Conceptual Partitioning: An Efficient Method for Continuous Nearest Neighbor Monitoring

Kyriakos Mouratidis[†] Marios Hadjieleftheriou[§] Dimitris Papadias[†]
[†] Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
{kyriakos, dimitris}@cs.ust.hk
[§] Department of Computer Science
Boston University
Boston, MA, USA
marioh@cs.bu.edu

ABSTRACT

Given a set of objects P and a query point q, a k nearest neighbor (k-NN) query retrieves the k objects in P that lie closest to q. Even though the problem is well-studied for static datasets, the traditional methods do not extend to highly dynamic environments where multiple continuous queries require real-time results, and both objects and queries receive frequent location updates. In this paper we propose *conceptual partitioning* (CPM), a comprehensive technique for the efficient *monitoring* of continuous NN queries. CPM achieves low running time by handling location updates only from objects that fall in the vicinity of some query (and ignoring the rest). It can be used with multiple, static or moving queries, and it does not make any assumptions about the object moving patterns. We analyze the performance of CPM and show that it outperforms the current state-of-the-art algorithms for all problem settings. Finally, we extend our framework to aggregate NN (ANN) queries, which monitor the data objects that minimize the aggregate distance with respect to a set of query points (e.g., the objects with the minimum sum of distances to all query points).

1. INTRODUCTION

Early work in spatial databases focused on the *point* k-NN query that retrieves the k ($\geq$1) objects from a static dataset that are closest (according to Euclidean distance) to a static query point. The existing algorithms (e.g., [H84, RKV95, HS99]) consider that the data are indexed with a spatial access method and utilize some pruning bounds to restrict the search space. In addition, several papers study variations of NN search such as *reverse* NNs [SRAA01] and *constrained* NNs [FSAA01]. Recently, the focus has shifted towards moving NN queries and/or objects in client-server architectures. Song and Roussopoulos [SR01] reduce the number of moving NN queries over static objects by introducing some redundancy. In particular, when a k-NN query is processed, the server sends to the client a number $m > k$ of neighbors. The k nearest neighbors at a new location q' will be among the m objects of the first query q provided that the distance between q and q' is within a range determined by k and m. For the same settings (moving query - static data objects), Zhang et al. [ZZP+03] propose the concept of *location-based* queries that return the NN of q along with its *Voronoi cell*, i.e., the area around the query point where the NN set remains the same. The Voronoi cell is computed on-the-fly using an R-tree on the data objects. Given clients and data objects that move with linear and known velocities, *time-parameterized* [TP03] queries report, in addition to the current NN set, its validity period and the next change of the result (that will occur at the end of the validity period). *Linear NN* [BJKS02, TP03] queries return all NN sets up to a future timestamp q_t assuming that there are no updates of the velocity vectors between the current time and q_t.

All the above techniques target the efficient processing of a single *snapshot* query since they report the NN set at the query time, possibly with some validity information (e.g., expiry time, Voronoi cell), or generate future results based on predictive features (e.g., velocity vectors of queries or data objects). On the other hand, continuous monitoring: (i) involves multiple long-running queries (from geographically distributed clients), (ii) is concerned with both computing *and* keeping the results up to date, (iii) usually assumes main-memory processing to cope with the intensive (object or query) location updates, (iv) attempts to minimize factors such as the CPU or communication cost (as opposed to I/O overhead). Continuous monitoring of spatial queries is becoming increasingly important due to the wide availability of inexpensive and compact positioning devices, the evolution of mobile communications and the need for improved location-based services. Consequently, several techniques (reviewed in Section 2) have been developed in the last few years for continuous range and NN queries.

In this paper, we propose the *conceptual partitioning monitoring* (CPM) method for NN queries in highly dynamic environments. The data objects are indexed by a main-memory grid G consisting of cells with size $\delta \times \delta$ (assuming two-dimensional space). Each cell c in the grid is associated with the list of objects residing therein. The running queries are stored along with their current result in a query table QT. When a query q arrives at the system, its initial result is computed by the NN search module of CPM. CPM organizes the cells into (hyper) rectangles based on their proximity to q. This conceptual partitioning provides a natural processing order of the cells in G, so that the NN search considers the minimal set of cells in order to retrieve the NNs of q. We refer to the set of encountered cells as the *influence region* of q. The next task of CPM is to monitor the results of the queries upon the arrival of object updates. Clearly, only updates affecting the influence region of a query can potentially invalidate its current result. To restrict processing to such updates and to efficiently compute the changes in the results, we maintain book-keeping information in the object index and the query table. We also show that it is often possible to compute the new result of an affected query among the objects that issue updates, without searching in

SIGMOD 2005, June 14–16, 2005, Baltimore, Maryland, USA.

G at all. Finally, we tackle the case that the query points also move. As we show qualitatively and verify experimentally, CPM outperforms the existing state-of-the-art methods, usually by more than an order of magnitude.

Furthermore, CPM provides a general methodology that can be applied to several types of spatial queries. As a case study we use *aggregate nearest neighbor* (ANN) queries. Given a set of query points $Q = \{q_1,q_2,\ldots,q_m\}$ and an object p, the aggregate distance $adist(p,Q)$ is defined as a monotonically increasing function f over the individual distances $dist(p,q_i)$ between p and each point $q_i \in Q$. Assuming, for example, n users at locations q_1, ... q_n and f=*sum*, an ANN query outputs the data object p that minimizes $adist(p,Q) = \sum_{q_i \in Q} dist(p,q_i)$, i.e., the *sum* of distances that the users have to travel in order to meet at the position of p. Similarly, if f=*max*, the ANN query reports the object p that minimizes the maximum distance that any user has to travel to reach p. In turn, this leads to the earliest time that all users will arrive at the location of p (assuming that they move with the same speed). The *sum* ANN query has been studied in [PSTM04] for static queries and data indexed by R-trees. The adaptation of CPM to the continuous monitoring of ANN queries can handle arbitrary aggregate functions and preserves the excellent performance of the algorithm in the presence of frequent updates.

The rest of the paper is organized as follows. Section 2 surveys related work on continuous monitoring of spatial queries, focusing mostly on NN search. Section 3 presents the conceptual partitioning monitoring method. Section 4 provides an analysis of the space and time requirements of CPM, as well as a qualitative comparison with existing systems. Section 5 discusses ANN monitoring, while Section 6 experimentally evaluates CPM. Finally, Section 7 concludes the paper with directions for future work.

2. RELATED WORK

The first monitoring method for spatial queries, called *Q-index* [PXK+02], assumes static range queries over moving objects. The queries are indexed by an R-tree and moving objects probe the index to find the queries that they influence. *Q-index* avoids the expensive (due to intensive updates) maintenance of an index on the objects. In addition, it utilizes the concept of *safe regions* to reduce the number of updates. In particular, each object p is assigned a circular or rectangular region, such that p needs to issue an update only if it exits this area (otherwise, it does not influence the result of any query). MQM [CHC04], another range monitoring method, partitions the workspace into rectangular sub-domains. Each object in the system is assigned a *resident domain*, consisting of adjacent sub-domains. An object is aware only of the range queries intersecting its resident region, and reports its location to the server when it crosses the boundary of any of these queries. The number of sub-domains that form an object's resident region depends on how many queries it can store and process concurrently. When an object exits its resident region, it requests a new one from the server. To decide the new resident region, the server uses a binary partitioning tree, which maintains for each sub-division of the workspace the queries that intersect it. This method applies only to static ranges.

To deal with moving range queries, Gedik and Liu [GL04] propose another distributed system, called *Mobieyes*. *Mobieyes* partitions the workspace using a grid and maintains the *monitoring regions* of the queries. The monitoring region of a query is defined as the union of the grid cells it can potentially intersect, provided that its center remains within its current cell. Objects falling in the monitoring region of a query receive information about the query position and velocity, and notify the server when they enter or leave the predicted query region. Note that this way the objects store locally and monitor their spatial relationship only with queries that they might actually affect when they move, saving their limited storage and processing resources. On the other hand, queries issue updates to the server when they change velocity vector, or when they move out of their current cell.

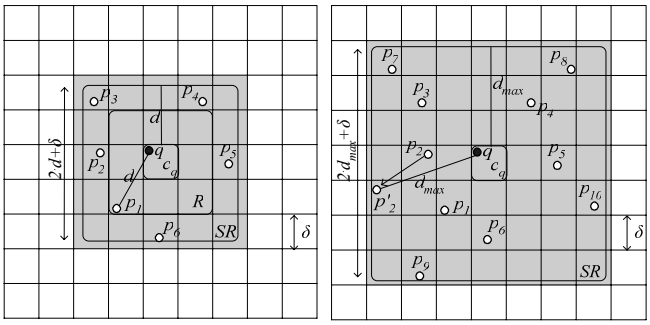
Mokbel et al. [MXA04] present SINA, a system that centrally processes continuous range queries over mobile data. SINA is based on *shared execution* and *incremental evaluation*. Shared execution is achieved by implementing query evaluation as a spatial join between the objects and the queries. Incremental evaluation implies that the query processor computes only the updates of the previously reported answers, as opposed to re-evaluating the queries from scratch. The result updates are either positive or negative. The former category corresponds to objects entering the range of a query, while the latter one to objects leaving a range. Both the object and the query indexes are implemented as disk-resident regular grids. Let U_P and U_q be the set of objects and queries that issue location updates since the previous evaluation cycle. Processing begins with the *hashing phase* that joins U_P and U_q in-memory to produce positive updates. Next, the *invalidation phase* generates negative updates for objects in U_P that move out of their current cell and queries in U_q that exit cells that they used to overlap with. Finally, movement within the same cell is handled in the *joining phase*; for each cell that contains objects in U_P or intersects queries in U_q, SINA joins the new objects with the existing queries, and the new queries with the static objects. The resulting updates are merged with the updates of the previous phases (potentially canceling out some of them), and are reported to the client.

