Continuous Nearest Neighbor Monitoring in Road Networks

Kyriakos Mouratidis[1] Man Lung Yiu[2] Dimitris Papadias[3] Nikos Mamoulis[2]

[1]School of Information Systems
Singapore Management University
80 Stamford Road, Singapore 178902
kyriakos@smu.edu.sg

[2]Department of Computer Science
University of Hong Kong
Pokfulam Road, Hong Kong
{mlyiu2, nikos}@cs.hku.hk

[3]Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
dimitris@cs.ust.hk

ABSTRACT

Recent research has focused on continuous monitoring of nearest neighbors (NN) in highly dynamic scenarios, where the queries and the data objects move frequently and arbitrarily. All existing methods, however, assume the Euclidean distance metric. In this paper we study k-NN monitoring in road networks, where the distance between a query and a data object is determined by the length of the shortest path connecting them. We propose two methods that can handle arbitrary object and query moving patterns, as well as fluctuations of edge weights. The first one maintains the query results by processing only updates that may invalidate the current NN sets. The second method follows the shared execution paradigm to reduce the processing time. In particular, it groups together the queries that fall in the path between two consecutive intersections in the network, and produces their results by monitoring the NN sets of these intersections. We experimentally verify the applicability of the proposed techniques to continuous monitoring of large data and query sets.

1. INTRODUCTION

Nearest neighbor search is one of the fundamental problems in the field of spatial databases. A k-NN query computes the k data objects that lie closest to a given query point. Early methods are restricted to *snapshot* queries over static data. Due to the wide availability of positioning devices and the rise of location-based services, recently the research focus has shifted to *continuous* k-NN (CkNN) queries over mobile data. These queries run for long time periods and demand constant update of the results. Existing techniques for CkNN monitoring (e.g., [16, 25, 26]) aim at Euclidean spaces. In most real-world scenarios, however, the users (i.e., queries) and the data objects are restricted to move in a transportation network. Typically, the edges of the network are road segments and their weights correspond to their lengths, or to the time required to travel them. In this context, the distance between two objects is defined as the length (i.e., sum of weights) of the shortest path connecting them. Although several papers (e.g., [4, 13, 18]) study different versions of snapshot k-NN queries in road networks, there is no previous work on continuous monitoring.

This paper constitutes the first attempt on this important problem. We assume a central server that monitors the positions of CkNN queries and objects, as well as the current edge weights, which may fluctuate over time (e.g., due to varying traffic conditions). The task of the server is to continuously compute and update the result of each query. We propose fast algorithms to perform this task and provide answers in real-time. The server does not have any knowledge about the object/query velocity vectors and trajectories. Furthermore, since weights may fluctuate, some results may change even though the objects and the queries have remained static. This situation does not occur in the Euclidean version of the problem.

Efficient CkNN monitoring in networks is crucial for many applications including location-based services and mobile computing. As an example of a real-world scenario, consider that the queries correspond to vacant cabs, and the data objects are pedestrians that ask for a taxi. As cabs and pedestrians move, each free taxi driver wishes to know his/her k closest clients in terms of traveling time. As the reverse case, clients looking for available taxis may constitute the queries, and taxis the data objects. Clearly, in both cases the Euclidean k-NNs are useless, since the path between a cab and a pedestrian is restricted by the underlying road network. Furthermore, both the data objects and the queries are highly dynamic and unpredictable (e.g., taxis may move fast, new clients may appear or existing customers disappear), invalidating existing snapshot k-NN techniques for road networks.

Similar to most on-line monitoring systems, we assume main-memory evaluation. Our first contribution is the *incremental monitoring algorithm* (IMA). IMA retrieves the initial result of a query q by expanding the network around

VLDB '06, September 12-15, 2006, Seoul, Korea.

it until k NNs are found. To facilitate processing of subsequent updates, it stores the shortest paths (from q) to the network nodes encountered during the NN search in the form of an *expansion tree*. The main idea is that only updates from objects and edges falling in the expansion tree can alter the NN set of q; irrelevant updates are simply ignored. On the other hand, when the updates affect the result of q or when q moves to a new location, IMA determines the part of the expansion tree that remains valid, and re-uses it to accelerate the computation of the new NNs of q.

Our second contribution is the *group monitoring algorithm* (GMA). GMA is based on the following observation. Consider a path between two consecutive intersection nodes in the network. The k-NNs of any query in the path belong to the union of the data objects falling in the path and the k-NN sets of its two endpoints. Thus, GMA monitors the k-NNs of the intersections (using IMA) and utilizes them to efficiently compute the results of all the queries in the path. In summary, GMA benefits from (i) the shared execution among queries in the same path, and (ii) the reduction of the problem from monitoring moving queries to (monitoring) static network nodes.

The rest of the paper is organized as follows. Section 2 reviews related work in snapshot NN algorithms and spatial query monitoring. Section 3 states our assumptions and describes the basic data structures. Sections 4 and 5 present IMA and GMA, respectively. Section 6 experimentally compares the proposed methods through simulations in real road networks. Finally, Section 7 concludes the paper with a summary and directions for future work.

2. RELATED WORK

Section 2.1 surveys snapshot NN methods, and Section 2.2 focuses on continuous monitoring of spatial queries in the Euclidean space.

2.1 Snapshot NN Methods

The first k-NN algorithms in the database literature process snapshot (i.e., one-time) queries over static objects, assuming that the distance function is some Minkowski metric (such as the Euclidean). They index the data with a spatial access method (e.g., an R-tree [7]) and utilize distance bounds between the index nodes and the query point to restrict the search space [8, 20]. Subsequent research considered k-NN queries in client-server architectures. The general idea is to provide the client with extra information (along with the k-NN set) in order to reduce the number of subsequent queries for updating the result. Assuming static data objects, the method of [28] returns to the client the validity time of the current k-NN result, considering his/her maximum possible velocity. Zhang et al. [27] improve upon this method by computing the region around the client that the NN set remains the same. Song and Roussopoulos [23] report more than k NNs so that if the client moves within a maximum distance, the new k NNs can be computed among the returned objects. Tao and Papadias [24] propose *time-parameterized* k-NN queries for clients and objects that move linearly with constant velocity. Such queries return the current result along with its validity period and its next change. [1, 24] describe algorithms for computing *linear* NNs. A linear query returns all the k-NN sets of the client up to a future timestamp, provided that the velocity vectors in the system do not change.

The above methods target spaces where the distance between objects and queries is defined as a function of their coordinates. Hence, they are inapplicable to road networks, where the distance also depends on the connectivity and weights of the underlying road segments (edges). Jensen et al. [11] formalize the problem of k-NN search in road networks and present a system prototype for such queries. Papadias et al. [18] describe a framework that integrates network and Euclidean information, and answers k-NN, range, closest pairs, and e-distance join queries. They index the data objects with an R-tree and utilize connectivity and location information to guide the search. Kolahdouzan and Shahabi [13] retrieve the k-NNs based on pre-computed network Voronoi cells. The first NN of any query falling in such a polygon is the corresponding data object. If additional NNs are required, the search considers the adjacent Voronoi polygons iteratively. Shahabi et al. [21] propose an embedding technique that approximates the network distance with computationally simple functions in order to retrieve fast, but approximate, k-NN results.

Several papers study alternative forms of NN search in road networks. Given a static dataset (e.g., gas stations) and a query path, Shekhar and Yoo [22] retrieve the object (station) that causes the shortest detour if included in the path. Given a user-specified trajectory, a *path* k-NN query retrieves the k-NNs (in terms of network distance) at any point in the trajectory. Assume that we know the $k+1$ NNs at some point o in the query trajectory, and that their network distance from o is $dist_i$, for $i = 1, \ldots, k+1$. Let δ be the smallest difference between the distances of two consecutive NNs, i.e., $\delta = \min_{1 \leq i \leq k}\{dist_{i+1} - dist_i\}$. Kolahdouzan and Shahabi [12] propose a path NN algorithm based on the observation that any point in the query trajectory within distance $\delta/2$ from o has exactly the same NN set as o. Cho and Chung [4] solve the same problem, by retrieving the k-NN sets of all network nodes in the query path, and uniting them with the data objects falling in the path. It can be easily proven that the resulting set contains the k-NNs of any point in the query trajectory. Both path NN methods are inapplicable to continuous NN monitoring because they (i) only work for static data objects, and (ii) assume that the query trajectory is known in advance. In summary all existing techniques for NN processing in road networks require that objects and queries are either static or move with known velocities or trajectories. Furthermore, they can be characterized as "snapshot" methods since they return a single result and terminate.

