To show the effectiveness of our approach, we choose various inherently different spatial queries and model these queries to versatile top-k queries. Then, we conduct experiments to show the efficiency of our unified algorithm. More specifically, we show the performance of our algorithm for monitoring constrained kNN queries, furthest k neighbor queries and aggregate kNN queries.

Below we summarize our contributions in this paper.

- We define versatile top-k queries and show that various spatial queries can be modeled to a versatile top-k query by choosing a suitable scoring function.
- To the best of our knowledge, we are first to present a unified algorithm to efficiently monitor various spatial queries. We prove that our algorithm is optimal in number of grid cells it visits to monitor the spatial queries.
- We conduct extensive experiments to study the performance of our unified algorithm for monitoring various inherently different spatial queries. The experimental results show that our unified algorithm outperforms existing best known algorithms for these specific queries.

2 Background

2.1 Preliminaries

In this sectiom, we formally define few inherently different spatial queries.

k nearest neighbors query. A kNN [10, 16, 11, 14, 22, 20] query returns the k closest objects from the query point q. Given a set of objects O and a query

point q, the kNN query returns a set N of k objects such that for each $n_i \in N$ the $dist(n_i, q) \leq dist(o', q)$ where $o' \in O - N$.

kNN queries have a wide range of applications. For example, a person may issue a kNN queries to find k closest restaurants to his location.

Constrained k nearest neighbors query. A constrained kNN query [7, 9] returns k objects closest to the query point q among the objects that lie inside a constrained region CR. Given a set of objects O, a query point q and a constrained region CR, the constrained kNN query returns a set N containing k objects such that for each $n_i \in N$, $dist(n_i, q) \leq dist(o', q)$ where $o' \in O - N$ and both o' and n_i lie in the constrained region CR.

Consider that a user wants to find k post offices closest to his location from a suburb named Randwick. He may issue a constrained kNN query where the constrained region corresponds to the suburb Randwick.

Furthest k neighbors query. A furthest k neighbors query [17, 6, 1] returns k furthest objects from the query point q. Given a set of objects O and a query point q, the furthest k neighbors query returns an answer set N containing k objects such that for each $n_i \in N$, $dist(n_i, q) \geq dist(o', q)$ where $o' \in O - N$.

Consider that a town planner wants to establish a service center in a town. Before establishing this service center, the town planner may need to find the furthest service user to identify the maximum range the service would need to cover.

Aggregate k nearest neighbors query. Given a set of objects O and a query set Q with m numbers of instances $\{q_1, ..., q_m\}$, the aggregate kNN query[21, 15, 13] returns k objects with the minimum aggregate distance from Q. Let $aggdist(o, Q)$ be the aggregate distance of an object o from Q. An aggregate kNN query returns an answer set N containing k objects such that for each $n_i \in N$, $aggdist(n_i, Q) \leq aggdist(o', Q)$ where $o' \in O - N$. Below we define the aggregate distance functions for some common aggregate kNN queries.

1. Sum-aggregate kNN query uses $aggdist(o, Q) = \sum_{q_i \in Q} dist(o, q_i)$.
2. Max-aggregate kNN query uses $aggdist(o, Q) = \max_{q_i \in Q}(dist(o, q_i))$.
3. Min-aggregate kNN query uses $aggdist(o, Q) = min_{q_i \in Q}(dist(o, q_i))$.

Consider that a group of friends wants to meet at a restaurant such that the total distance traveled by them to reach the restaurant is minimum. A sum-aggregate kNN query ($k = 1$) returns the location of such a restaurant.

2.2 Related Work

[22, 20, 14] are some notable techniques that use grid-based index to monitor spatial queries. Conceptual Partitioning Monitoring (*CPM*) technique [14] organizes the grid cells into conceptual rectangles and assigns each rectangle a direction and a level number. The direction is *R, D, L, U* (for right, down, left, up) and the level number is the number of cells in between the rectangle and q as shown in Fig. 1.

CPM first initializes an empty min-heap H. It inserts the query cell c_q with key set to zero and the level zero rectangles (R_0, D_0, L_0, U_0) with the keys set to minimum distances between the query q and the rectangles into H. The entries are de-heaped iteratively. If a de-heaped entry e is a cell then it checks all its objects and updates $q.kNN$ (the set of kNN for the query q) and $q.dist_k$ (the distance of current k^{th}NN from q). If e is a rectangle, it inserts all the cells inside the rectangle and the next level rectangle in the same direction into the heap H. The algorithm stops when the heap becomes empty or when e's distance is greater than $q.dist_k$.