All the aforementioned methods focus on range query monitoring, and their extension to NN queries is either impossible or non-trivial. Henceforth, we discuss algorithms that target explicitly NN processing. Koudas et al. [KOTZ04] describe DISC, a technique for e-approximate k-NN queries over streams of multidimensional points. The returned k^{th} NN lies at most e distance units farther from q than the actual k^{th} NN of q. DISC partitions the space with a regular grid of granularity such that the maximum distance between any pair of points in a cell is at most e. To avoid keeping all arriving data in the system, for each cell c it maintains only K points falling therein and discards the rest. It is proven that an exact k-NN search in the retained points corresponds to a valid ek-NN answer over the original dataset provided that $k \leq K$. DISC indexes the data points with a B-tree that uses a space-filling curve mechanism to facilitate fast updates and query processing. The authors show how to adjust the index to: (i) use the minimum amount of memory in order to guarantee a given error bound e, or (ii) achieve the best possible accuracy, given a fixed amount of memory. DISC can process both snapshot and continuous ek-NN queries.

Yu et al. [YPK05] propose a method, hereafter referred to as YPK-CNN[1], for continuous monitoring of exact k-NN queries.

[1] Yu et al. [YPK05] actually propose three methods. YPK-CNN is shown to be the best in their experimental evaluation.

Objects are assumed to fit in main memory and are indexed with a regular grid of cells with size $\delta \times \delta$. YPK-CNN does not process updates as they arrive, but directly applies the changes to the grid. Each NN query installed in the system is re-evaluated every T time units. When a query q is evaluated for the first time, a two-step NN search technique retrieves its result. The initial step visits the cells in a square R around the cell c_q covering q until k objects are found. Figure 2.1a, shows an example of a single NN query where the first candidate NN is p_1 with distance d from q; p_1 is not necessarily the actual NN since there may be objects (e.g., p_2) in cells outside R with distance smaller than d. To retrieve such objects, the second step searches in the cells intersecting the square SR centered at c_q with side length $2 \cdot d + \delta$, and determines the actual k NN set of q therein. In Figure 2.1a, YPK-CNN processes p_1 up to p_6 and returns p_2 as the actual NN. The accessed cells appear shaded.

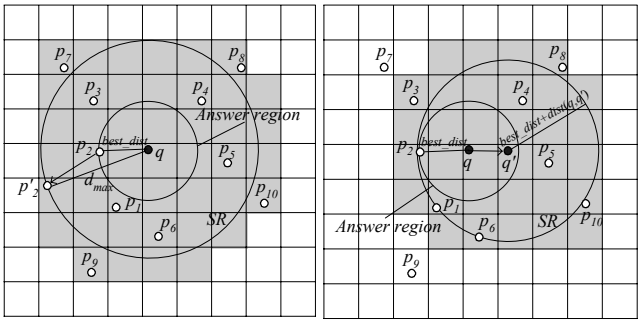


(a) NN search (b) Update handling

Figure 2.1: YPK-CNN examples

When re-evaluating an existing query q, YPK-CNN makes use of its previous result in order to restrict the search space. In particular, it computes the maximum distance d_{max} of the current locations of the previous NNs (i.e., d_{max} is the distance of the previous neighbor that moved furthest). The new SR is a square centered at c_q with side length $2 \cdot d_{max} + \delta$. In Figure 2.1b, assume that the current NN p_2 of q moves to location p'_2. Then, the rectangle defined by $d_{max} = dist(p'_2,q)$ is guaranteed to contain at least one object (i.e., p_2). YPK-CNN collects all objects (p_1 up to p_{10}) in the cells intersecting SR and identifies the new NN p_1. Finally, when a query q changes location, it is handled as a new one (i.e., its NN set is computed from scratch). Yu et al. also discuss the application of YPK-CNN with a hierarchical grid that improves performance for highly skewed data.

SEA-CNN [XMA05] focuses exclusively on monitoring the NN changes, without including a module for the first-time evaluation of an arriving query q (i.e., it assumes that the initial result is available). Objects are stored in secondary memory, indexed with a regular grid. The *answer region* of a query q is defined as the circle with center q and radius *best_dist*, where *best_dist* is the distance of the current k^{th} NN. Book-keeping information is stored in the cells that intersect the answer region of q to indicate this fact. When updates arrive at the system, depending on which cells they affect and whether these cells intersect the answer region of the query, SEA-CNN determines a circular search region SR around q, and computes the new k NN set of q therein. To determine the radius r of SR, the algorithm distinguishes the following cases: (i) If some of the current NNs move within the answer region or some outer objects enter the answer region, SEA-CNN sets r=*best_dist* and processes all objects falling in the answer region in order to retrieve the new NN set. (ii) If any of the current NNs moves out of the answer region, processing is similar to YPK-CNN; i.e., $r = d_{max}$ (where d_{max} is the distance of the previous NN that moved furthest from q), and the NN set is computed among the objects lying in SR. Assume that in Figure 2.2a the current NN p_2 issues an update reporting its new location p'_2. SEA-CNN sets $r=d_{max}=dist(p'_2,q)$, determines the cells intersecting SR (these cells appear shaded), collects the corresponding objects (p_1 up to p_{10}), and retrieves the new NN p_1. (iii) Finally, if the query q moves to a new location q', then SEA-CNN sets $r = best\_dist + dist(q,q')$, and computes the new k NN set of q by processing all the objects that lie in the circle centered at q' with radius r. For instance, in Figure 2.2b the algorithm considers the objects falling in the shaded cells (i.e., objects from p_1 up to p_{10} except for p_7 and p_9) in order to retrieve the new NN (p_5).



(a) p_2 issues an update (b) q moves to q'

Figure 2.2: SEA-CNN update handling examples

Table 2.1 summarizes the properties of existing methods for monitoring spatial queries. The processing type refers to whether mobile objects have some computing capabilities, or the entire processing cycle takes place in a central server. For instance, *Q-index* is classified as a distributed method since the objects decide whether they exit their safe regions before they issue an update. On the other hand, SINA follows a centralized paradigm since each object issues an update whenever it moves, independently of whether it influences any query or not. In summary, the only existing techniques applicable to continuous monitoring of exact k-NN queries are YPK-CNN and SEA-CNN. Similar to these methods CPM also assumes centralized processing (in main memory[2]). We compare CPM against YPK-CNN and SEA-CNN both qualitatively (in Section 4) and experimentally (in Section 6). In the next section, we present CPM in detail.

Method	Query	Memory	Processing	Result
Q-index	Range	Main	Distributed	Exact
MQM	Range	Main	Distributed	Exact
Mobieyes	Range	Main	Distributed	Exact
SINA	Range	Disk	Centralized	Exact
DISC	NN	Main	Centralized	Approximate
YPK-CNN	NN	Main	Centralized	Exact
SEA-CNN	NN	Disk	Centralized	Exact

Table 2.1: Properties of monitoring methods

[2] Even though SEA-CNN assumes that objects reside in secondary memory, it can be also used for memory-resident data.

3. CONCEPTUAL PARTITIONING MONITORING

In accordance with real-world scenarios, we assume 2D[3] data objects and queries that change their location frequently and in an unpredictable manner. An update from object p is a tuple $<p.id, x_{old}, y_{old}, x_{new}, y_{new}>$, implying that p moves from (x_{old}, y_{old}) to (x_{new}, y_{new}). A central server receives the update stream and continuously monitors the k NNs of each query q installed in the system. Similar to existing approaches (e.g., YPK-CNN, SEA-CNN), we use a grid index since a more complicated data-structure (e.g., main memory R-tree) would be very expensive to maintain dynamically. The extent of each cell on every dimension is δ, so that the cell $c_{i,j}$ at column i and row j (starting from the low-left corner of the data space) contains all objects with x co-ordinate in the range $[i\cdot\delta, (i+1)\cdot\delta)$ and y co-ordinate in the range $[j\cdot\delta, (j+1)\cdot\delta)$. Conversely, an object with co-ordinates (x,y) belongs to the cell $c_{i,j}$, where $i=\lfloor x/\delta \rfloor$ and $j=\lfloor y/\delta \rfloor$. CPM (and SEA-CNN) can also be applied with the hierarchical grid of [YPK05].

Section 3.1 describes the NN computation algorithm, which constitutes the core module of CPM. Then, Sections 3.2 and 3.3 discuss the handling of location updates. Table 3.1 summarizes the primary symbols and functions we use throughout this section.