2.2 Continuous Monitoring in the Euclidean Space

The existing research on continuous monitoring of spatial queries has focused exclusively on Euclidean spaces. *Q-index* [19] monitors static range queries over moving objects. It indexes the ranges using an R-tree and probes moving objects against the index in order to update the affected results. *Mobieyes* [6] and MQM [3] utilize the computational capabilities of the data objects to reduce the load at the central server. SINA [15] performs continuous evaluation of range queries using a three-step spatial join between moving objects and ranges.

Regarding k-NN monitoring, Koudas et al. [14] present a system for approximate k-NN queries over streams of multidimensional points. Currently, there exist three algorithms

for exact k-NN monitoring in the Euclidean space: YPK-CNN [26], SEA-CNN [25] and CPM [16]. All three methods index the data with a regular grid. To retrieve the initial result of a new query q, they search in the cells around it. To keep the result up-to-date in subsequent timestamps, YPK-CNN (SEA-CNN) computes the current distance d of the farthest previous NN and processes the objects falling in the square with side-length $2{\cdot}d$ (circle with radius d) centered at q. Result maintenance is different in CPM. Let C be the circle with center at q that encloses the previous NN locations. If the NNs that move outside C are more than the outer objects that move into C, then the query is recomputed from scratch. Otherwise, the k objects in C that lie closest to q form the new result.

The grid index used by YPK-CNN, SEA-CNN and CPM cannot capture the constraints imposed by a road network. Furthermore, these methods process objects and updates falling in circles and rectangles, while there is no trivial mapping or interpretation of these shapes into the network distance space. Finally, they cannot handle updates due to changes of the edge weights (even if the objects and the queries have remained static). Thus, they are inapplicable to road networks. Summarizing, despite the amount of related work in Euclidean spaces and the real need for efficient NN monitoring in road networks, currently there exists no method addressing the problem.

3. ASSUMPTIONS AND DATA STRUCTURES

We assume a workspace (e.g., a city road network) containing a set of data objects and a set of CkNN queries. The data objects are the entities of interest. The queries correspond to users that request continuous monitoring of their k closest objects. Both objects and queries move arbitrarily in the network. Whenever they change location, they issue an update to the monitoring server containing their id, their old coordinates and their new coordinates. The network is a graph consisting of nodes and edges. The edges have non-negative weights modeling, for example, the time required to travel from one endpoint to the other. For simplicity we consider that the edges are bidirectional, but our methods can be easily applied to networks with unidirectional edges (e.g., one-way roads or roads where the weight is different for the two directions). The weights fluctuate, depending on the traffic conditions. Whenever an edge weight changes, the server receives an update (e.g., issued by sensors monitoring the congestion level on the corresponding road segment). The network distance between a query and an object is defined as the total cost (i.e., the sum of edge weights) along the shortest path connecting them.

In our system, information about the network, the objects, and the queries is stored in three memory-resident data structures. The first one is a spatial index SI on the network edges. Given the coordinates of an object p, we use SI to identify the edge where p lies. For this purpose we use a PMR quadtree [9]. Each leaf quad contains the ids of the edges intersecting it. The tree is built by iteratively inserting the network edges. If the number of edge ids in a leaf quad exceeds a threshold, it is split into four new ones and becomes their predecessor in SI. In order to retrieve the edge containing an object p, we descend SI top-down until we reach the leaf quad covering p. Among the edges intersecting this quad, we identify the one that contains p.

The second structure is the edge table ET, which maintains network and data object information. It is a hash-table on edge id, that stores for each edge e: (i) its endpoints (network nodes) $e.start$ and $e.end$, (ii) the sets of edges adjacent to each of its endpoints, (iii) its weight $e.w$, (iv) the list of objects currently in e, and (v) the *influence list* $e.IL$ containing the queries affected by e, along with the corresponding influencing intervals. To exemplify the information stored in $e.IL$, consider Figure 1, where q is a 3-NN query with NNs p_1, p_2, and p_3 (at distances 3, 5, and 7, respectively). The network distance $d(p_3, q) = 7$ of the furthest NN is denoted by $q.kNN\_dist$. An edge e *affects* q, if it contains an interval where the network distance is less than $q.kNN\_dist$. We call this interval the *influencing interval* of e. In our example, edge n_2n_5 affects q and the influencing interval is the entire edge. Edge n_5n_4 also affects q; its influencing interval starts at n_5 and ends at the vertical mark with distance $q.kNN\_dist$ (all the points to the right of the mark have distance larger than $q.kNN\_dist$). We store q in the influence list of each affecting edge e, together with the corresponding influencing interval. We use the influence list information to process only object and edge updates that affect the result of q and ignore the rest.

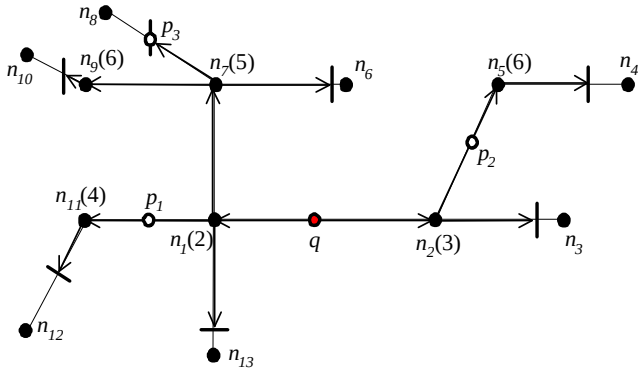


Figure 1: Affecting edges and expansion tree (3-NN query)

The third data structure is the query table QT, which maintains information about the queries in the system. QT stores for each query q (i) its coordinates, (ii) the number $q.k$ of required NNs, (iii) its current k-NN set $q.result$, and (iv) its *expansion tree* $q.tree$. The expansion tree of q is a tree rooted at q that contains the shortest path between q and every node in the network with distance less than or equal to $q.kNN\_dist$. The tree reaches up to the marks defined by $q.kNN\_dist$. Recall that $q.kNN\_dist$ is the distance between q and its kth NN. The bold arrows in Figure 1 show $q.tree$ in our running example. The numbers in parentheses are the distances of the nodes in $q.tree$. The expansion tree information is used to facilitate handling of query movements and edge weight changes. Computing and maintaining $q.tree$ is discussed in the next section. To conclude Section 3, we present in Table 1 the primary symbols used in the paper, along with their interpretation. The asterisk next to a definition indicates that the corresponding symbol or structure will be introduced in Section 5.

Symbol	Description
SI	spatial index
ET	edge table
QT	query table
NT	active node table*
ST	sequence table*
$d(p,q)$	network distance between object p and query q
$e.IL$	influence list of edge e
$q.k$	number of NNs required by query q
$q.result$	k-NN set of q
$q.kNN\_dist$	distance between q and its kth NN
$q.tree$	expansion tree of q
$n.k$	number of NNs required by active node n*
$n.S$	set of sequences with n as an endpoint*
$n.Q$	set of queries falling in the sequences in $n.S$*

Table 1: Primary symbols

4. INCREMENTAL MONITORING ALGORITHM

In this section we describe the incremental monitoring algorithm (IMA). Section 4.1 discusses the initial result computation of a query arriving at the system. Regarding result maintenance, for the sake of presentation we isolate each type of updates. Sections 4.2 and 4.3 focus on object and query updates, respectively. Section 4.4 deals with edge weight updates. Finally, Section 4.5 describes the complete IMA, which considers all three types of updates and determines their processing order.