The update of each object is handled as follows. (1) Incoming update: CPM removes the kth NN and inserts the object in $q.kNN$. (2) Outgoing update: CPM removes the object from $q.kNN$ and finds the next NN by visiting the remaining entries in the heap. In case the query moves, CPM starts from the scratch. CPM outperforms both YPK-CNN [22] and SEA-CNN [20]. CPM can also be extended to solve few other spatial queries.

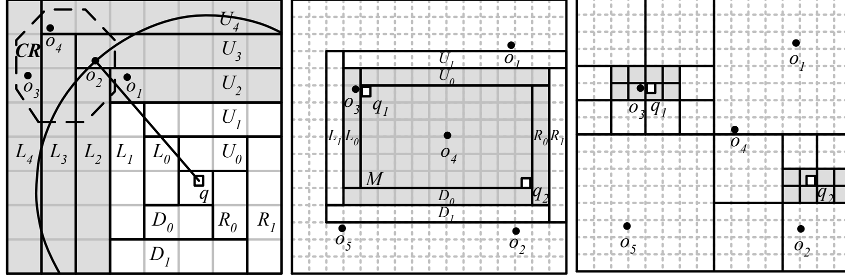


Fig. 1. CPM Constrained NN **Fig. 2.** Cells accessed by CPM **Fig. 3.** Cells accessed by an optimal algorithm

CPM can be used to answer continuous *constrained kNN* queries by making a small change. More specifically, the algorithm inserts only the rectangles and the cells that intersect the constrained region into the heap. Figure 1 shows an example where the constrained region is a polygon R. The constrained NN is o_2 and the rectangles/cells shown shaded are inserted into the heap by CPM.

CPM can also be extended to answer *Aggregate kNN* queries. In case of a conventional kNN query, the algorithm starts with the query cell c_q. For aggregate kNN query, the algorithm computes a rectangle M such that all query instances lie in M. At the initial phase, the algorithm inserts the rectangle M and the zero level rectangles in the heap with their aggregated distance $aggdist(e, Q)$ (e.g., $\min_{q_i \in Q} dist(e, q_i)$) from the query set Q. For instance, in the Fig. 2 the min-aggregate NN is o_3 and all rectangles shown in solid lines are inserted in the heap to compute o_3. Please note that the algorithm inserts all the shaded cells into the heap. Note that CPM inserts many un-necessary cells into the heap. Fig. 3 shows the minimum number of cells (shown shaded) that are needed to be inserted in

heap by an optimal algorithms. We show that our approach significantly reduces the number of cells inserted in the heap.

CircularTrip [5] and iSEE [19] also efficiently monitor kNN queries. CircularTrip visits the cells around query point round by round until all NNs are retrieved. On the other hand iSEE computes a visit order list around the query point to efficiently answer the kNN query. However, extension of these algorithms for other spatial queries (e.g., aggregate kNN query) is non-trivial.

3 Problem definition

Let p be a point and R and R_c be two hyper-rectangles in a d-dimensional space $\mathbb{R}^d$. If R contains R_c (i.e., R_c is inside the hyper-rectangle R) then R_c is called the child of R. Consider a function $f(p)$ that returns the score of a given point p. An upper bound score $S^U(R)$ of a hyper-rectangle R is defined as,

$$S^U(R) = \max_{p \in R}(f(p))$$

where, $p \in R$ denotes a p that lies in the hyper-rectangle R. Similarly, the lower bound score $S^L(R)$ is defined as,

$$S^L(R) = \min_{p \in R}(f(p))$$

Versatile Scoring function: A function $f()$ is called a versatile scoring function iff $S^U(R) \geq S^U(R_c)$ and $S^L(R) \leq S^L(R_c)$ for any R and R_c where R_c is a child rectangle of R. We denote the versatile scoring function as $vsf()$. The versatile score of a given point p is denoted as $vsf(p)$.

Consider a function $f(p) = dist(p, q)$ where $dist(p, q)$ is the Euclidean distance[3] between the point p and a given point q. Hence, the upper bound score $S^U(R)$ is the maximum Euclidean distance between the rectangle R and the fixed point q. Similarly, the lower-bound score $S^L(R)$ is the minimum Euclidean distance between the rectangle R and the fixed point q. Note that $f(p) = dist(p, q)$ is a versatile scoring function.

3.1 Versatile top-k queries

Consider a set of objects $O = \{o_1, \ldots, o_N\}$ where o_i denotes the spatial location of ith object. Also, consider a versatile scoring function $vsf()$ to compute the score of the objects. A top-k query returns a set of k objects $N = \{n_1, \ldots, n_k\}$ such that $vsf(n_i) \leq vsf(o')$ for any $n_i \in N$ and any $o' \in O - N$. Hence, top-k query returns k objects having smallest scores.