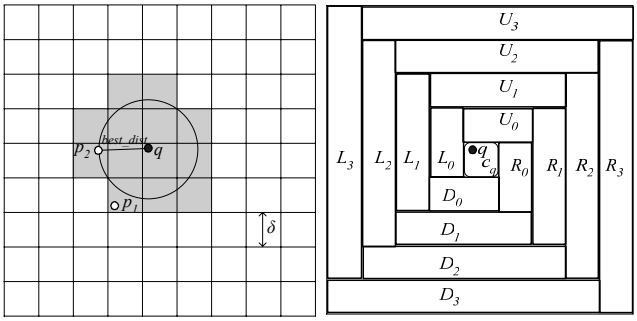
Symbol	Description
P	The set of moving objects
N	Number of objects in P
G	The grid that indexes P
δ	Cell side length
q	The query point
c_q	The cell containing q
n	The number of queries installed in the system
$dist(p,q)$	Euclidean distance from object p to query point q
$best\_NN$	The best NN list of q
$best\_dist$	The distance of the k^{th} NN from q
$mindist(c,q)$	Minimum distance between cell c and query point q

Table 3.1: Frequently used symbols and functions

3.1 The NN computation module of CPM

Given a cell c and a query q, $mindist(c,q)$ is the minimum possible distance between any object $p\in c$ and q. Let $best\_NN$ be the list of the k best NNs (of q) found so far, and $best\_dist$ be the distance of the k^{th} of them. If $mindist(c,q)\geq best\_dist$, we can safely prune c because it cannot contain any object lying closer to q than any of the current NNs. Based on this observation, a naive way to process a NN query q in P, is to sort all cells $c\in G$ according to $mindist(c,q)$, and visit them in ascending $mindist(c,q)$ order. For each considered cell, we compute $dist(p,q)$ for the objects p inside, and update accordingly the $best\_NN$ list. The search terminates when the cell c under consideration has $mindist(c,q) \geq best\_dist$. Figure 3.1a illustrates this process for a 1-NN query q. The algorithm visits only the shaded cells and encounters in total two objects, p_1 and p_2. Between them, p_2 is returned as the result of the query.

It can be easily shown that the above algorithm processes only the cells that intersect the circle centered at q with radius equal to the distance between q and its k^{th} NN. These cells have to be visited anyway in order to avoid false misses; therefore, the naïve algorithm is optimal in terms of the number of processed cells. Nevertheless, in practice it may be very expensive, since it requires computing the $mindist$ for all cells and subsequently sorting them. CPM overcomes this problem and avoids unnecessary computations by utilizing a conceptual space partitioning.



(a) Retrieval of one NN (b) Partitioning into rectangles

Figure 3.1: NN search and conceptual partitioning

Figure 3.1b illustrates the conceptual partitioning of the space around the cell c_q of q. Each rectangle $rect$ is defined by a $direction$ and a $level\ number$. The direction could be U, D, L, or R (for up, down, left and right) depending on the relative position of $rect$ with respect to q. The level number indicates the number of rectangles between $rect$ and c_q. Lemma 3.1 regulates the visiting order among rectangles of the same direction.

Lemma 3.1: For rectangles DIR_j and DIR_{j+1} of the same direction DIR with level numbers j and $j+1$, respectively, it holds that $mindist(DIR_{j+1},q) = mindist(DIR_j,q) + \delta$.

Proof: Without loss of generality, assume that the direction is D. The minimum distance of q from either rectangle equals the length of its projection on the top edge of the rectangle. Since the side length of the cells is δ, it follows that $mindist(DIR_{j+1},q) = mindist(DIR_j,q) + \delta$. □

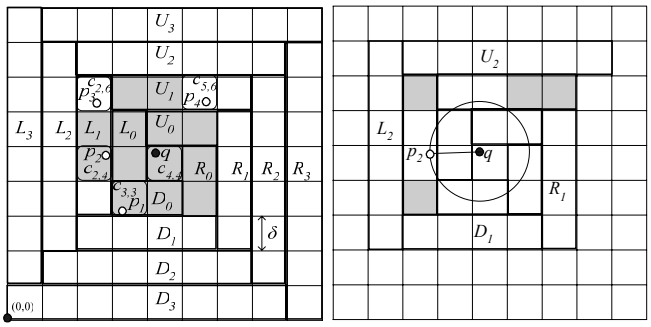
Based on Lemma 3.1, the NN computation module of CPM visits cells in ascending $mindist(c,q)$ order, thus, preserving the property of processing the minimal set of cells. In particular, CPM initializes an empty heap H and inserts (i) the cell c_q with key $mindist(c_q,q)=0$, and (ii) the level zero rectangles for each direction DIR, with key $mindist(DIR_0,q)$. Then, it starts de-heaping entries iteratively. If the de-heaped entry is a cell, it examines the objects inside and updates accordingly the $best\_NN$. If the de-heaped entry is a rectangle DIR_{lvl}, it inserts into H (i) each cell $c\in DIR_{lvl}$ with key $mindist(c,q)$ and (ii) the next level rectangle DIR_{lvl+1} with key $mindist(DIR_{lvl+1},q) = mindist(DIR_{lvl},q) + \delta$. The algorithm terminates when the next entry in H (corresponding either to a cell or a rectangle) has key greater than or equal to $best\_dist$.

Proof of correctness: Let $best\_NN$ be the list of NNs returned by the algorithm, and $best\_dist$ be the distance of the k^{th} NN. Clearly, all cells c inserted at some point into H do not contain any better NN than the objects in $best\_NN$. This is guaranteed by the sorting property of the heap and the fact that $dist(p,q) \geq mindist(c,q)$ holds $\forall p\in c$. In order to prove correctness, it suffices to show that each cell that was not inserted into H cannot contain any object

[3] We focus on two-dimensional Euclidean spaces, but the proposed techniques can be applied to higher dimensionality and other distance metrics. Furthermore, for ease of presentation, the examples demonstrate retrieval of a single NN.

closer to q than *best_dist*. This part of the proof is based on the observation that, at any point, the heap H contains exactly four rectangle entries, one for each direction. We call these rectangles *boundary boxes*. Let the boundary box of direction *DIR* be DIR_{lvl}. The algorithm has considered all cells falling into rectangles DIR_i with $i<lvl$. From Lemma 3.1 it follows that all cells c belonging to DIR_i with $i>lvl$ have $mindist(c,q)>mindist(DIR_{lvl},q)$. Since $mindist(DIR_{lvl},q) \geq best\_dist$ for each boundary box DIR_{lvl}, and since all the unexplored space falls in some rectangle of some direction *DIR* with level greater than *lvl*, *best_NN* is the correct result of q. □

In the example of Figure 3.2a, CPM initially inserts into the heap the cell $c_q = c_{4,4}$ and the rectangles of level zero, i.e., $H = \{<c_{4,4},0>, <U_0,0.1>, <L_0,0.2>, <R_0,0.8>, <D_0,0.9>\}$ (the numbers indicate *mindist* assuming that δ=1). Then it de-heaps $c_{4,4}$, which is empty[4] and ignored. The next entry in H is U_0. CPM en-heaps the cells of U_0, as well as rectangle U_1 and proceeds in the same way until it de-heaps $<c_{3,3},1>$, where it finds the first candidate NN p_1 with $best\_dist=dist(p_1,q)$=1.7. Since, the next entry in H has key less than *best_dist*, it continues until it de-heaps $c_{2,4}$ and discovers the new candidate p_2, with $best\_dist = dist(p_2,q)$ = 1.3. The algorithm terminates (with p_2 as the NN) when the top heap entry is $c_{5,6}$ because $mindist(c_{5,6},q) \geq best\_dist$.



(a) NN computation (b) Search heap contents

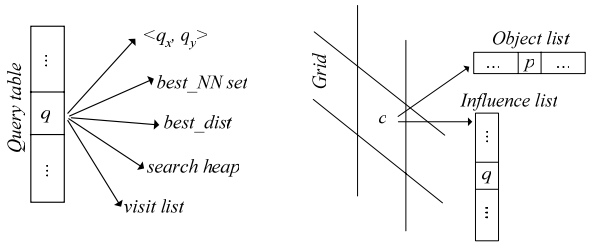
Figure 3.2: A NN computation example

The final point that requires clarification concerns the book-keeping information and related structures maintained for efficient search and handling of updates (to be discussed shortly). CPM keeps (in main memory) a query table *QT* that stores for each query, its co-ordinates, the current result, the *best_dist*, the *visit list*, and the search heap H:

- *best_dist* determines the *influence region* of q, i.e., the set of cells that intersect the circle centered at q with radius *best_dist*. Only updates affecting these cells can influence the NN result.
- The *visit list* of q consists of all cells c processed during NN search, sorted on $mindist(c,q)$. Each cell entry de-heaped from H is inserted at the end of the list. In our example, the *visit list* of q contains the shaded cells in Figure 3.2a.
- The search heap H contains the cell and rectangle entries that were en-heaped, but not de-heaped during NN search (i.e., their *mindist* from q is greater than or equal to *best_dist*). The contents of H in our example are the shaded cells in Figure 3.2b, plus the four boundary boxes U_2, D_1, L_2, and R_1.

[4] Note that, from now on, we ignore the empty cells in our examples for the sake of clarity.

In addition, each cell c of the grid is associated with (i) the list of data objects within its extents, and (ii) the list of queries whose influence region contains c. For example, cell $c_{3,3}$ contains q in its influence list, while $c_{5,6}$ does not. The structures of the query table and the object grid are shown in Figure 3.3.