4.1 Initial Result Computation

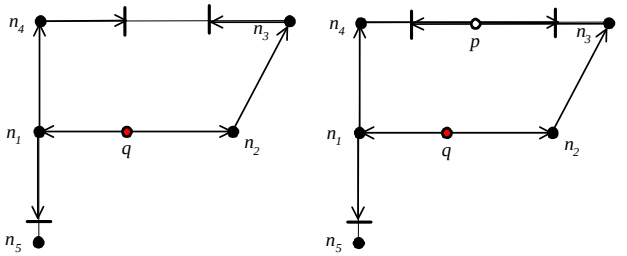
When a new query q arrives at the system, IMA retrieves its initial k-NN set with a technique based on Dijkstra's algorithm [5]. Specifically, it expands the network around q up to distance $q.kNN\_dist$, building at the same time the expansion tree of q and updating the influence lists of the affecting edges. Figure 2 shows the pseudo-code for the initial result computation, assuming that e is the edge containing q. First, IMA initializes an empty min-heap H that organizes the encountered nodes and inserts into $q.result$ the objects in e. Next, it en-heaps in H the endpoints of e with keys equal to the corresponding fraction of $e.w$. It sets q as the root of the expansion tree $q.tree$, with children the endpoints of e. Then, it iteratively de-heaps nodes from H; we say that each de-heaped node n is *verified*, implying that we know its distance $d(n,q)$. For each de-heaped node n, IMA considers the objects in the adjacent edges, and updates the current k-NN set if necessary. Next, it checks each un-verified node n_{adj} that is directly reachable via n. If n_{adj} is encountered for the first time, it is inserted into H with key $dist = d(n,q) + nn_{adj}.w$ and included in $q.tree$ as a child of n. Otherwise, n_{adj} is already in H; if its current key is larger than $dist$, its value is decreased to $dist$ and n_{adj} is made a child of n in $q.tree$.

The search terminates when the next node in H has key larger than $q.kNN\_dist$. The expansion tree built during the retrieval of the initial result contains the shortest path to each verified node. An interesting observation regards the marks of affecting edges that are not included in any shortest path (e.g., n_2n_3 in Figure 1). To compute these marks, we maintain these edges in set $partial\_edges$. Lines 12-13 insert each encountered edge into $partial\_edges$. Line 11 removes from $partial\_edges$ the edges that are part of some shortest path; when verifying a node, we know that the shortest path to it passes through the edge connecting

```
Initial Result (q)
1.  Initialize empty heap H; Set q.kNN_dist=∞, partial_edges=∅
2.  Set q as the root of the expansion tree q.tree
3.  Let e be the edge containing q
4.  Insert the k best objects in e into q.result; Update q.kNN_dist
5.  En-heap e.start and e.end with key/distance according to e.w
6.  Insert sub-edges qe.start and qe.end into partial_edges
7.  While the next node n in H has key d(n,q) < q.kNN_dist
8.    De-heap n //n is verified
9.    Let e be the edge between n and its predecessor
10.   Insert q into e.IL (with influencing interval the entire edge)
11.   Delete e from partial_edges
12.   For each adjacent node n_adj of n except for its predecessor
13.     Insert nn_adj into partial_edges
14.     Update q.result and q.kNN_dist with objects on edge nn_adj
15.     If n_adj has not been de-heaped before //n_adj is not verified
16.       dist=d(n,q)+nn_adj.w //distance of n_adj if reached via n
17.       If n_adj is not in H
18.         Set n as the predecessor of n_adj (in the expansion tree)
19.         Insert n_adj into H with key dist
20.       Else //i.e., n_adj is already in H
21.         If dist is less than the current key of n_adj in H
22.           Set n as the predecessor of n_adj (in the expansion tree)
23.           Update the key of n_adj in H to dist
24. Delete from q.tree the un-verified nodes
25. For each edge e in partial_edges
26.   Compute the mark(s) on e at distance q.kNN_dist from q
27.   Insert the mark(s) as leafs in q.tree
28.   Insert q into e.IL along with the influencing interval(s)
```

Figure 2: The initial result algorithm of IMA

it with its predecessor in $q.tree$. In lines 25-27, we compute for all edges in $partial\_edges$ the influencing interval(s) and include the corresponding mark(s) in the expansion tree. Note that an edge in $partial\_edges$ might have two marks (equivalently, two influencing intervals) if both its endpoints are verified. For example, in Figure 3(a), edge n_3n_4 is not part of $q.tree$ and it has two influencing intervals. The influence list information is stored into ET in lines 10 (when the entire e affects q) and 28 (when part(s) of e affect q). Finally, note that an object (falling in edge e) is considered twice for inclusion in $q.result$, if both endpoints of e are verified in line 8, but e is not part of $q.tree$. In the example of Figure 3(b), object p is encountered when de-heaping n_3 and when de-heaping n_4. To avoid reporting objects like p twice, when inserting an object into $q.result$, we check if it already exists therein, and keep only the instance with the smallest distance.



(a) Non-overlapping intervals (b) Overlapping intervals

Figure 3: Edge n_3n_4 has two marks/influencing intervals

4.2 Processing of Object Updates

For simplicity, assume that there is only one query q in the system. An object update contains the object id $p.id$,

and its old and new coordinates, p_{old} and p_{new} respectively. Our first task is to delete p from the object list of its old edge e_{old} and insert it into the object list of its new edge e_{new} (we identify e_{old} and e_{new} using SI). Concerning result maintenance, the crucial observation is that an object movement can alter the result of q only if $d(p_{old}, q) \leq q.kNN\_dist$ or $d(p_{new}, q) \leq q.kNN\_dist$. In other words, we can ignore all objects further away than $q.kNN\_dist$. To identify the updates that affect q, we utilize the influence list information. In particular, we only process the ones where $e_{old}.IL$ or $e_{new}.IL$ contain q and the corresponding influencing intervals include p_{old} and p_{new}, respectively. Figure 4 continues the example of Figure 1 where, after the initial result retrieval, four object updates arrive at the system. Among them, we only consider the movements of p_1, p_3, p_4 and ignore that of p_5; even though the new location p_5' of p_5 falls in an affecting edge, the corresponding influencing interval does not include p_5'.

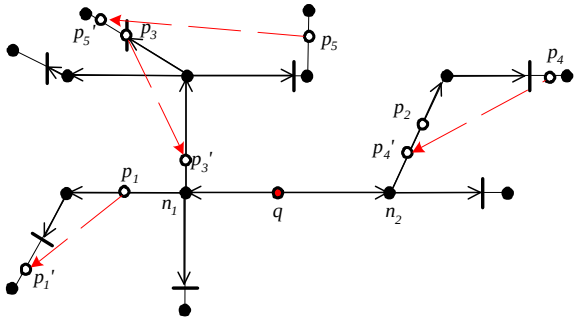


Figure 4: Four object updates (3-NN query)

We distinguish three types of object updates that affect q. The first type is current NNs that move within distance $q.kNN\_dist$ from q (e.g., object p_3 in Figure 4). The second type is *incoming* objects. Incoming objects used to lie further than $q.kNN\_dist$ but their new location is closer to q than $q.kNN\_dist$ (e.g., object p_4). The third type is *outgoing* objects. These are current NNs that move further away than $q.kNN\_dist$ from q (e.g., object p_1). In general, there are multiple updates of all three types at every timestamp. There are two result maintenance scenarios, depending on the relative number of incoming and outgoing objects.

In case that the outgoing objects are no more than the incoming ones, there are at least k objects within distance $q.kNN\_dist$. Furthermore, (i) the current NNs that did not move have the same distance as stored in $q.result$, and (ii) the distances of NNs that move within $q.kNN\_dist$ and of incoming objects can be retrieved by utilizing the information maintained in $q.tree$. For example, we compute the distances of p_3' and p_4' according to $d(n_1, q)$ and $d(n_2, q)$, respectively, and the weights of their containing edges. To determine the new result of q, we first remove from $q.result$ the outgoing NNs. Then, we form the union of the remaining NNs and the incoming objects, and report as $q.result$ the k best objects among them. Note that the new $q.kNN\_dist$ is smaller than or equal to the old one. Therefore, we *shrink* the expansion tree and accordingly update the influence lists of the affecting edges. In our running example, the new result of q contains objects p_3, p_4, p_2 (with $q.kNN\_dist = d(p_2, q)$). Figure 5 shows the updated expansion tree of q.