In this paper we study the continuous monitoring of top-k query where the top-k results are updated with the changes in the datasets. We follow *timestamp* model where the results are required to be updated after every t time units.

[3] Other L_p distance metrics can also be used.

3.2 Modeling spatial query to top-k query

We can model various spatial queries to a versatile top-k query by defining a suitable versatile scoring function. The versatile scoring functions for some popular spatial queries are given below.

k nearest neighbors queries:

$$vsf(o) = dist(o, q)$$

Here, the $dist(o, q)$ is the Euclidean distance between an object o and the query point q.

Furthest k neighbors queries:

$$vsf(o) = -dist(o, q)$$

Please not that the object furthest from the query q has the smallest score. Hence, the further objects are preferred in this case.

Aggregate k nearest neighbors queries:

Below we define the scoring functions for Sum, Max and Min aggregate k nearest neighbors queries, respectively.

i) Sum-Aggregate k nearest neighbors queries: $vsf(o) = \sum_{q_i \in Q} dist(o, q_i)$

ii) Max-Aggregate k nearest neighbors queries:$vsf(o) = \max_{q_i \in Q}(dist(o, q_i))$

iii) Min-Aggregate k nearest neighbors queries: $vsf(o) = \min_{q_i \in Q}(dist(o, q_i))$

Constrained kNNs queries:

$$vsf(o) = \begin{cases} dist(o, q), & \text{if } o \text{ lies inside the constrained region } CR; \\ \infty, & \text{otherwise.} \end{cases}$$

Note that we can also define the versatile scoring functions for other queries like constrained furthest neighbors query and constrained aggregate kNNs query etc.

Next, we define the versatile scoring function to model another spatial query which is not essentially a top-k query. This demonstrates that our proposed unified algorithm can be applied to answer several other queries that are not top-k queries.

Circular Range queries: Given a set of objects O, a query point q and a positive value r. A circular range query [8, 12, 3] returns every object $n \in O$ that lies within distance r of the query location q (i.e., every object such that $dist(n, q) \leq r$). We call such query a circular range query because the search space is a circle around the query point q with the radius r. Below we define the versatile scoring function for the circular range query.

$$vsf(o) = \begin{cases} 1, & \text{if } dist(o, q) \leq r; \\ \infty, & \text{otherwise.} \end{cases}$$

Here, r is the radius of the circular range query.

To model the circular range query to a versatile top-k query we need to make the following small changes: i) every object with score equal to kth object's score

must also be reported: ii) an object with score ∞ must not be reported. Note that we if the range contains more than k objects then all the objects inside the range are reported. On the other hand, if the range contains less than k objects then objects outside the range are not reported.

4 Technique

4.1 Conceptual Grid Tree

In this section, we briefly describe the conceptual grid tree which we introduced in [4] and later used to answer other spatial queries in [3, 9]. The conceptual grid tree is the backbone of our approach. First, we briefly describe the grid index and then we describe the conceptual grid tree which is a conceptual visualization of the grid index.

In a grid based index ,the whole space[4] is divided into a number of cells, where the size of each cell is $\delta \times \delta$. Hence, the extent of each cell in a dimension is δ. A particular cell is denoted as $c[i, j]$ where i is the column number and j is the row number. The lower left cell of the grid is $c[0, 0]$. An object o with the position (x, y) is located into the cell $c[\lfloor x/\delta \rfloor, \lfloor y/\delta \rfloor]$. I.e., a cell $c[i, j]$ contains all the objects with x-coordinate in the range $i.\delta$ to $(i + 1).\delta$ and y-coordinate in the range $j.\delta$ to $(j + 1).\delta$.

In our proposed conceptual grid tree structure we assume a grid that consists of $2^n \times 2^n$ cells[5]. The grid is treated as a conceptual tree where the *root* contains all $2^n \times 2^n$ grid cells. Each intermediate entry e in a level l (for root $l = 0$) is recursively divided into four children of equal sized rectangles such that each child of an entry e contains $x/4$ cells where x is the number of cells contained by the intermediate entry e. I.e., if an entry e at level l contains $2^{n-l} \times 2^{n-l}$ cells then each child of the entry e will contain $2^{n-l-1} \times 2^{n-l-1}$ cells. Every leaf level entry contains four grid cells.

The root, intermediate entries and the grid cells are shown in Fig. 4. In Fig. 4 the grid size is 4×4 (i.e., $2^2 \times 2^2$ grid cells). Hence, the root contains all $2^2 \times 2^2$ cells. An intermediate entry with level 1 contains $2^{2-1} \times 2^{2-1}$ cells (i.e., four cells).