(a) Query table (b) Object grid

Figure 3.3: Query table and object grid structures

Figure 3.4 presents the full functionality of the CPM NN computation including the maintenance of the data structures. The influence lists of the encountered cells are updated in line 11, while, in line 12, each processed cell is inserted into the *visit list* of q. Line 18 stores the new *best_dist* value in the query table. Upon termination, the heap H is also stored in *QT*. The algorithm is optimal in the sense that it processes the minimal set of cells for retrieving the NN set of q. As opposed to the naïve algorithm discussed in the beginning of the section, the only redundant *mindist* computations concern the cells that were en-heaped but not de-heaped (i.e., the shaded cells in Figure 3.2b). As shown in Section 4.1, the number of such cells and rectangles is small. Furthermore, as discussed next, CPM utilizes these computations for the efficient handling of updates.

```
NN Computation (G, q)
// Input= G: the grid indexing P
// q: the query
1.  best_dist = ∞; best_NN = NULL;
2.  Insert a new entry for q into the query table
3.  Initialize an empty heap H
4.  Insert <c_q, 0> into H
5.  For each direction DIR insert <DIR_0, mindist(DIR_0,q)> into H
6.  Initialize an empty list visit_list
7.  Repeat
8.    Get the next entry of H
9.    If it is a cell entry <c, mindist(c,q)>
10.     For each object p∈c, update best_NN & best_dist if necessary
11.     Insert an entry for q into the influence list of c
12.     Insert <c, mindist(c,q)> at the end of visit_list
13.   Else // it is a rectangle entry <DIR_lvl, mindist(DIR_lvl,q)>
14.     For each cell c in DIR_lvl
15.       Insert <c, mindist(c,q)> into H
16.     Insert <DIR_lvl+1, mindist(DIR_lvl,q)+δ> into H
17. Until the next entry has key ≥ best_dist or H is empty
18. Update the influence region information of q to <q, best_dist>
```

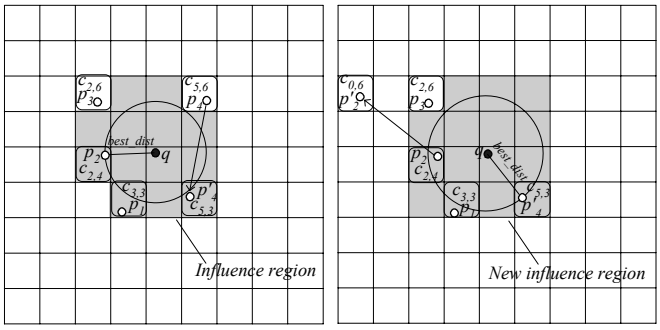
Figure 3.4: The NN computation module of CPM

3.2 Handling a single object update

Assume, for simplicity, that a single update from $p \in P$ arrives at a time. The first step is to delete p from its old cell c_{old}. CPM scans the influence list of c_{old} and identifies the queries that contain p in their *best_NN* set. Specifically, for each query q (in the influence list of c_{old}), if $p \in q.best\_NN$ and $dist(p,q) \leq best\_dist$, then the k

NN set of q remains the same, but the order of the NNs can potentially change. Therefore, CPM updates the order in $q.best\_NN$ to reflect the new $dist(p,q)$. On the other hand, if $p \in q.best\_NN$ and $dist(p,q) > best\_dist$ (i.e., p is a NN that has moved farther from q than $best\_dist$), there may exist objects (not in $q.best\_NN$) that lie closer to q than p; thus, q is marked as affected to indicate this fact and ignored for now. Next, CPM inserts p into its new cell c_{new}, and scans the influence list of c_{new}. For each entry q therein, if q has been marked as affected it ignores it. Otherwise, if $dist(p,q) < q.best\_dist$, it evicts the current k^{th} NN from the result, inserts p into $q.best\_NN$, and updates $q.best\_dist$. The last step re-computes the NN set of every query q that is marked as affected.

Figure 3.5a illustrates update handling, assuming that object p_4 moves to position p'_4. CPM first deletes p_4 from the object list of $c_{5,6}$, which has an empty influence list and, hence, the deletion does not affect any result. Next, it inserts p_4 into its new cell $c_{5,3}$, whose influence list contains an entry for q. Since $dist(p'_4,q) > best\_dist$, update handling terminates without any change in the result. Assume that, later on, object p_2 moves to a new position p'_2, as shown in Figure 3.5b. Since the old cell $c_{2,4}$ contains q in its influence list, CPM checks the query table entry for q and detects that $p_2 = best\_NN$. Query q is marked as affected because $dist(p'_2,q) > best\_dist$. The insertion of p_2 into its new cell $c_{0,6}$ does not trigger any additional processing (because the influence list of $c_{0,6}$ is empty). Finally, CPM invokes the NN *re-computation module* to find the new NN (p'_4) of the affected query q.



(a) p_4 issues an update (b) p_2 issues an update

Figure 3.5: Update examples

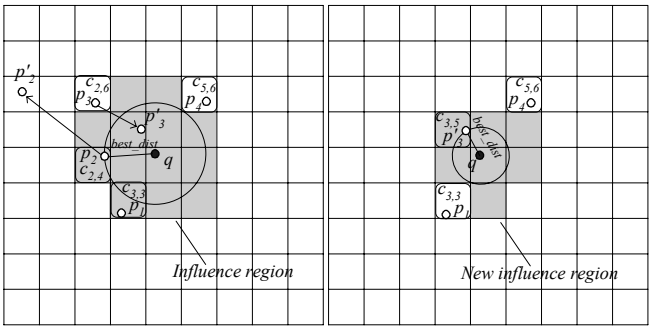
Figure 3.6 illustrates the re-computation module that retrieves the new NN set of the affected queries. The algorithm is based on the same principles as the NN search module of CPM (Figure 3.4), but re-uses the information stored in the query table to reduce the running time. In particular, it starts processing sequentially the cells stored in the *visit list* of q, and then it continues with the entries of the search heap H. Note that all the cells in the *visit list* have *mindist* less than (or equal to) the entries of H. It follows that the NN re-computation algorithm considers cells c in ascending $mindist(c,q)$ order, which guarantees the correctness of the result, as well as the minimality of the set of processed cells. The benefits of NN re-computation over computation from scratch are: (i) it utilizes the previously computed *mindist* values, and (ii) it significantly reduces the number of heap operations (insertions/deletions). Recall that the cost of each heap operation is logarithmic to the heap size, while the "get next" operation on the *visit list* (in line 3 of Figure 3.6) is O(1).

```
NN Re-Computation (G, q)
// Input= G: the grid indexing P, q: the affected query
1.  best_dist = ∞; best_NN = NULL;
2.  Repeat
3.    Get the next element <c, mindist(c,q)> of visit_list
4.    For each object p∈c, update best_NN & best_dist if necessary
5.    Insert an entry for q into the influence list of c
6.  Until the next element has key ≥ best_dist or visit_list is empty
7.  If the first entry in H has key < best_dist
8.    (Same as lines 7-17 of Figure 3.4)
9.  Set influence region information of q to <q, best_dist>
```

Figure 3.6: The NN re-computation module of CPM

3.3 Handling multiple updates

So far we have dealt with processing a single update. However, in the general case, there is a set U_P of object updates that arrive during the time interval between two consecutive update handling cycles. Processing incrementally each update in U_P, as discussed in Section 3.2, guarantees correctness of the result. However, this can be improved upon. Consider the example of Figure 3.7a, where U_P contains location updates for p_2 and p_3. If p_2 is processed first, q will be marked as affected (p_2 is the current NN and moves farther than $best\_dist$), triggering the NN re-computation module. This, however, is unnecessary because object p_3 moves closer to q than the previous $best\_dist$, and we could simply replace the outgoing NN p_2 with the incoming p_3.



(a) p_2 and p_3 issue updates (b) p_3 becomes the NN of q

Figure 3.7: An update handling example

In general, let O be the set of outgoing NNs (i.e., NNs that move farther from q than $best\_dist$) and I be the set of incoming objects (i.e., objects other than the current NNs that move closer to q than $best\_dist$). The circle with center q and radius $best\_dist$ contains objects $I \cup best\_NN - O$. If $|I| \geq |O|$ (where $|I|$ and $|O|$ are the cardinalities of I and O, respectively), this circle includes at least k objects. Therefore, we can form the new NN set from the k best objects in $I \cup best\_NN - O$ without invoking re-computation. We embed this enhancement in the CPM algorithm as follows. Before processing U_P, we record the current $best\_dist$ of q. During update handling, we maintain the *in_list* of the k best incoming objects (we do not need more than the k best incomers in any case). At the end of the procedure, if *in_list* contains more than $|O|$ objects, we merge the NNs in $best\_NN - O$ with *in_list*, and keep the best k among them to form the new result of q. We resort to NN re-computation only if *in_list* contains fewer than $|O|$ objects.