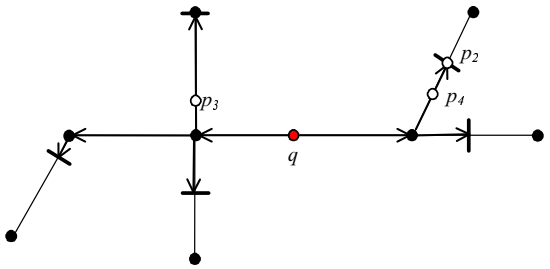


Figure 5: Shrunk expansion tree after object updates

If the outgoing objects are more than the incoming ones, there are fewer than k objects within distance $q.kNN\_dist$ from q. Therefore, we have to expand our search further than $q.kNN\_dist$ to retrieve the new result of q. Consider the example of Figure 4, but assume that p_4 is static. In this case there is an outgoing NN (p_1) but no incoming object. To avoid re-computation from scratch, we utilize the knowledge about the objects that remain within distance $q.kNN\_dist$ (i.e., p_2, p_3) and directly insert them into the new $q.result$. To determine the remaining NNs (the third NN in our example), we start expansion from the marks of $q.tree$. The search proceeds in a way similar to the algorithm in Figure 2; it initializes H to contain the marks of $q.tree$, and considers as verified all the nodes in the current $q.tree$. In our example, it retrieves p_4 as the third NN. Note that $q.tree$ grows according to the new $q.kNN\_dist$ and that we have to update the influence lists of the affecting edges. Figure 6 shows the expansion tree reflecting the new $q.kNN\_dist = d(p_4, q)$.

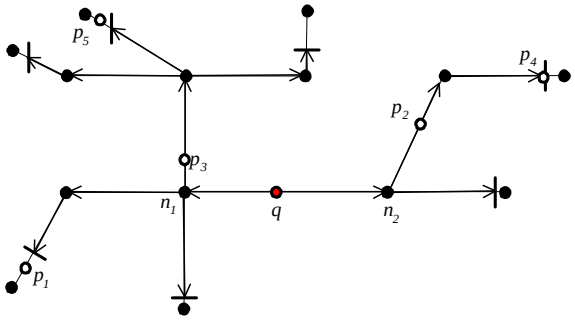


Figure 6: Grown expansion tree after object updates

In the general case, there are multiple queries in the system. At every timestamp, we maintain for each query the number of outgoing NNs and the set of incoming objects. We process object updates one by one. For an update $<p.id, p_{old}, p_{new}>$, let S_{old} be the set of queries affected by p_{old} (i.e., p is one of the previous NNs of each query in S_{old}) and S_{new} be the set of queries affected by p_{new} (i.e., p has moved closer than $q.kNN\_dist$ for every query q in S_{new}). For each q in $S_{new} - S_{old}$, we insert p into the set of incoming objects. For each q in $S_{old} - S_{new}$, we count p as an outgoing NN. For each q in $S_{old} \cap S_{new}$, we update the distance of p in $q.result$. Finally, we compute the queries affected by the updates as described above. Objects that appear in (disappear from) the system are handled as incoming (outgoing) ones.

4.3 Processing of Query Updates

In addition to objects, queries may also move in the network. Consider a query q which moves to a new position q', and assume that there is no object or edge update in the system at that timestamp. A straightforward way to retrieve the new result of q is re-computation from scratch. However, in some cases it is possible to save computations by utilizing the current expansion tree. In particular, if q' falls in some edge of $q.tree$, then the sub-tree of $q.tree$ that is rooted at q' remains valid, as well as the NNs in it (subject to some trivial distance updates).

Continuing the example of Figure 1, assume that q moves to position q', as shown in Figure 7. The new location of the query is contained in the current expansion tree; this can be easily detected, since the influence list of edge n_1n_7 contains q, and q' falls in the corresponding influencing interval. We say that the sub-tree of $q.tree$ that is rooted at q' is *valid*, implying that we know the shortest paths (from q') to the nodes it contains, and we can easily compute their new distance from the query. Similarly, we can directly update the distances of the current NNs that fall in the valid sub-tree (e.g., p_3). On the other hand, we cannot infer the distances of NNs and nodes that fall in the remaining part of $q.tree$ because we do not know the new shortest path to reach them. Consider, for example, node n_5. Due to the query movement, the shortest path to n_5 passes outside $q.tree$, and so does the path to the current second NN p_2. Therefore, we discard the invalid part of $q.tree$ along with the corresponding NNs, and delete q from the influence lists of the edges it spans.

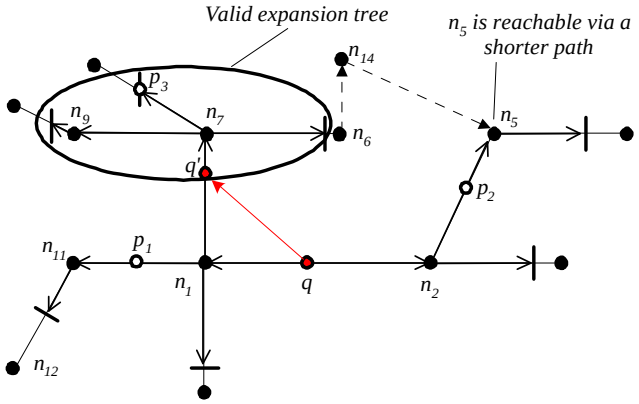


Figure 7: Valid expansion sub-tree when q moves

To retrieve the new result of q, we first insert into $q.result$ the current NNs that fall in the valid expansion sub-tree (with updated distances). Then, we compute the remaining NNs with the algorithm of Section 4.1, by initializing heap H to contain the marks of the valid expansion sub-tree and the location of q' (which is treated as a pseudo-node splitting edge n_1n_7). Finally, we update the query table QT with the new information about q. When q moves outside $q.tree$, we have to perform NN computation from scratch (using the algorithm of Figure 2).

4.4 Processing of Edge Updates

Clearly, only updates of the affecting edges can alter the result of a query q. For these edges, we distinguish two cases, depending on whether their weight increases or decreases. Figure 8 illustrates the first scenario, continuing the example of Figure 1, and assuming that the weight of n_1n_7 changes to a higher value. Consider the nodes/objects in the sub-tree of $q.tree$ that is rooted at n_1. The distance to reach them through n_1n_7 is now larger. This implies that there might exist shorter alternative paths to these nodes/objects. For example, the shortest path to node n_9 could now pass through n_{12} (instead of n_7). Therefore, we remove the invalid sub-tree from $q.tree$, delete q from the corresponding influence lists, and evict the NNs therein from $q.result$. Then, we compute the new NN set with the algorithm of Figure 2, by initializing heap H to contain the marks in the valid part of $q.tree$ and node n_1 (which is treated as a mark on edge n_1n_7).

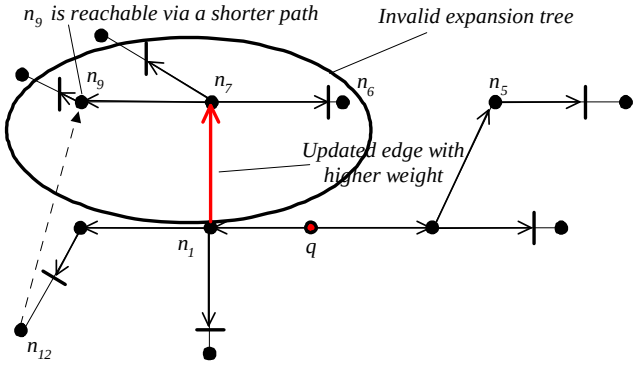


Figure 8: Invalid sub-tree when an edge weight increases

In case that the weight of an affecting edge decreases, update handling prunes $q.tree$ in a different way. Assume that in our running example, edge n_1n_7 receives an update decreasing its weight from 3 to 1, as shown in Figure 9 (the numbers in parentheses indicate the new distances of the nodes in $q.tree$). The sub-tree rooted at n_1 remains valid, since the current path to all nodes/objects therein becomes shorter by 2 units, and clearly remains the optimal path from q. On the other hand, the shortest paths to nodes/objects outside this sub-tree might now pass through the updated edge; e.g., similar to Figure 7, node n_5 might now be reachable through n_1n_7 with a smaller distance than 6. However, the update cannot affect the paths to nodes/objects that lie closer than $d(n_7, q) = 3$, because any path passing through n_1n_7 has length at least $d(n_7, q)$. Therefore, the part of $q.tree$ that remains valid is (i) the sub-tree rooted at the updated edge and (ii) the remaining part of $q.tree$ that has distance less than the furthest endpoint of the edge. In our example, the valid expansion tree contains nodes n_7, n_9 (in (i)) and n_1, n_2 (in (ii)). As shown in Figure 9, part (ii) reaches up to the three new marks, at distance $d(n_7, q) = 3$ from q.

To compute the new result, we update in $q.result$ the distances of the NNs falling in sub-tree (i) above (i.e., p_3). Then, we discard the invalid part of $q.tree$, along with the NNs falling therein (i.e., p_2), and update the corresponding influence lists. Finally, we perform a NN search starting from the marks in sub-tree (i) and the three new marks in part (ii).