Please note that the Grid-tree is just a conceptual visualization of the grid and it does not exist physically (i.e., we do not need pointers to store entries and its children). In Fig. 4 the rectangles with dotted lines are considered as conceptual structure and the rest are physical structure. Therefore, root and the intermediate entries are conceptual only and they are not stored in the memory. To retrieve the children of an entry (or root), we divide its rectangle into four equal sized rectangles such that each child has side length $d/2$ where d is the

[4] For the ease of demonstration of our algorithm we use two dimensional space, although our technique can be applied to higher dimensional space.

[5] If the grid size is not $2^n \times 2^n$, it can be divided into several smaller grids such that each grid is $2^i \times 2^i$ for $i > 0$. For example, a 8×10 grid can be divided into 5 smaller grids (i.e., one 8×8 grid and four 2×2 grids)

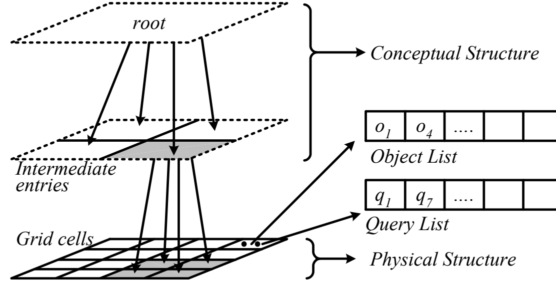


Fig. 4. Conceptual Grid Tree Structure

side length of its parent. A rectangle with side length equal to δ (the width of a grid cell) refers to a cell $c[i, j]$ of the grid.

4.2 Unified Algorithm

Initial computation Most of the spatial queries algorithms that can be applied on other tree structure (e.g., R-tree) can easily be applied on the conceptual grid tree. The advantage of using this grid tree over previously used grid based access methods is that if an intermediate entry of the tree lies in the pruned region then none of its cells are accessed.

Algorithm 1 **CGTree-based Unified Initial Computation**

Input: q: query point with the versatile scoring function($vsf()$); k: an integer
Output: top-k query results

```
1: q.score_k = ∞; q.kA = φ; H = φ
2: Initialize a min-heap H with root entry of the conceptual grid tree
3: while H ≠ φ do
4:    de-heap an entry e
5:    if S^L(e) ≥ q.score_k then
6:       return q.kA
7:    if e is a cell in the grid then
8:       update q.kA and q.score_k by the objects in e
9:    else
10:      for each of the four children c do
11:         if S^L(c) ≤ q.score_k then
12:            insert c into H with key S^L(c)
13: return q.kA
```

The initial computation of the unified algorithm using the Conceptual Grid-Tree is presented in Algorithm 1. The main idea is similar to that of applying BFS search [11] on R-tree based data structure. Specifically, the algorithm starts by inserting the root of the Grid-tree into a min-heap. The algorithm iteratively

de-heaps the entries. If a de-heaped entry e is a grid cell then it visits the cell and updates $q.kA$ and $q.score_k$ where $q.kA$ is the answer set and $q.score_k$ is the kth smallest score of objects in $q.kA$ (line 8). If $|q.kA| < k$ (i.e, the size of the answer set is less then k) then $q.score_k$ is set to infinity. Please recall that the width of a cell is δ. So, the algorithm checks the width of each entry e to identify whether e is a grid cell or not (line 7).

If the de-heaped entry e is not a grid cell, then the algorithm inserts its children into the heap H with their lower bound scores (lines 10 to 12). The algorithm terminates when the heap becomes empty (line 3) or when a de-heaped entry e has its lower bound score $S^L(e) \geq q.score_k$ (line 5). This guarantees the correctness of the algorithm. This is because any cell c for which $S^L(c) \geq q.score_k$ cannot contain an object that has a score smaller than $q.score_k$ (and cannot be the answer for this reason). When the de-heaped entry e has its lower bound score $S^L(e) \geq q.score_k$, every remaining entry e' in the heap H has its lower bound score $S^L(e') \geq q.score_k$ because the entries are accessed in ascending order of their lower bound scores.

Continuous Monitoring Before we present the continuous monitoring algorithm, we introduce the data structure that is used for efficient update of the results.

The system stores a query table and an object table to record the information about the queries and the objects. An object table stores the id and location of all objects. The query table stores the query id, query location, the answer set $q.kA$ and the *cellList* (cells that the query has visited to retrieve all objects in the answer set $q.kA$).