Figure 3.8 shows the complete update handling module of CPM. An important remark is that if $|I| \geq |O|$, the influence region of q shrinks. Consequently, line 22 deletes q from the influence lists of

the cells that no longer belong to it. Note that, at any time, the *visit list* contains a superset of the cells in the influence region of q. Therefore, we can simply scan the cells c in the *visit list* with $mindist(c,q)$ between the new and the old value of *best_dist*, and delete q from their influence lists. The new influence region of q in our example is shown in Figure 3.7b. After update handling, the *visit list* contains a superset of the cells in the influence region (i.e., the *visit list* still includes the shaded cells in Figure 3.7a).

```
Update Handling (G, QT, U_P)
// Input= G: the grid, QT: query table, U_P: set of updates in P
1.  For each query q in QT
2.    Set q.out_count=0; // Counter of outgoing NNs
3.    Initialize a sorted list q.in_list of size k
4.  For each update <p.id,x_old,y_old,x_new,y_new> ∈ U_P
5.    Delete p from its old cell c_old
6.    For each query q in the influence list of c_old
7.      If p∈q.best_NN
8.        If dist(p,q) ≤ q.best_dist // p remains in the NN set
9.          Update the order in q.best_NN
10.       Else // p is an outgoing NN
11.         Evict p from q.best_NN
12.         q.out_count = q.out_count +1;
13.   Insert p into its new cell c_new
14.   For each query q in the influence list of c_new
15.     If dist(p,q)≤q.best_dist and p∉q.best_NN // p is an incomer
16.       Update q.in_list with p
17. For each query q in QT
18.   If q.in_list contains at least q.out_count objects
19.     candidate_list = q.in_list ∪ q.best_NN ;
20.     q.best_NN = the best k objects in candidate_list
21.     Update q.best_dist, Set inf. region of q to <q, p.best_dist>
22.     Delete q from inf. lists of cells no longer in its inf. region
23.   Else // Not enough incoming objects
24.     NN Re-Computation (G, q);
```

Figure 3.8: The update handling module of CPM

In addition to data objects, queries may also be dynamic; i.e., some are terminated, new ones arrive at the system, while others move. When a query is terminated, we delete its entry from *QT* and remove it from the influence lists of the cells in its influence region. For new arrivals, we execute the NN computation algorithm of Figure 3.4. When an existing query q moves, we treat the update as a termination of the old query, and an insertion of a new one, posed at its new location. Queries that receive updates are ignored when handling object updates in order to avoid waste of computations for obsolete queries. Figure 3.9 presents the complete CPM algorithm, covering all update types.

```
NN Monitoring (G, QT)
// Input= G: the grid indexing P, QT: query table
1.  In every processing cycle do
2.    U_q = set of query updates
3.    U_P = set of updates in P
4.    Invoke Update Handling (G, QT, U_P) ignoring queries in U_q
5.    For each query q in U_q
6.      If q is a terminated or a moving query
7.        Delete q from QT and from inf. lists of cells in its inf. region
8.      If q is a new or a moving query
9.        NN Computation (G, q);
10.   Inform client for updated results
```

Figure 3.9: The CPM algorithm

In general, the nearest neighbors of q are concentrated in a small area of the workspace and the influence region of q contains few cells. Therefore, the influence list overhead, and the search heap/*visit list* sizes are expected to be small. However, in case that the physical memory of the system is exhausted, we can directly discard the search heap and the *visit list* of q to free space. Even without this information, CPM can continue monitoring q; the difference is that we have to invoke the NN computation algorithm from scratch (instead of NN re-computation) in line 24 of the update handling module of Figure 3.8.

4. PERFORMANCE ANALYSIS

Section 4.1 analyzes the performance of CPM in terms of space requirements and running time. Section 4.2 compares CPM with the existing algorithms for continuous NN monitoring.

4.1 Analysis of CPM

In order to study the performance of CPM and analyze the effect of the cell size δ, we assume that the objects (queries) are uniformly distributed[5] in a unit square workspace. First, we provide formulae for the space/time overhead with respect to: (i) the number of cells C_{inf} in the influence region of a k-NN query q, (ii) the number O_{inf} of objects in the influence region, and (iii) the total number C_{SH} of cells stored either in the *visit list* or in the search heap of q. Then, we estimate the values of these parameters as functions of δ, and conclude with observations about the expected performance of CPM in practice.

For simplicity, we assume that the minimum unit of memory can store a (real or integer) number. The amount of memory required for an object is s_{obj}=3 for its id and two co-ordinates. Similarly, each heap/*visit list* entry consumes s_{etr}=3 memory units for the cell (rectangle) column/row and *mindist*. The first component of the space overhead is the size of the grid index. The grid contains N objects, consuming $s_{obj} \cdot N = 3 \cdot N$ space, plus the auxiliary influence lists of the cells. For each query q, we insert its id into the influence lists of C_{inf} cells. Assuming n concurrent k-NN queries, the grid index has total size $Space_G = 3 \cdot N + n \cdot C_{inf}$. The query table contains one entry for each query q. The memory dedicated for an entry is $s_{obj} + 2 \cdot k + s_{etr} \cdot (C_{SH}+4)$; s_{obj}=3 is required for the id and co-ordinates of q, while $2 \cdot k$ space is used for the object ids of the k NNs and their distances from q. The $s_{etr} \cdot (C_{SH}+4) = 3 \cdot (C_{SH}+4)$ component corresponds to the storage overhead of the *visit list* and the search heap H; these two structures combined contain C_{SH} cells plus four rectangle entries. It follows that the size of the query table is $Space_{QT} = n \cdot (15+2 \cdot k+3 \cdot C_{SH})$. In total, the memory requirements of CPM are $Space_{CPM} = Space_G + Space_{QT} = 3 \cdot N + n \cdot (15+2 \cdot k+3 \cdot C_{SH}+C_{inf})$ memory units.

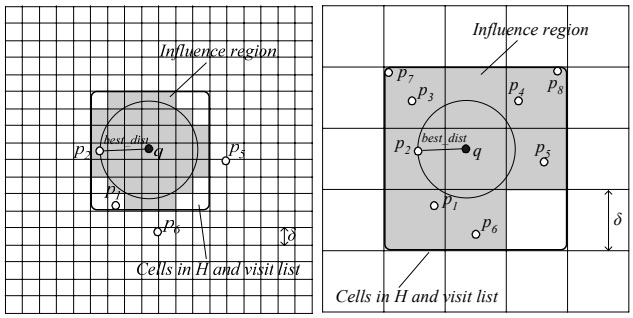
In order to estimate the running time per processing cycle, we assume that $N \cdot f_{obj}$ objects and $n \cdot f_{qry}$ queries issue location updates following random displacement vectors. The total cost is $Time_{CPM} = N \cdot f_{obj} \cdot Time_{ind} + n \cdot f_{qry} \cdot Time_{mq} + n \cdot (1-f_{qry}) \cdot Time_{sq}$, where $Time_{ind}$ is the index update time for a single object, $Time_{mq}$ is the time required for the NN computation of a moving query, and $Time_{sq}$ is the time required for updating the NNs of a static query. The

[5] Although, admittedly, the uniformity assumption does not hold in practice, similar to previous work [YPK05], we use it to obtain general observations about the effect of the problem parameters.

object lists of the cells are implemented as hash tables so that the deletion of an object from its old cell and the insertion into its new one takes expected $Time_{ind}$=2. For each moving query we have to invoke the NN computation algorithm of Figure 3.4 with cost $Time_{mq} = C_{SH}\cdot\log C_{SH} + O_{inf}\cdot\log k + 2\cdot C_{inf}$. The first factor is due to the heap operations. The number of entries in H throughout the NN search procedure is upper-bounded by $C_{SH}+4 \approx C_{SH}$. Since insertion and deletion is logarithmic to the size of the heap, the overall time spent on heap operations is $C_{SH}\cdot\log C_{SH}$. The algorithm processes O_{inf} objects, taking $O_{inf}\cdot\log k$ time cumulatively; each object is probed against the *best_NN* list to update the result, taking $\log k$ time with a red-black tree implementation of *best_NN*. Removing or inserting q from/into the influence list of a cell takes constant expected time (the lists are implemented as hash-tables). Therefore, updating the influence lists of all cells falling in the old and the new influence region costs $2\cdot C_{inf}$. For estimating $Time_{sq}$, observe that at any time instant, the objects are distributed uniformly in the workspace. This implies that the circle with radius *best_dist* always contains k objects, or equivalently, there are as many incoming objects as outgoing NNs. Let there be $|O|$ outgoing NNs. In the worst case, all the remaining k-$|O|$ NNs move. Re-ordering the remaining NNs and inserting the $|I|=|O|$ incomers into *best_NN* takes $Time_{sq} = k\cdot\log k$. Summing over all queries and the index update time, the computational overhead of a processing cycle is $Time_{CPM} = 2\cdot N\cdot f_{obj} + n\cdot f_{qry}\cdot(C_{SH}\cdot\log C_{SH} + O_{inf}\cdot\log k + 2\cdot C_{inf}) + n\cdot(1-f_{qry})\cdot k\cdot\log k$.

It remains to estimate the numbers C_{inf} (O_{inf}) of influencing cells (objects) and cells C_{SH} in the *visited list* and heap of a random query q. Let Θ_q be the circle centered at q with radius equal to *best_dist*. For uniform data, the ratio of the area of Θ_q to the area of the workspace equals k/N so that $best\_dist=\sqrt{k/\pi\cdot N}$. The influence region of q consists of cells intersecting Θ_q. The number of these cells is roughly $C_{inf} = \pi\cdot\lceil best\_dist/\delta\rceil^2$, and the corresponding objects are $O_{inf} = C_{inf}\cdot N\cdot\delta^2$ (each cell contains $N\cdot\delta^2$ objects on average). As δ decreases, C_{inf} increases, the shape of the influence region better approximates Θ_q, and O_{inf} approaches k (which is its minimum value). On the other hand, a large δ leads to a small number of cells which, however, contain a large number of objects. Figure 4.1 illustrates the effect of δ on C_{inf} and O_{inf}, assuming a 1-NN query q. The shaded cells correspond to the influence region of q, which in Figure 4.1a contains C_{inf}=39 cells and O_{inf}=1 objects. For a larger value of δ, in Figure 4.1b, C_{inf}=8 and O_{inf}=8. To estimate C_{SH}, assume for simplicity that q is located at the center of its cell c_q. The boundary boxes are of the same level in each direction. It follows that C_{SH} is the number of cells that intersect the circumscribed square of Θ_q. Thus, C_{SH} can be approximated by $4\cdot\lceil best\_dist/\delta\rceil^2$. Similar to C_{inf}, C_{SH} decreases as δ increases, e.g., in Figure 4.1a, C_{SH}=49, while in Figure 4.1b, C_{SH}=9.