A special case occurs when the edge e containing q (e.g.,

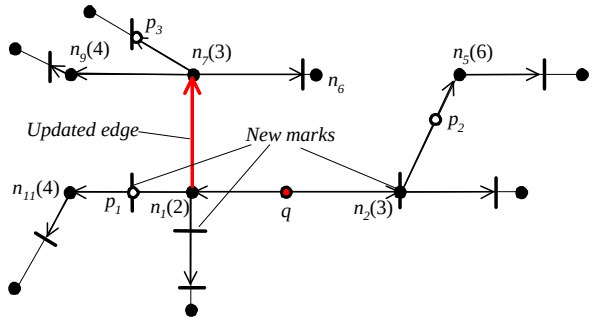


Figure 9: Valid tree when the weight of n_1n_7 decreases

n_1n_2 in Figure 1) receives a weight update. Without loss of generality, assume that $e.start$ (i.e., n_1) is the closest endpoint of e to q. If the update decreases the weight of e, then the distances of the nodes/objects reachable through $e.end$ (i.e., n_2) decrease more with respect to those reachable through $e.start$. This implies that the sub-tree rooted at $e.start$ (i.e., n_1) is invalid and has to be pruned. The remaining part of $q.tree$ remains valid. On the other hand, if the weight of e increases, then the distances of the nodes/objects that are reachable through $e.end$ (i.e., n_2) increase more compared to those reachable through $e.start$ (i.e., n_1). Therefore, we prune the part of $q.tree$ rooted at q towards the side of $e.end$.

4.5 The Complete IMA Algorithm

So far we have considered each type of updates individually. In this subsection we deal with concurrency issues that may arise in the general case, where updates of all three types arrive simultaneously at the system. For ease of presentation, we discuss monitoring of a single query before considering multiple ones. Assume that there is a CkNN query q running at the server, and that we receive, in the same timestamp, updates for the query location, for weight changes of edges affecting q, and for object movements within distance $q.kNN\_dist$ from q. Our aim is to process the updates in an order that guarantees correctness and saves as many computations as possible. In particular, we target at maximizing the size of the remaining (valid) part of the expansion tree of q.

IMA first checks whether q moves out of $q.tree$. If this is the case, re-computation from scratch is necessary, and the monitoring algorithm ignores all other updates (to save extra processing time on an already invalid expansion tree). If q moves inside $q.tree$, IMA ignores the query movement at this point, and proceeds with the weight changes of affecting edges. Note that pruning $q.tree$ according to the query update before considering edge updates may lead to an invalid tree. Consider the example of Figure 7 and assume that besides the query movement, the weights of edges n_1n_{11} and $n_{11}n_{12}$ decrease at the same timestamp. If we processed the query update first, then we would prune $q.tree$ as shown in the figure, and we would ignore the changes of edge weights. The resulting tree, however, is not necessarily valid. For instance, node n_9 could now be reachable faster through edges n_1n_{11} and $n_{11}n_{12}$ (than via node n_7). Therefore, we have to trim $q.tree$ based on the edge updates before handling the query movement.

IMA prunes $q.tree$ according to the edge changes, in the way discussed in Section 4.4. Among the updates in affecting edges, it is important to process decreasing weights before increasing ones in order to ensure correctness. Consider the example of Figure 8, and assume that the weight of edge n_1n_7 increases by 0.5 units, while the weight of n_7n_6 decreases by 2. If we processed the former update first, then we would simply discard the sub-tree rooted at n_1n_7 and we would keep the remaining $q.tree$. However, this part is not necessarily valid! The reason is that some nodes in it (e.g., n_5) might be reachable through a shorter path that passes from the updated edge n_7n_6 (in total, the distance of n_6 is now smaller). We avoid this problem by processing the decreasing weight (for n_7n_6) before the increasing one (for n_1n_7).

When finished with edge updates, IMA considers the query movement and further prunes $q.tree$ with the technique of Section 4.3. Next, it considers object updates that affect the remaining (valid) part of $q.tree$. Incoming objects are inserted into $q.result$, outgoing NNs are deleted, and the distances of NNs moving within the tree are updated in $q.result$. Note that after this step, if there are many incomers, the cardinality of $q.result$ may exceed k. In this case, IMA keeps only the k best objects, and sets $q.kNN\_dist$ to the distance of the kth (furthest) one. If there are fewer than k objects in $q.result$, then $q.kNN\_dist$ is equal to infinity.

After processing all updates, if every mark in the valid expansion tree has distance higher than $q.kNN\_dist$, then $q.result$ is the actual result. Otherwise, there may be objects outside $q.tree$ that lie closer to q. To retrieve these objects, IMA uses the algorithm of Figure 2 with search heap H initialized to contain the marks. Note that after the NN search, some parts of $q.tree$ might have distance larger than $q.kNN\_dist$. IMA deletes these parts and accordingly updates the influence lists of the corresponding edges. For example, consider Figure 9, and assume that (besides the edge update) there are three objects moving in edge n_1n_2, leading to $q.kNN\_dist = 2.5$. Clearly, the expansion tree has to shrink to reflect the new $q.kNN\_dist$.

Figure 10 illustrates the complete IMA algorithm that processes multiple queries simultaneously. The sets U_{obj}, U_{qry} and U_{edg} contain the updates received in the current timestamp for data objects, queries and edges, respectively. Lines 1-3 exclude from update handling the queries in U_{qry} that move outside their current expansion tree. Lines 4-10 and 11-13 process decreasing and increasing weights in U_{edg}, respectively, and prune the expansion trees of affected queries. Lines 14-15 handle queries q in U_{qry} that move within $q.tree$, invalidating the appropriate part of the tree. Lines 16-19 consider object movements, and treat them as incoming, outgoing, or moving NNs for the influenced queries. Finally, lines 20-26 re-compute the result of each query q affected by U_{obj}, U_{qry} and U_{edg}, utilizing the remaining part of $q.tree$ (if any). For the sake of readability, in Figure 10 we ignore the case where some user terminates an existing query, or installs a new one. In the former case we simply remove q from QT and delete it from the influence lists of the affecting edges. We perform these tasks before processing any update, in order to avoid redundant computations for terminated queries. On the other hand, we insert the newly installed queries into QT and compute their initial results after all updates take place in the system (i.e., after line 19 in Figure 10).

```
Incremental Monitoring Algorithm (U_obj,U_qry,U_edg)
1.  For each query q in U_qry that moves outside q.tree
2.    Delete q from the influence lists of affecting edges
3.    Discard q.tree and q.result //re-computation is necessary
4.  For each edge e in U_edg with decreasing weight
5.    For each query q in e.IL
6.      Delete the invalid part of q.tree
7.      Remove q from the influence lists in the invalid part
8.      Remove from q.result the NNs in the invalid part
9.      Update distances of nodes in the valid part of q.tree
10.     Update in q.result the distances of NNs in the valid q.tree
11. For each edge e in U_edg with increasing weight
12.   For each query q in e.IL
13.     (same as lines 6-8)
14. For each query q in U_qry that moves inside q.tree
15.   (same as lines 6-10)
16. For each object p in U_obj
17.   Delete p from its old edge; Insert p into its new edge
18.   For each query affected by p
19.     Treat p as outgoing, incoming, or moving NN
20. For each query q affected in lines 1, 5, 12, 14, 18
21.   q.result = k best objects among remaining NNs and incomers
22.   Insert into min-heap H the marks of the valid q.tree
23.   If head of H has key < distance of the kth object in q.result
24.     Consider all nodes in the valid q.tree as verified
25.     Perform NN search initializing the heap to H
26.   If necessary, shrink q.tree; Update influence lists accordingly
```

Figure 10: The IMA algorithm

The above mechanism handles concurrency issues and generates the correct results provided that the objects, the queries, and the edges issue at most one update per timestamp. If this assumption does not hold, we have to perform a preprocessing step before running the IMA algorithm. In particular, if an object p issues multiple location updates, then we replace them in U_{obj} with a single one $<p.id, p_{old}, p_{new}>$, where p_{old} is the old location in the first update received and p_{new} is the final location in the last update. Preprocessing is similar for moving queries. Finally, when multiple weight updates arrive for an edge e, they are aggregated into a single one indicating the overall weight change with respect to the previous timestamp.