Each cell of the grid stores two lists namely *object list* and *query list*. The object list of a cell c contains the object id of every object that lies in c. The query list of a cell c contains the id of every query q that has visited c (by visiting c we mean that it has considered the objects that lie inside it (line 8 of Algorithm 1)). The query list is used to quickly identify the queries that might have been affected by the object movement in a cell c.

Handling a single update: Assume that an object o reports a location update and o_{old} and o_{new} correspond to its old and new locations, respectively. The object update can affect the results of a query q in the following three ways;

1. *internal update:* $vsf(o_{old}) \leq q.score_k$ and $vsf(o_{new}) \leq q.score_k$; clearly, only the order of the answer set may have been affected, so we update the order of $q.kA$ accordingly.
2. *incoming update:* $vsf(o_{old}) > q.score_k$ and $vsf(o_{new}) \leq q.score_k$; this means that o is now a part of $q.kA$. Hence, o is inserted in $q.kA$.
3. *outgoing update:* $vsf(o_{old}) \leq q.score_k$ and $vsf(o_{new}) > q.score_k$; i.e., o is not part of the answer set anymore. Therefore, we delete o from $q.kA$.

The complete update handling module: The update handling module consists of two phases. In the first phase, we receive the object updates. For each object update, we reflect its effect on the results according to the three scenarios

Algorithm 2 **Continuous Monitoring**

Input: location updates
Output: $q.kA$
Phase 1: *receive updates*
1: **for** each object update o **do**
2: Affected queries $Q_{aff} = c_{o_{old}}.q_list \cup c_{o_{new}}.q_list$
3: **for** each query q in (Q_{aff}) **do**
4: if internal update; update the order of $q.kA$
5: if incoming update; insert o in $q.kA$
6: if outgoing update; remove o from $q.kA$
Phase 2: *update results*
7: **for** each query q **do**
8: if $|q.kA| \geq k$; keep top k objects in $q.kA$ and update $q.score_k$
9: if $|q.kA| < k$; expand $q.kA$
10: **return** $q.kA$

described earlier. In the second phase, we compute the final results. Algorithm 2 presents the details.
Phase 1: First, we receive the object updates and for each object update, we identify the queries that might have been affected by this update. It can be immediately verified that only the queries in the query lists of c_{old} and c_{new} may have been affected where c_{old} and c_{new} denote the old and new cells of the object, respectively. For each affected query q, the update is handled (lines 3 to 6) as mentioned previously (e.g., internal update, incoming update or outgoing update).
Phase 2: After all the updates are received, the results of the queries are updated as follows; if $q.kA$ contains more than k objects in it (more incoming updates than the outgoing updates), the results are updated by keeping only the top k objects. Otherwise, if $q.kA$ contains less than k objects, we expand the search region so that $q.kA$ contains k objects. The expansion is similar to the Algorithm 1 except the following changes. Any entry e that has $S^U(e) \leq q.score_k$ are not inserted into the heap. This is because such entries have already been explored. The stopping criteria is same as the initial computation i.e., we stop when a de-heaped entry e has $S^L(e) \geq q.score_k$.

If a query changes its location the versatile score become invalid. Hence, the results are computed by calling the Algorithm 1 (i.e., we compute the result for the query from the scratch).

Proof of optimality and correctness Before we prove the optimality, we define two terms; *accessing* and *visiting* a cell. We say that a cell has been accessed if the algorithm inserts it in the heap (e.g., line 12 of Algorithm 1). If a cell is de-heaped from the heap and the algorithm retrieves the objects in this cell, we say that the cell has been visited by the algorithm (e.g., line 8 of Algorithm 1). Please note that the cost of visiting a cell is usually significantly higher than the cost of accessing a cell.

We prove that our algorithm is opitmal in number of visited cells (i.e., it does not visit any unnecessary cell to answer the query). To prove the correctness, we show that our algorithm visits all the cells that must be visited to compute the correct results.

Proof. Let $q_{old}.score_k$ and $q_{new}.score_k$ be the scores of kth object before and after the update, respectively. Consider the case when $q_{old}.score_k \geq q_{new}.score_k$ (i.e., the number of incoming updates is at least equal to the number of outgoing updates). This implies $|q.kA| \geq k$ (line 8 of Algortihm 2) and we do not need to visit any new cell to update the result. Therefore, we only need to consider the case when $q_{old}.score_k < q_{new}.score_k$ (line 9 of Algorithm 2). Below, we prove the optimality and correctness of our algorithm for this case.