In summary, the space consumed by the influence lists of the cells and the query table, is inversely proportional to δ^2. Similarly, both the size of the influence lists and the size of the query table are linear to n and k. Concerning the computational cost of CPM, index update time is linear to N and f_{obj}. The result maintenance task takes linear time with respect to n, and is expected to grow as f_{qry} increases. The time of NN computation for a new or a moving query depends strongly on the cell size; a small value for δ incurs high overhead due to heap operations, while a large value implies a high number O_{inf} of processed objects.

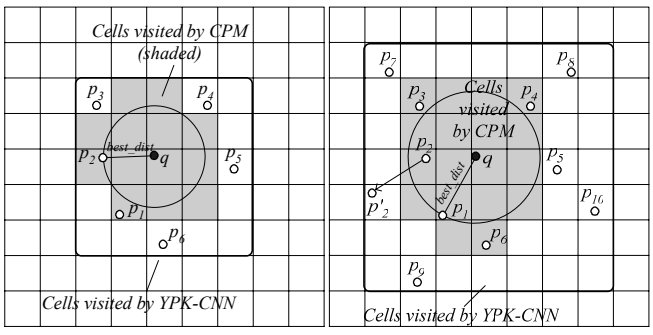


(a) Small δ (b) Large δ

Figure 4.1: The effect of δ on the performance of CPM

4.2 Qualitative comparison with existing methods

Next, we illustrate the superiority of CPM over the existing methods through some update handling scenarios. YPK-CNN re-evaluates periodically every query q, even if the object updates have not affected any cell in its vicinity. This is due to the fact that it does not include a mechanism for detecting queries influenced by location updates. Furthermore, in the general case, YPK-CNN visits more cells than necessary when performing NN search for moving and new queries. Consider the 1-NN computation of query q in Figure 4.2a. As discussed in Section 2 (the example is the same as Figure 2.1), YPK-CNN processes 25 cells and six objects (p_1 up to p_6). Finally, it also incurs redundant computations for static queries. Assuming that in Figure 4.2b the current NN p_2 moves to location p'_2, YPK-CNN processes 49 cells and ten objects (p_1 up to p_{10}). Clearly, the unnecessary computations increase with $dist(p'_2,q)$. On the other hand, CPM (i) only processes queries whose influence region intersects some updated cell, and (ii) the NN computation and re-computation modules restrict the search space to the minimum number of cells around q (i.e., shaded cells in Figure 4.2).

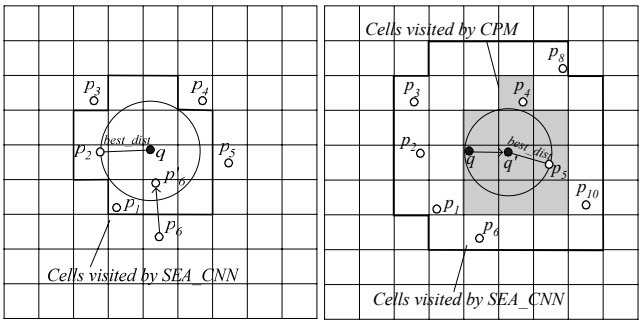


(a) NN search (b) Update handling

Figure 4.2: CPM versus YPK-CNN

SEA-CNN also performs redundant computations in several cases. First, assume that the only updates are from incoming objects and/or NNs that move within distance *best_dist* from q. For instance, in Figure 4.3a, p_6 moves closer to q than *best_dist*. SEA-CNN visits all cells intersecting the circle centered at q with radius $r = best\_dist$ and determines the new NN (p'_6) among the processed objects p_1, p_2 and p'_6. On the other hand, CPM directly compares $dist(p'_6,q)$ with *best_dist* and sets p'_6 as the result without visiting any cells. When k is larger, the computational waste of SEA-CNN increases because it considers a higher number of objects, even though there might be few changes in the

result. Another weak point of SEA-CNN concerns handling of outgoing NNs, which is similar to YPK-CNN. Recall that when p_2 moves to p'_2, SEA-CNN processes ten objects p_1 up to p_{10} (see Figure 2.2a), while CPM considers only four objects (see Figure 4.2b). SEA-CNN incurs higher cost than CPM also in the case that q changes position. In Figure 4.3b, assuming that q moves to q', CPM considers only cells intersecting the circle with center at q' and radius $dist(p_5,q')$, and retrieves the NN (p_5) by processing only two objects (p_4 and p_5) in total. SEA-CNN considers 33 cells and eight objects. A final remark about SEA-CNN is that it does not handle the case where some of the current NNs go off-line. On the contrary, CPM trivially deals with this situation by treating off-line NNs as outgoing ones.



(a) p_6 issues an update (b) q moves to q'

Figure 4.3: CPM versus SEA-CNN

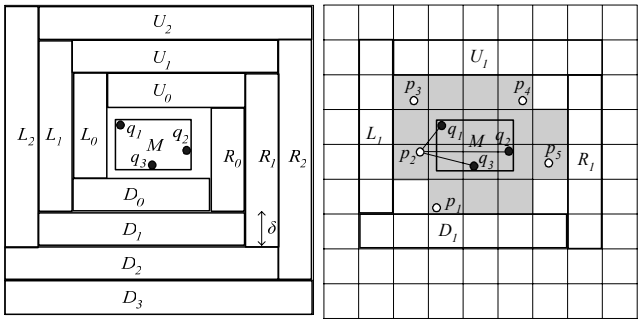
Summarizing, the speed of the objects does not affect the running time of CPM since update handling is restricted to the influence regions of the queries. On the other hand, the performance of both YPK-CNN and SEA-CNN (as also observed in [YPK05] and [XMA05]) degrades with object speed because the search region for a static query is determined by how far the furthest previous NN has moved since the last evaluation. For moving queries, CPM examines the minimum possible number of cells (which is independent of the query moving distance), whereas the cost of SEA-CNN increases with the velocity of q.

5. AGGREGATE NNS AND OTHER QUERY TYPES

In this section we extend the CPM algorithm to aggregate NN queries starting with the *sum* function. Given a set of query points $Q = \{q_1,q_2,\ldots,q_m\}$, a *sum* ANN query continuously reports the data object p that minimizes $adist(p,Q) = \sum_{q_i\in Q} dist(p,q_i)$. The basis of our method remains the conceptual partitioning of the space around the query Q. Since Q now consists of a set of query points, the partitioning applies to the space around the minimum bounding rectangle M of Q. Figure 5.1a exemplifies the partitioning into rectangles in the case of a 1-ANN query $Q = \{q_1,q_2,q_3\}$. We define $amindist(c,Q) = \sum_{q_i\in Q} mindist(c,q_i)$, which is a lower bound for the distance $adist(p,Q)$ of any object $p\in c$. The definition of $amindist(DIR_{lvl},Q)$ for a rectangle DIR_{lvl} is similar. The cell processing order is derived by corollary 5.1, which is based on the same geometric observations as Lemma 3.1 (and, hence, we omit its proof).

Corollary 5.1 (*f=sum*): For rectangles DIR_j and DIR_{j+1} of the same direction DIR with level numbers j and $j+1$, it holds that $amindist(DIR_{j+1},Q) = amindist(DIR_j,Q) + m\cdot\delta$, where m is the number of points in Q.

The ANN search module of CPM is essentially the same as the algorithm in Figure 3.4. The difference is that in the beginning of the search, we en-heap (in line 4) all cells c intersecting M. The sorting key is $amindist(c,Q)$ and $amindist(DIR_{lvl},Q)$ for the en-heaped cells and rectangles, respectively. When an object p is processed, we compute $adist(p,Q)$ and update accordingly the list of best ANNs found so far (i.e., *best_NN*). The algorithm terminates when the next entry in H has *amindist* greater than or equal to *best_dist*. In our example, the algorithm terminates with p_2 as the result, after processing all the shaded cells in Figure 5.1b. Similar to Section 3.1, the influence region of Q is the set of cells c with $amindist(c,Q)\leq best\_dist$; only updates affecting these cells can change the ANN result. Note that the influence region of a query is no longer a circle, but has an irregular shape (i.e., the shaded region in Figure 5.1b). Update handling is the same as in Section 3, the difference being that we use the aggregate distance function instead of the Euclidean one.



(a) Partitioning into rectangles (b) Influence region

Figure 5.1: ANN monitoring for *f=sum*

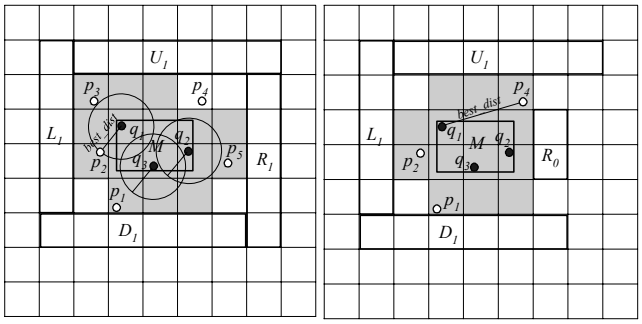
When *f=min*, an ANN query Q retrieves the object(s) in P with the smallest distance(s) from any point in Q. The ANN search considers cells and rectangles in ascending *amindist* order. For a cell c, $amindist(c,Q) = \min_{q_i\in Q} mindist(c,q_i)$, while for a rectangle DIR_{lvl}, $amindist(c,DIR_{lvl}) = \min_{q_i\in Q} mindist(DIR_{lvl},q_i)$. Corollary 5.2 dictates the cell processing order.