5. GROUP MONITORING ALGORITHM

The group monitoring algorithm (GMA) integrates IMA with the shared execution paradigm. The concept of *sequence* is central to this method, and it is extensively used in the following discussion. A sequence is a path between two nodes n_i and n_j, such that (i) the degrees of n_i and n_j are not equal to 2 and (ii) all intermediates nodes in the path have degree 2. This means that n_i and n_j are either *intersection* nodes (with degree above 2) or *terminal* nodes (with degree 1). Note that each edge in a network belongs to exactly one sequence, i.e., every graph is partitioned in a set of sequences that cover all nodes and whose edges do not overlap. The network of Figure 11 contains seven sequences $\{n_1n_8\}$, $\{n_1n_9\}$, $\{n_1n_7, n_7n_6, n_6n_5\}$, $\{n_1n_2\}$, $\{n_2n_3\}$, $\{n_2n_5\}$ and $\{n_5n_4\}$. GMA is based on the following lemma.

LEMMA 1. *The k-NN set of any query q falling in a sequence s is contained in the union of (i) the objects in s, (ii) the k-NN sets of the intersection nodes (endpoints) of s.*

Assume that q_1 and q_2 in Figure 11 are 2-NN queries. Lemma 1 implies that the two NNs of any query in the sequence between n_1 and n_5 (i.e., q_1 and q_2) belong to the union of the objects in the sequence (i.e., $\{p_4, p_5\}$) and the 2-NNs of its endpoints n_1 (i.e., $\{p_1, p_5\}$) and n_5 (i.e., $\{p_3, p_2\}$). GMA monitors the two NNs of n_1 and n_5 in order to efficiently report the answers of q_1 and q_2, while they remain within the sequence.

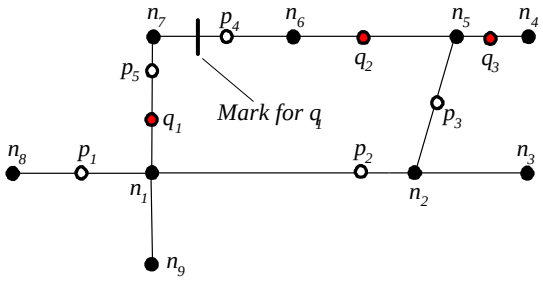


Figure 11: Example of sequences and GMA functionality

In general, GMA groups together the queries falling in the same sequence and monitors static nodes (at the endpoints of the sequence), instead of each query individually. Consider an intersection node n in the network. Let $n.S$ be the set of sequences containing n as one of their endpoints, and $n.Q$ be the set of all queries falling in some sequence of $n.S$. We say that n is an *active* node if $n.Q$ is non-empty. Returning to the example of Figure 11, intersection nodes n_1 and n_5 are active ($n_1.Q = \{q_1, q_2\}$ and $n_5.Q = \{q_1, q_2, q_3\}$), while n_2 is inactive because $n_2.Q = \emptyset$. For each active node n in the system we monitor the $n.k$ nearest objects, where $n.k = \max_{q \in n.Q} q.k$ (i.e., the maximum number of NNs required by any query in $n.Q$). For n_1 we monitor two NNs. Assuming that q_3 is a 3-NN query, for n_5 we monitor three NNs. Note that only intersection nodes can be active. In sequence $\{n_5n_4\}$, for instance, terminal node n_4 is inactive; the NNs of q_3 belong to the union of the objects in n_5n_4 and the 3-NN set of n_5.

Monitoring the NNs of active nodes is performed with IMA, as shown in Figure 10, except that lines 1-3 and 14-15 are never executed since the active nodes are static. We use an active node table NT to store information about the NNs and expansion trees of these nodes. The structure of NT is similar to that of QT in IMA. The influence lists of the edges in GMA contain the ids of the active nodes that they affect, along with the corresponding influencing intervals. In GMA we use one additional data structure, the sequence table ST, which stores, for each sequence, the set of its edges. Also, the edge table ET keeps, for each edge, the id of the sequence it belongs to. ST and ET implicitly maintain $n.S$ and $n.Q$ for every active node n.

Concerning the actual queries of the users, GMA uses different NN search and maintenance algorithms than IMA. To illustrate the initial result computation in GMA, consider the 2-NN query q_1 in Figure 11. Assume that q_1 is the first query installed in the system. Since q_1 falls in the sequence from n_1 to n_5, we set nodes n_1 and n_5 as active and compute their two NNs. To evaluate q_1, a straightforward application of Lemma 1 would be to merge the retrieved 2-NN sets of n_1 and n_5 with all the objects in edges n_1n_7, n_7n_6, and n_6n_5. This method, however, can be very expensive, because a sequence may contain numerous edges and objects. Instead, we perform NN search by expand-

ing around q_1, utilizing if necessary the NNs of n_1 and n_5. First, we consider edge n_1n_7, and insert p_5 into $q_1.result$. Among the two reached nodes (n_1 and n_7), n_1 is the closest one. Node n_1 is active, and hence we update $q_1.result$ with its NNs, p_1 and p_5. Since p_1 is already in $q.result$, we do not re-insert it. Subsequent search continues only towards n_5. The next node is n_7, whose distance $d(n_7, q_1)$ is smaller than $q_1.kNN\_dist = d(p_1, q_1)$. Therefore, we consider edge n_7n_6 and process the objects in it (i.e., p_4). The algorithm terminates at this point with $q_1.result = \{p_5, p_1\}$, since the next node n_6 has $d(n_6, q_1) > q_1.kNN\_dist$. As opposed to IMA, GMA does not compute (or store) expansion trees for queries during NN retrieval.

Regarding result maintenance in GMA, let q be a query that lies in sequence s. There are four events that may alter $q.result$: (i) movement of q, (ii) changes in the NN sets of active nodes in s, (iii) object updates in s, and (iv) edge updates in s. A straightforward maintenance algorithm would re-compute q in any of these events. However, even though (i) requires NN search from scratch, the other three events do not always affect q. To illustrate this, consider q_1 in the previous example. A NN change for active node n_5 cannot alter $q_1.result$, because it lies further away than $q_1.kNN\_dist$. Similarly, an edge or object update further than $q_1.kNN\_dist$ does not affect q_1.

In order to detect the events among (ii), (iii), and (iv) that actually invalidate the result of q, we maintain influence list information in the edges of s that affect the query. This information is stored during the initial result computation. In our example, the NN search for q_1 considers edges n_1n_7 and n_7n_6. These edges receive q_1 in their influence lists. The influencing interval of n_1n_7 is the entire edge, while that of n_7n_6 extends from n_7 up to the mark with distance $q_1.kNN\_dist$ from q_1. Note that a query q can be included only in influence lists of edges within the sequence s containing it, since the NN search for q does not consider edges outside s. Returning to our example, assume that the NN set of n_1 changes (event (ii)). GMA infers that q_1 may be affected, since n_1 falls in the influencing interval of n_1n_7. On the other hand, an object movement (event (iii)) can invalidate $q.result$ if its old or new location falls in edge n_1n_7 or in the influencing interval of n_7n_6. Similarly, weight changes in edges that contain q_1 in their influence list (i.e., n_1n_7 and n_7n_6) may alter its NN set. If q_1 is affected by any of the above events, GMA re-computes it from scratch.

Before presenting the complete monitoring algorithm, we clarify that in GMA the influence list of an edge e contains (i) the active nodes affected by e, and (ii) the queries in its sequence influenced by e. The information in (i) is used by the IMA module that monitors the results of active nodes, while (ii) facilitates update handling for the queries within each sequence. Figure 12 illustrates GMA for the general case of multiple, concurrent queries.