Let C be the set of minimum cells that have to be visited in order to guarantee the correct results. First, we identify C and show that our algorithm does not visit any unnecessary cell c' such that $c' \notin C$. A cell c' for which $S^U(c') \leq q_{old}.score_k$ is not required to be visited. This is because all the objects in this cell have been considered earlier. Similarly, a cell c' for which $S^L(c') \leq q_{new}.score_k$ is not required to be visited. This is because every object in such cell has score at least equal to $q_{new}.score_k$. Therefore, the set C of minimum cells consists of every cell c that satisfies the following two inequalities.

$$S^U(c) > q_{old}.score_k \qquad (1)$$
$$S^L(c) < q_{new}.score_k \qquad (2)$$

Please note that in our update handling algorithm, we ignore the cells that have $S^U(c) \leq q_{old}.score_k$ and terminate the algorithm when $S^L(c) \geq q_{new}.score_k$ (see Section 4.2 Phase 2). Thus, we satisfy both of the above inequalities. Therefore, our algorithm does not visit any un-necessary cell and is optimal in the number of visited cells.

Please note that the initial computation can be considered as a special case of update handling where $q_{old}.score_k$ is set to zero.

As a proof of correctness, we show that our algorithm visits all the cells in the set C. Recall that we maintain the cells in a heap based on their lower bound scores. Therefore, the cells are visited in the ascending order of their lower bound scores and it guarantees that every cell c for which $S^L(c) < q_{new}.score_k$ is visited.

5 Experiments

We choose three inherently different spatial queries and run experiments to evaluate the efficiency of our unified algorithm. More specifically, we run the experiments for the constrained kNN queries, the aggregate kNN queries and the furthest k neighbors queries. Since our algorithm is based on the **C**onceptual **G**rid **T**ree, we refer to it as CGT.

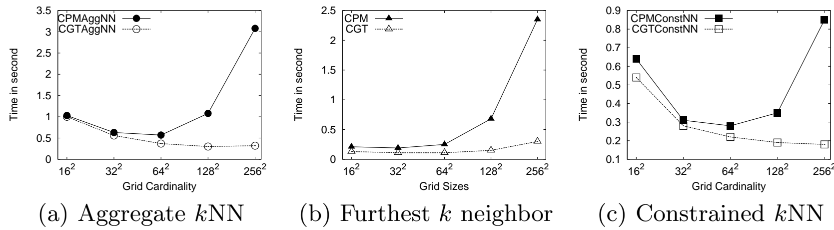
We compare our algorithm with CPM [14]. As mentioned by the authors, it can be modified to answer constrained kNN and aggregate kNN queries. We

extend CPM to answer furthest k neighbors query as follow. We compute the furthest conceptual rectangles from the query cell c_q in all four directions (i.e., right, down, left, up). Initially, we insert the furthest rectangles in the heap with their keys set to the maximum distances between the query and the rectangles. After de-heaping a rectangle, the previous level (closer to the query cell) rectangle in the same direction is inserted in the heap. We use a max heap and thus retrieve the rectangles in descending order of their maximum distances from the query.

Parameter	**Range**
Number of objects (×1000)	20, 40, 60, 80, **100**
Number of queries	100, 500, **1000**, 2500, 5000
Value of k	2, 4, 8, **16**, 32, 64, 128
Object/query agility (in %)	10, 30, **50**, 70, 90
Aggregate function (for aggregate queries only)	**sum**, max, min
Number of query instances (for aggregate queries only)	5, **10**, 25, 50, 100

Table 1. System Parameters

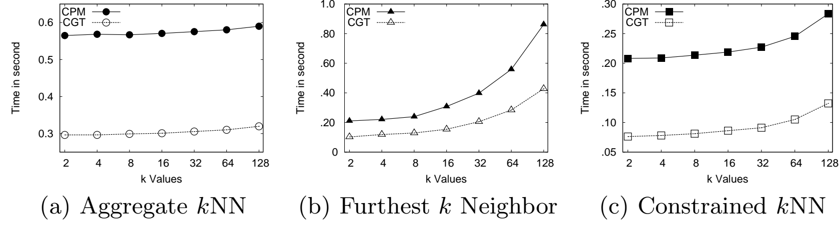
Our experiment settings are similar to those used in [14]. More specifically, we use Brinkhoff data generator [2] to generate objects moving on the road network of Oldenburg, a German city. The queries are generated similarly. Each query is monitored for 100 time stamps and the experiment figures show the total computation time for a single query for the duration of 100 time stamps. Table 1 shows the parameters used in our experiments and the default values are shown in bold. Agility corresponds to the percentage of objects and queries that issue location updates at a given timestamp.