Corollary 5.2 (*f=min* or *f=max*): For rectangles DIR_j and DIR_{j+1} of the same direction DIR with level numbers j and $j+1$, it holds that $amindist(DIR_{j+1},Q) = amindist(DIR_j,Q) + \delta$.

The ANN search and update handling modules of CPM are similar to the *sum* case. Furthermore, for the *min* function, we can improve the O(m) time required to compute $amindist(DIR_0,Q)$ to O(1). The MBR M of Q contains by definition one point of Q on each edge. Therefore, computing $amindist(DIR_0,Q)$ for each direction DIR reduces to calculating the minimum distance between rectangle DIR_0 and the closest edge of M. For example, $amindist(D_0,Q)$ equals to the distance between the top edge of D_0 and the bottom edge of M. An interesting observation about the *min* aggregate function is that the influence region of Q contains cells that intersect at least one of the circles centered at some q_i with radius *best_dist*. Figure 5.2a shows an example where $Q = \{q_1,q_2,q_3\}$ and *f=min*. The result of the query is p_2, and the influence region of Q appears shaded.

When *f=max*, CPM monitors the object(s) of P that have the lowest maximum distance(s) from points in Q. For each cell c, $amindist(c,Q) = \max_{q_i\in Q} mindist(c,q_i)$, while for each boundary box DIR_{lvl}, $amindist(DIR_{lvl},Q) = \max_{q_i\in Q} mindist(DIR_{lvl},q_i)$. Corollary 5.2 holds also in the case of *max*, whereas computing $amindist(DIR_0,Q)$ for each direction DIR can be performed in

O(1) time: $amindist(DIR_0,Q)$ equals the minimum distance between DIR_0 and the opposite edge of M. In Figure 5.2b we illustrate the case where $Q = \{q_1,q_2,q_3\}$ and $f=max$. The result of the query is object p_4, and the corresponding influence region consists of the shaded cells.



(a) $f=min$ (b) $f=max$

Figure 5.2: ANN monitoring for $f=min$ and $f=max$

Finally, CPM can easily handle constrained variations of NN (and ANN) search that retrieve the NNs of a query point in a user-specified area of the data space. Ferhatosmanoglu et al. [FSAA01] propose algorithms for static datasets indexed by R-trees. The adaptation of CPM to this problem inserts into the search heap only cells and conceptual rectangles that intersect the constraint region. Assume, for instance, that in Figure 5.3 we want to monitor the NN to the *northeast* of q. CPM en-heaps only the cells $c_{4,4}$, $c_{4,5}$, $c_{5,4}$, $c_{5,5}$ and rectangles U_0, R_0, U_1, R_1. Inside $c_{5,5}$, object p_3 is identified as the NN. Note that object p_1 (the unconstrained NN) is not encountered at all since its cell is not visited, whereas p_2 is processed but not reported.

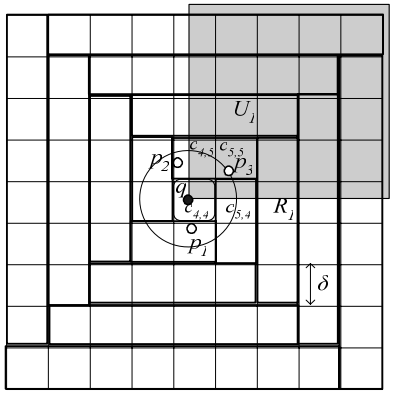


Figure 5.3: Monitoring of a constrained NN query

6. EXPERIMENTAL EVALUATION

In this section we evaluate the performance of CPM and compare it with YPK-CNN and SEA-CNN. In accordance with the experimental study of [XMA05], our datasets are created with the spatiotemporal generator of [B02]. The input of the generator is the road map of Oldenburg (a city in Germany). The output is a set of objects (e.g., cars, pedestrians) moving on this network, where each object is represented by its location at successive timestamps. An object appears on a network node, completes the shortest path to a random destination, and then disappears. We use the default velocity values of the generator for *slow*, *medium*, and *fast* object speeds. Objects with slow speed cover a distance that equals 1/250 of the sum of the workspace extents per timestamp. Medium and fast speeds correspond to distances that are 5 and 25 times larger, respectively. The NN queries are generated similarly, i.e., they are objects moving on the same network, but they stay in the system throughout the simulation. The queries are evaluated at every timestamp and the simulation length is 100 timestamps. In the implementation of SEA-CNN, we use the NN search algorithm of YPK-CNN to compute the initial results of the queries, or to retrieve the new NN sets when some of the current NNs disappear. Table 5.1 summarizes the parameters under investigation, along with their ranges and default values. In each experiment we vary a single parameter, while setting the remaining ones to their default values. For all simulations we use a Pentium 2.4 GHz CPU with 1 GByte memory.

Parameter	Default	Range
Object population (N)	100K	10, 50,100,150,200 (K)
Number of queries (n)	5K	1,2,5,7,10 (K)
Number of NNs (k)	16	1,4,16,64,256
Object/Query speed	medium	slow, medium, fast
Object agility (f_{obj})	50%	10,20,30,40,50 (%)
Query agility (f_{qry})	30%	10,20,30,40,50 (%)

Table 6.1: System parameters (ranges and default values)

Initially, we generate 5K queries and 100K objects, according to the default parameters of Table 6.1. We process the queries with each monitoring algorithm, and measure the overall running time by varying the grid granularity. Figure 6.1 illustrates the results for grid sizes ranging between 32×32 and 1024×1024. CPM clearly outperforms both competitors for all grid sizes. SEA-CNN is worse than YPK-CNN because it incurs unnecessary computations for moving queries, as explained in Section 4.2. A 128×128 grid (i.e., $\delta = 1/128$) constitutes a good tradeoff between the CPU time and the space requirements for all methods[6]. Therefore, we perform the remaining experiments using $\delta = 1/128$.

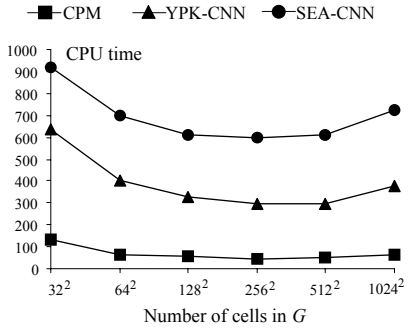
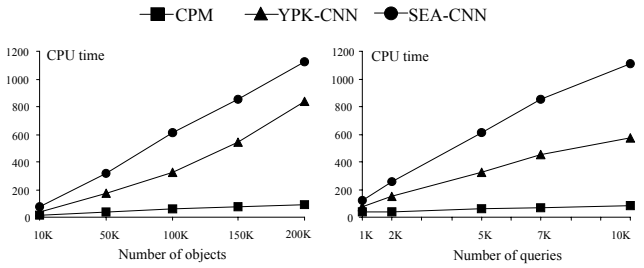


Figure 6.1: CPU time versus grid granularity

Next we examine scalability issues. Figure 6.2a measures the effect of the object population N on the running time. The generator is tuned so that the average object population during the simulation equals the desired value N. Similarly, Figure 6.2b illustrates the CPU overhead as a function of the number n of queries in the system. The cost of all algorithms increases linearly

[6] The space overhead is 2.854 MBytes, 3.074 MBytes, and 3.314 MBytes for YPK-CNN, SEA-CNN and CPM, respectively.

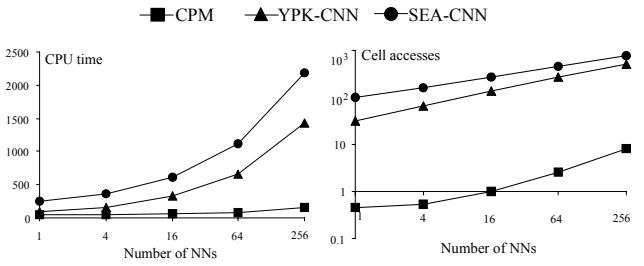
to both N and n. However, YPK-CNN and SEA-CNN are much more sensitive than CPM to these parameters, confirming the scalability of our approach.



(a) Effect of N (b) Effect of n

Figure 6.2: CPU time versus N and n

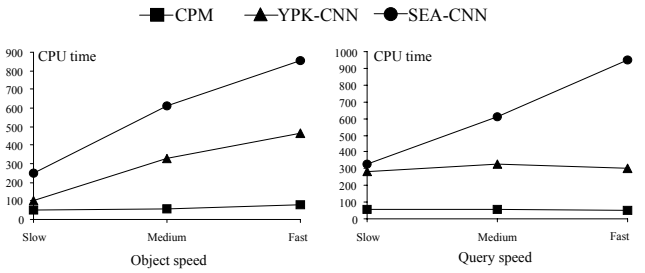
Figure 6.3a shows the CPU time as a function of the number k of NNs (using the default values for the remaining parameters). Figure 6.3b plots (in logarithmic scale) the number of cell accesses per query per timestamp. A cell visit corresponds to a complete scan over the object list in the cell. Note that a cell may be accessed multiple times within a cycle, if it is involved in the processing of multiple queries. For CPM, cell accesses occur during the NN computation algorithm (for moving queries), and during NN re-computation (for stationary queries, when there are more outgoing NNs than incomers). YPK-CNN re-evaluates the queries in every timestamp, and therefore induces cell visits for each query in every processing cycle. SEA-CNN accesses cells whenever some update affects the answer region of a query and/or when the query moves. CPM significantly outperforms its competitors because: (i) it does not search the grid if the update information suffices to maintain the results, and (ii) even if the updates necessitate computation from scratch or re-computation of the NN sets, CPM processes the minimal number of cells. An interesting observation is that for k=1 and k=4, CPM accesses less than one cell per query on the average. This happens because queries of case (ii) have a small cost (i.e., 1-2 cell visits), which is counter-balanced by queries of case (i) that do not incur any visits.