Lines 1-5 monitor the NNs of the active nodes. A query movement is treated as a deletion (of the old query) and an insertion (at its new location). GMA scans U_{qry} to form the sets of inserted and deleted queries, Q_{ins} and Q_{del} respectively. For each query in Q_{ins} and Q_{del} it (i) updates $n.k$ for the active nodes, (ii) inserts new nodes into NT, and (iii) sets nodes with $n.Q = \emptyset$ as inactive. Then, it invokes IMA to update the results of active nodes. Lines 6-15 determine the actual user queries that are affected by the updates, and place them into set *affected_queries*. First (lines 7-8), for each active node n whose NNs changed in line 5, GMA utilizes the influence lists of the edges adjacent to n to determine the affected queries. For each such edge e, it scans the queries in $e.IL$ and includes in *affected_queries* the ones whose corresponding influencing interval includes n. Lines 9-12 consider object updates. Assume an object movement from p_{old} to p_{new}. GMA uses the influence list of the edge containing p_{old} to determine which queries are affected by the update, and inserts them into *affected_queries*. Processing is similar for p_{new}. Next, it processes U_{edg}. For each edge e in U_{edg}, GMA includes in *affected_queries* all the queries q in $e.IL$ (lines 13-15). Finally, it computes the NNs of each query in *affected_queries* $\cup\ Q_{ins}$ in the way described earlier.

```
Group Monitoring Algorithm (U_obj,U_qry,U_edg)
1.  Scan U_qry and form sets Q_ins and Q_del
2.  For each query q in Q_ins (Q_del)
3.     Insert (delete) q in the corresponding edge
4.     Update n.k for endpoints (active nodes) of containing sequence
5.  Invoke IMA for active nodes, taking into account changes in n.k
6.  Set affected_queries = ∅
7.  For each active node n whose result changed in line 5
8.     Insert into affected_queries each query in n.Q affected by n
9.  For each update <p.id, p_old, p_new> in U_obj
10.    Let s_old (s_new) be the sequence containing p_old (p_new)
11.    For each query q in s_old (s_new) that is affected by p_old (p_new)
12.       Insert q into affected_queries
13. For each edge e in U_edg
14.    Let s be the sequence containing e
15.    Insert in affected_queries each query in s that is affected by e
16. For each query q in affected_queries or Q_ins
17.    Compute q.result from scratch (utilizing active node NN sets)
```

Figure 12: The GMA algorithm

Concurrency control is achieved by scheduling the updates in a way similar to IMA. In terms of running time, GMA is expected to outperform IMA when (i) the number of queries is large with respect to the number of network nodes, and (ii) when the queries are concentrated in a small part of the network. In terms of memory requirements, GMA uses an extra structure (the sequence table), but IMA stores the expansion trees of all the queries in the system.

6. EXPERIMENTAL EVALUATION

This section evaluates IMA and GMA using sub-networks of the San Francisco road map [2]. In the default setting, we use a sub-network with 10K edges containing N objects and Q queries, where N and Q are system parameters. The initial positions of objects and queries follow either uniform or Gaussian distribution (with mean at the center of the workspace and standard deviation 10% of the maximum network distance from the center). The initial weights of the edges correspond to their lengths (i.e., the Euclidean distance between their endpoints). At every timestamp, a percentage f_{edg} of the edges receive a weight update, where f_{edg} is called the *edge agility*. An update increases or decreases the weight by 10% over its previous value. Similarly, f_{obj} is the *object agility*, i.e., the percentage of objects that move per timestamp, and f_{qry} is the *query agility*. A moving object (query) performs a random walk in the network and covers a fixed distance v_{obj} (v_{qry}), where v_{obj} (v_{qry}) is the object (query) speed. Updates of all three types occur at each timestamp.

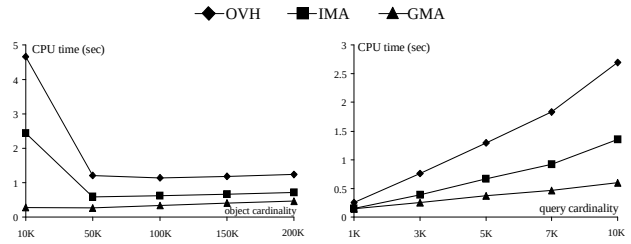
Queries require continuous monitoring of their k NNs for 100 timestamps. As a benchmark against IMA and GMA,

we use an overhaul method (OVH) that computes each query from scratch at every timestamp, using the algorithm of Figure 2. Table 2 includes the parameters under investigation. In each experiment we vary a single parameter (in the range shown), and set the remaining ones to their default values. Unless otherwise specified, the diagrams illustrate the processing time per timestamp (in seconds). For all simulations, we use a Pentium 2.3 GHz CPU with 1 GByte memory.

Parameter	Default	Range
Number of objects (N)	100K	10, 50, 100, 150, 200 (K)
Number of queries (Q)	5K	1, 3, 5, 7, 10 (K)
Object distribution	Uniform	Gaussian, Uniform
Query distribution	Gaussian	Gaussian, Uniform
Number of NNs (k)	50	1, 25, 50, 100, 200
Edge agility (f_{edg})	4%	1, 2, 4, 8, 16 (%)
Object speed (v_{obj})	1 edge/ts	0.25, 0.5, 1, 2, 4
Object agility (f_{obj})	10%	0, 5, 10, 15, 20 (%)
Query speed (v_{qry})	1 edge/ts	0.25, 0.5, 1, 2, 4
Query agility (f_{qry})	10%	0, 5, 10, 15, 20 (%)

Table 2: System parameters

Figure 13(a) measures the effect of the object cardinality N (ranging from 10K to 200K) on the running time. GMA outperforms both IMA and OVH, with IMA being the runner up in all cases. GMA monitors only 844 active nodes on the average, which explains its wide difference from IMA, and indicates that there are long sequences including many edges and queries. For $N = 10$K the network is sparse (one object per edge) and, thus, the NN search considers many edges for all algorithms. On the other hand, when the data are dense (e.g., for $N = 200$K), NN computation processes fewer edges which, however, contain many objects. This is the reason that the cost initially decreases and then slightly increases for $N > 50$K. The important observation is that all algorithms scale very well with the object cardinality.

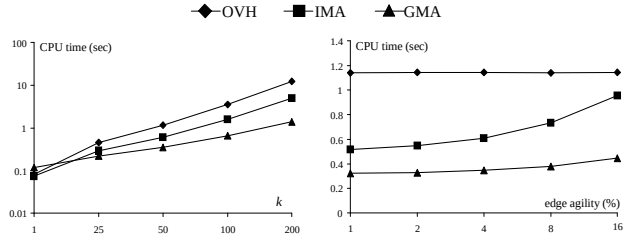


(a) Effect of N (b) Effect of Q

Figure 13: CPU time versus object and query cardinality

Figure 13(b), shows the running time versus the number of queries Q. For $Q = 1$K, GMA is slightly faster than IMA, and twice as fast as OVH. The difference grows for larger Q; for $Q = 10$K, GMA is more than two times faster than IMA, and 4.5 times faster than OVH. GMA scales better than IMA with the number of queries, because it benefits from shared execution. For $Q = 10$K there are only 79% more active nodes than for $Q = 1$K (on the average, there are 533 and 956 active nodes, respectively).

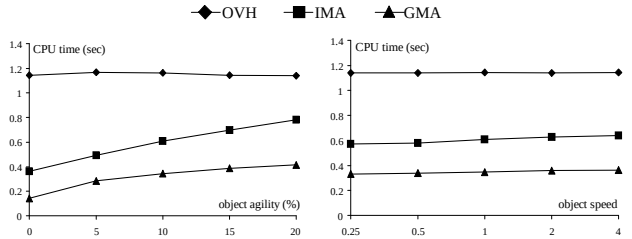
Figure 14(a) plots the CPU time (in logarithmic scale) versus the number k of NNs required, using the default settings for the other parameters. GMA is again the best algorithm, except for $k = 1$ where IMA is more efficient. This happens because, for $k = 1$, the NN of most queries is very close to them, and usually closer than any active node. Thus, monitoring active nodes incurs unnecessary overhead. However, as k increases, the expansion trees in IMA grow larger, along with the maintenance cost. GMA on the other hand, benefits from large k, because the results of active nodes are utilized to a higher degree (i.e., by more queries).



(a) Effect of k (b) Effect of f_{edg}

Figure 14: CPU time versus k and edge agility

Figure 14(b) illustrates the effect of the edge agility f_{edg}. The costs of both IMA and GMA increase for higher f_{edg}, since frequent weight updates invalidate more expansion trees and affect more queries. However, GMA is not very sensitive to edge agility; its running time for $f_{edg} = 16\%$ is only 37% higher than for $f_{edg} = 1\%$. In Figure 15(a) we vary the object agility f_{obj} from 0% (static data) to 20%. The cost of both IMA and GMA increases with object agility because frequent object updates imply many result invalidations. GMA is more robust to f_{obj} than IMA.