(a) Aggregate kNN (b) Furthest k neighbor (c) Constrained kNN

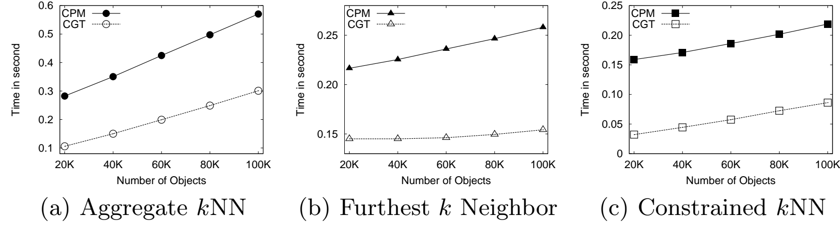
Fig. 5. Effect of grid cardinality

First we study the effect of grid cardinality. We vary the grid size and compare the algorithms for each of the three queries in Fig. 5. In accordance with previous work that use grid based approach, the performance degrades if the grid size is too small or too large. More specifically, if the grid has too low cardinality, the cost of the queries increases because each cell contains larger number of objects.



(a) Aggregate kNN (b) Furthest k Neighbor (c) Constrained kNN

Fig. 6. Effect of the value of k



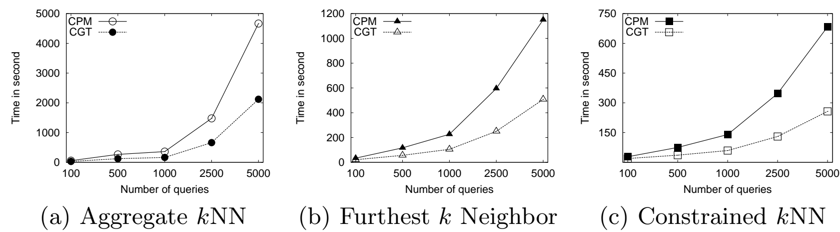
(a) Aggregate kNN (b) Furthest k Neighbor (c) Constrained kNN

Fig. 7. Effect of number of objects

On the other hand, if the grid cardinality is too high then most of the cells are empty and the cost increases because the number of visited cells becomes large.

Based on Fig. 5, we choose the default grid sizes for both of the algorithms. More specifically, the default grid size selected for CPM is 64×64 and for CGT is 128×128. In the remaining experiments, we choose these default grid sizes for both of the algorithms.

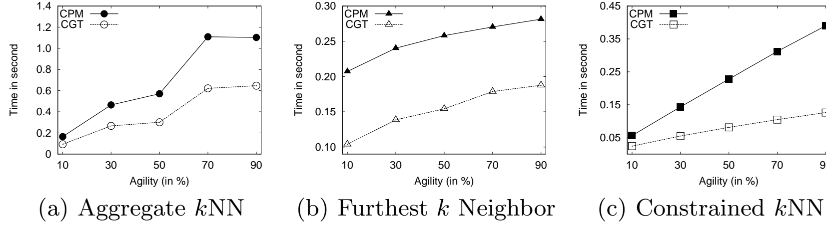
In Fig. 6, Fig. 7, Fig. 8 and Fig. 9, we study the effect of k, the number of data objects, the number of queries and the agility of the datasets, respectively. Although our algorithm is unified and does not require modification for different queries, it outperforms CPM for all different settings.



(a) Aggregate kNN (b) Furthest k Neighbor (c) Constrained kNN

Fig. 8. Effect of number of queries

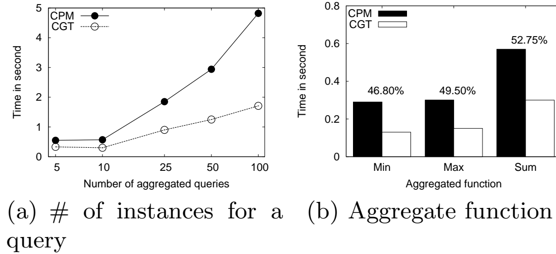
Since aggregate kNN queries has two extra parameters (the number of query instances and the aggregate function), we conduct more experiments to evaluate



(a) Aggregate kNN (b) Furthest k Neighbor (c) Constrained kNN

Fig. 9. Effect of data agility

the performance of our algorithm for these parameters. Fig. 10(a) shows the effect of number of query instances. As expected, the cost of each algorithm increases when the number of query instances is large. This is because the cost of aggregate function increases with the increase in the number of query instances.



(a) # of instances for a query (b) Aggregate function

Fig. 10. Aggregate kNN effect

Fig. 10(b) studies the effect of different aggregate functions (i.e., Sum, Max and Min). Our algorithm outperforms CPM for each of the aggregate functions. The percentage on top of each group represents the percentage of the time taken by our unified algorithm with respect to CPM.