(a) CPU time (b) Cell accesses

Figure 6.3: Performance versus k

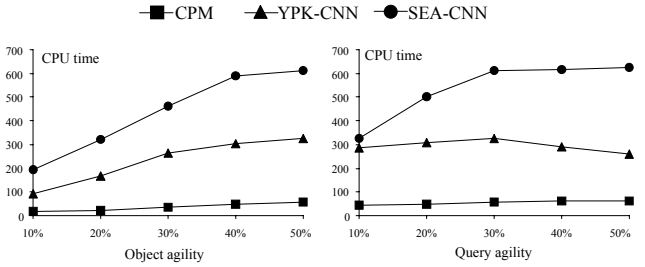
Figure 6.4a illustrates the CPU time with respect to the object speed. The performance of CPM is practically unaffected by the speed of objects. On the contrary, both YPK-CNN and SEA-CNN degenerate when objects move fast, as anticipated in Section 4.2. Figure 6.4b depicts the effect of the query speed on the running time of the algorithms. The cost of CPM and YPK-CNN is independent of the query velocity, since both techniques compute the results of the moving queries from scratch. On the other hand, SEA-CNN is negatively affected because, as discussed in Section 4.2, the search region grows when the queries move far from their previous position, increasing the number of computations.



(a) Effect of object speed (b) Effect of query speed

Figure 6.4: CPU time versus object and query speed

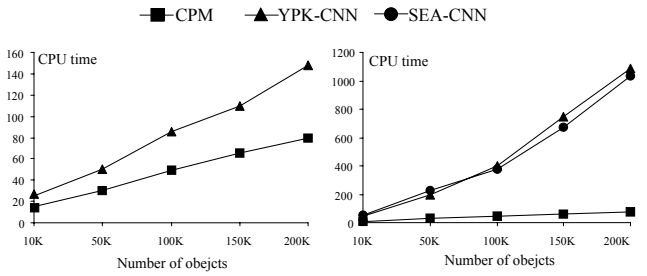
Figure 6.5a compares the performance of CPM, YPK-CNN and SEA-CNN versus the percentage of objects that move within a timestamp (i.e., the object agility f_{obj}). As expected (see Section 4.1), the running time of CPM scales linearly with the object agility, due to the increasing index update cost. In order to quantify the effect of the query agility f_{qry} (i.e., the probability that a query moves within a timestamp), we vary f_{qry} from 10% to 50% and keep the remaining parameters fixed to their default values. As shown in Figure 6.5b, the CPU time of CPM increases linearly with f_{qry} because NN computations (for moving queries) are more expensive than result maintenance for static queries. Note that YPK-CNN is rather insensitive to the query agility because the incremental maintenance of the NN set (for stationary queries) has similar cost to the two-step NN computation (for moving queries).



(a) Effect of object agility (f_{obj}) (b) Effect of query agility (f_{qry})

Figure 6.5: CPU time versus object and query agility

In the remaining two experiments, we compare individually the NN computation and result maintenance modules of the alternative methods. First, we monitor 5K constantly moving queries (i.e., queries that issue location updates in every timestamp), while varying the object population N. The query results are computed from scratch at every processing cycle; therefore, we can study the efficiency of the NN computation modules. SEA-CNN is omitted (since it does not include an explicit mechanism for obtaining the initial NN set). As shown in Figure 6.6a, CPM outperforms YPK-CNN and the performance gap increases with N. Finally, we process 5K static queries (i.e., f_{qry}=0%), while varying the object population N. This way we eliminate the NN computations from scratch (apart from the initial query evaluation) and measure the pure result maintenance cost. As shown in Figure 6.6b, the behavior of YPK-CNN and SEA-CNN is similar, while CPM induces considerably fewer computations.



(a) Constantly moving queries (b) Static queries

Figure 6.6: CPU time for constantly moving and static queries

7. CONCLUSIONS

This paper investigates the problem of monitoring continuous NN queries over moving objects. The task of the query processor is to constantly report the results of all queries, as location updates stream by from both the objects and the queries. Our contribution is an efficient processing method, referred to as the conceptual partitioning monitoring (CPM) algorithm. CPM is based on a conceptual partitioning of the space around each query q, in order to restrict the NN retrieval and the result maintenance computations to objects that lie in the vicinity of q. The core of CPM is its NN computation module, which retrieves the first-time results of incoming queries, and the new results of existing queries that change location. This module produces and stores book-keeping information to facilitate fast update handling. Keeping the NN set of a query q up-to-date is performed by processing on-line the object updates as they arrive. If the new NN set of a query can be determined solely by the previous result and the set of updates, then access to the object grid G is avoided. Otherwise, CPM invokes the NN re-computation module, which uses the book-keeping information stored in the system to reduce the running time (compared to NN computation from scratch). CPM is a generally applicable technique, since it does not require any knowledge about the object or query moving patterns (e.g., velocity vectors), and can concurrently process multiple (static or moving) queries. We analyze its performance and compare it with the existing state-of-the-art methods. As demonstrated by a qualitative analysis and by an extensive experimental study, CPM outperforms its competitors.

Finally, to support the generality of the proposed methodology, CPM is applied to aggregate NN monitoring, where a query consists of a set of points and the optimization goal depends on an aggregate function (such as *sum*, *min* and *max*). In the future, we intend to explore the problem of continuous monitoring for variations of NN search, such as reverse NNs. A preliminary approach on this topic considers one-dimensional streams and aggregate reverse NN [KMS02]. It would be interesting to develop alternative approaches for the continuous monitoring of multiple (conventional) reverse NN queries in spaces of higher dimensionality.

ACKNOWLEDGEMENTS

This work was supported by grant HKUST 6180/03E from Hong Kong RGC. The authors would like to thank Kevin Di Filippo for proof-reading the paper.

REFERENCES

[B02] Brinkhoff, T. A Framework for Generating Network-based Moving Objects. *GeoInformatica*, (6)2: 153-180, 2002.

[BJKS02] Benetis, R., Jensen, C., Karciauskas, G., Saltenis, S. Nearest Neighbor and Reverse Nearest Neighbor Queries for Moving Objects. *IDEAS*, 2002.

[CHC04] Cai, Y., Hua, K., Cao, G. Processing Range-Monitoring Queries on Heterogeneous Mobile Objects. *MDM*, 2004.

[FSAA01] Ferhatosmanoglu, H., Stanoi, I., Agrawal, D., Abbadi, A. Constrained Nearest Neighbor Queries. *SSTD*, 2001.

[GL04] Gedik, B., Liu, L. MobiEyes: Distributed Processing of Continuously Moving Queries on Moving Objects in a Mobile System. *EDBT*, 2004.

[H84] Henrich, A. A Distance Scan Algorithm for Spatial Access Structures. *ACM GIS*, 1984.

[HS99] Hjaltason, G., Samet, H. Distance Browsing in Spatial Databases. *ACM TODS*, 24(2): 265-318, 1999.

[KMS02] Korn, F., Muthukrishnan, S. Srivastava, D. Reverse Nearest Neighbor Aggregates Over Data Streams. *VLDB*, 2002.

[KOTZ04] Koudas, N., Ooi, B., Tan, K., Zhang, R. Approximate NN queries on Streams with Guaranteed Error/performance Bounds. *VLDB*, 2004.

[MXA04] Mokbel, M., Xiong, X., Aref, W. SINA: Scalable Incremental Processing of Continuous Queries in Spatio-temporal Databases. *SIGMOD*, 2004.

[PSTM04] Papadias, D., Shen, Q., Tao, Y., Mouratidis, K. Group Nearest Neighbor Queries. *ICDE*, 2004.

[PXK+02] Prabhakar, S., Xia, Y., Kalashnikov, D., Aref, W., Hambrusch, S. Query Indexing and Velocity Constrained Indexing: Scalable Techniques for Continuous Queries on Moving Objects. *IEEE Transactions on Computers*, 51(10): 1124-1140, 2002.

[RKV95] Roussopoulos, N., Kelly, S., Vincent, F. Nearest Neighbor Queries. *SIGMOD*, 1995.

[SR01] Song, Z., Roussopoulos, N. K-Nearest Neighbor Search for Moving Query Point. *SSTD*, 2001.

[SRAA01] Stanoi, I., Riedewald, M., Agrawal, D., Abbadi, A. Discovery of Influence Sets in Frequently Updated Databases. *VLDB*, 2001.

[TP03] Tao, Y., Papadias, D. Spatial Queries in Dynamic Environments. *ACM TODS*, 28(2): 101-139, 2003.

[XMA05] Xiong, X., Mokbel, M., Aref, W. SEA-CNN: Scalable Processing of Continuous K-Nearest Neighbor Queries in Spatio-temporal Databases. *ICDE*, 2005.

[YPK05] Yu, X., Pu, K., Koudas, N. Monitoring K-Nearest Neighbor Queries Over Moving Objects. *ICDE*, 2005.

[ZZP+03] Zhang, J., Zhu, M., Papadias, D., Tao, Y., Lee, D. Location-based Spatial Queries. *SIGMOD*, 2003.