6 Conclusion

We are first to present a unified algorithm to answer a broad class of spatial queries. Our proposed algorithm is optimal in the sense that it visits minimum number of cells throughout the life of a continuous query. Our extensive experimental results demonstrate that for each inherently different spatial queries our unified algorithm significantly outperforms existing best known algorithm.

Acknowledgments. Xuemin Lin was supported by the ARC Discovery Grants (DP110102937, DP0987557 and DP0881035).

References

1. Bae, S.W., Korman, M., Tokuyama, T.: All farthest neighbors in the presence of highways and obstacles. In: WALCOM. pp. 71–82 (2009)
2. Brinkhoff, T.: A framework for generating network-based moving objects. GeoInformatica 6(2), 153–180 (2002)
3. Cheema, M.A., Brankovic, L., Lin, X., Zhang, W., Wang, W.: Multi-guarded safe zone: An effective technique to monitor moving circular range queries. In: ICDE. pp. 189–200 (2010)
4. Cheema, M.A., Lin, X., Zhang, Y., Wang, W., Zhang, W.: Lazy updates: An efficient technique to continuously monitoring reverse knn. VLDB 2(1), 1138–1149 (2009)
5. Cheema, M.A., Yuan, Y., Lin, X.: Circulartrip: An effective algorithm for continuous nn queries. In: DASFAA. pp. 863–869 (2007)
6. Chen, Z., Ness, J.W.V.: Characterizations of nearest and farthest neighbor algorithms by clustering admissibility conditions. Pattern Recognition 31(10), 1573–1578 (1998)
7. Ferhatosmanoglu, H., Stanoi, I., Agrawal, D., Abbadi, A.E.: Constrained nearest neighbor queries. In: SSTD. pp. 257–278 (2001)
8. Gedik, B., Liu, L.: Mobieyes: Distributed processing of continuously moving queries on moving objects in a mobile system. In: EDBT. pp. 67–87 (2004)
9. Hasan, M., Cheema, M.A., Qu, W., Lin, X.: Efficient algorithms to monitor continuous constrained k nearest neighbor queries. In: DASFAA (1). pp. 233–249 (2010)
10. Henrich, A.: A distance scan algorithm for spatial access structures. In: ACM-GIS. pp. 136–143 (1994)
11. Hjaltason, G.R., Samet, H.: Ranking in spatial databases. In: SSD. pp. 83–95 (1995)
12. Lazaridis, I., Porkaew, K., Mehrotra, S.: Dynamic queries over mobile objects. In: EDBT. pp. 269–286 (2002)
13. Luo, Y., Chen, H., Furuse, K., Ohbo, N.: Efficient methods in finding aggregate nearest neighbor by projection-based filtering. In: ICCSA (3). pp. 821–833 (2007)
14. Mouratidis, K., Hadjieleftheriou, M., Papadias, D.: Conceptual partitioning: An efficient method for continuous nearest neighbor monitoring. In: SIGMOD. pp. 634–645 (2005)
15. Papadias, D., Tao, Y., Mouratidis, K., Hui, C.K.: Aggregate nearest neighbor queries in spatial databases. ACM Trans. Database Syst. 30(2), 529–576 (2005)
16. Roussopoulos, N., Kelley, S., Vincent, F.: Nearest neighbor queries. In: SIGMOD. pp. 71–79 (1995)
17. Suri, S.: Computing geodesic furthest neighbors in simple polygons. J. Comput. Syst. Sci. 39(2), 220–235 (1989)
18. Wu, K.L., Chen, S.K., Yu, P.S.: Incremental processing of continual range queries over moving objects. IEEE Trans. Knowl. Data Eng. 18(11), 1560–1575 (2006)
19. Wu, W., Tan, K.L.: isee: Efficient continuous k-nearest-neighbor monitoring over moving objects. In: SSDBM. p. 36 (2007)
20. Xiong, X., Mokbel, M.F., Aref, W.G.: Sea-cnn: Scalable processing of continuous k-nearest neighbor queries in spatio-temporal databases. In: ICDE. pp. 643–654 (2005)
21. Yiu, M.L., Mamoulis, N., Papadias, D.: Aggregate nearest neighbor queries in road networks. IEEE Trans. Knowl. Data Eng. 17(6), 820–833 (2005)
22. Yu, X., Pu, K.Q., Koudas, N.: Monitoring k-nearest neighbor queries over moving objects. In: ICDE. pp. 631–642 (2005)