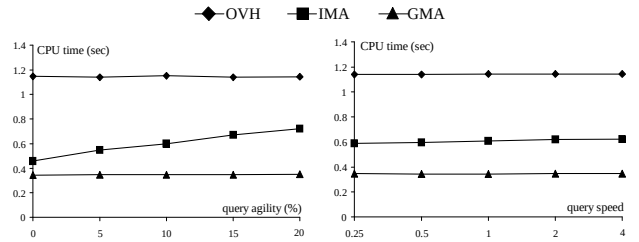
(a) Effect of f_{obj} (b) Effect of v_{obj}

Figure 15: CPU time versus object agility and speed

Figure 15(b) studies the effect of object speed v_{obj} (i.e., the distance covered by the moving objects), which ranges from 0.25 up to 4 times the average edge length. The performance of our algorithms is practically unaffected by v_{obj} because an object update is treated as a deletion (at its old location) and an insertion (at its new location). The expected number of affected queries is independent of how far (i.e., fast) the object moves.

Figure 16(a) plots the CPU time as a function of the query agility f_{qry}. The performance of IMA degrades with f_{qry} because a query movement invalidates (part of) its expansion tree. On the other hand, GMA is very robust to f_{qry} since a moving query is always monitored using the active nodes of the sequence that contains it (even if this sequence changes over time). Figure 16(b) varies the query speed v_{qry} between 0.25 and 4 times the average edge length per timestamp. As

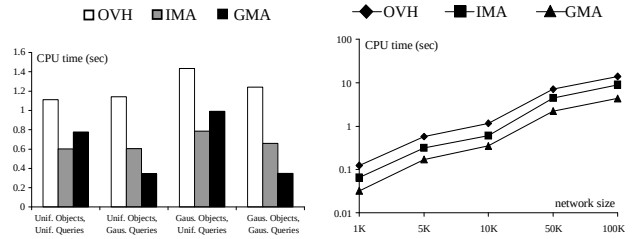
expected, GMA has almost constant cost for all values of v_{qry}. The running time of IMA slightly increases with v_{qry} because as the query moves faster, the part of its expansion tree that remains valid decreases.



(a) Effect of f_{qry} (b) Effect of v_{qry}

Figure 16: CPU time versus query agility and speed

In all previous experiments, the initial positions of the objects distribute uniformly in the network, while the queries follow a Gaussian distribution. In Figure 17(a), we measure the running time for different combinations of object and query distributions, using the default values for the remaining parameters. The Gaussian objects have standard deviation 50%. GMA is the best method for Gaussian queries because they are clustered in a small part of the network and permit effective information sharing using relatively few active nodes. On the other hand, IMA is the winner for uniform queries because they are sparsely distributed and each sequence in the network contains only few of them. Both GMA and IMA outperform OVH in all cases.

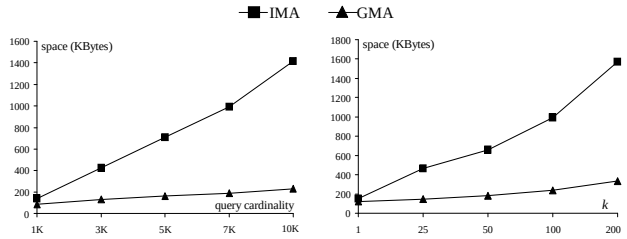


(a) Effect of distributions (b) Effect of network size

Figure 17: CPU time versus distribution and network size

In order to measure the effect of the network size, in Figure 17(b), we use sub-networks of the San Francisco road map containing from 1K up to 100K edges. The number of objects and queries is proportional to the number of edges so that each edge contains, on the average, 10 objects and 0.5 queries. For 10K edges (N = 100K, Q = 5K) the cost of our methods is 0.3-0.6 seconds per timestamp (for GMA and IMA, respectively), implying that they can be used when the interval between updates is no shorter than these values. Similarly, for 100K edges (N = 1M, Q = 50K), they can handle update intervals of less than 10 seconds.

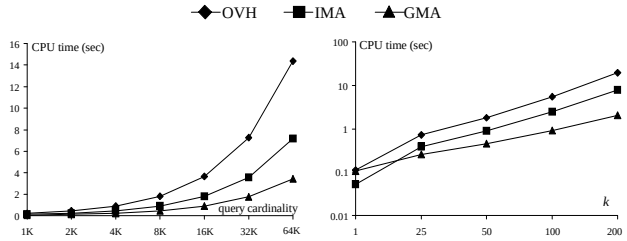
In Figure 18 we illustrate the memory overhead of GMA and IMA for the experiments of Figures 13(b) and 14(a). IMA consumes more space than GMA, mainly because it stores the expansion trees of all the queries in the system. The difference increases with the query cardinality and k, since in the first case there are more expansion trees to store, and in the second one the trees are larger. On the other hand, GMA scales gracefully, since it maintains expansion trees only for the active nodes. In Figure 18(a), the number of active nodes is small even for large query cardinality, as discussed in the context of Figure 13(b). In Figure 18(b), the number of active nodes is constant, but their expansion trees are larger for higher k.



(a) Effect of Q (b) Effect of k

Figure 18: Memory requirements versus Q and k

All the experiments presented so far used a simple generator. Even though there are more sophisticated ones, such as that of [2], they do not provide control over most of the parameters under investigation. For generality, however, we include two experiments for objects and queries generated with the simulator of [2] using its default parameters. We also use a different road network, the road map of Oldenburg, containing 6105 nodes and 7035 edges. In Figure 19(a), we generate N = 64K objects and vary the number of queries Q between 1K and 64K. Similar to Figure 13(b), the difference of GMA from IMA and OVH is larger for higher Q, due to the benefits from query grouping and shared execution. In Figure 19(b), we generate N = 64K objects and Q = 8K queries, and measure the effect of k on the performance of the algorithms. GMA outperforms both IMA and OVH, except for $k = 1$ where IMA is better. The reason for this behavior is the same as in Figure 14(a).



(a) Effect of Q (b) Effect of k

Figure 19: Experiments with the generator of [2]

7. CONCLUSIONS

This paper constitutes the first work addressing continuous k-NN monitoring in road networks. We propose two methods, the incremental monitoring algorithm (IMA) and the group monitoring algorithm (GMA). IMA monitors each query individually, processing only updates that might affect its result, and ignoring the rest. GMA groups together

the queries falling in the path between two consecutive intersections in the network. It monitors the k-NNs of the intersections, and utilizes them in order to facilitate evaluation of the queries in the path. IMA and GMA do not require any knowledge about the moving patterns of objects and queries, and they can handle weight changes in the edges of the network. We demonstrate their efficiency through realistic experiments. GMA is, in general, better than IMA in terms of space requirements and CPU cost.

Our algorithms aim at minimizing the CPU cost at the central processing server. Other techniques (e.g., [10, 17]) focus on reducing the communication overhead caused by frequent location updates; they assume that objects have some computational capabilities and knowledge of the queries so that they can issue location updates only when they influence some query result. An interesting direction for future work is to combine these approaches with ours and design a comprehensive system that minimizes both the CPU and the communication cost.

Another challenging research direction is monitoring of different queries in road networks, such as continuous reverse nearest neighbor ones (CRNN). Consider a set of queries and a set of data objects moving in a network. Our task is to constantly report for each query q the set of objects that are closer to q than to any other query. As an example, consider a taxi driver who wishes to know the clients (pedestrians asking for taxi) that are closer to his/her position than to any other vacant cab. In this case, the taxis correspond to the queries, and the clients to the data objects.

8. ACKNOWLEDGMENTS

This work was supported by grants HKUST 6178/04E, HKUST 6184/05E, and HKU 7160/05E from Hong Kong RGC. The authors would like to thank Spiridon Bakiras for his valuable comments.

9. REFERENCES

[1] R. Benetis, C. S. Jensen, G. Karciauskas, and S. Saltenis. Nearest and reverse nearest neighbor queries for moving objects. *VLDB Journal*, 15(3):229–250, 2006.

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

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

[4] H.-J. Cho and C.-W. Chung. An efficient and scalable approach to CNN queries in a road network. In *VLDB*, 2005.

[5] E. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[6] B. Gedik and L. Liu. MobiEyes: Distributed processing of continuously moving queries on moving objects in a mobile system. In *EDBT*, 2004.

[7] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, 1984.

[8] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM TODS*, 24(2):265–318, 1999.

[9] E. G. Hoel and H. Samet. Efficient processing of spatial queries in line segment databases. In *SSD*, 1991.

[10] H. Hu, J. Xu, and D. L. Lee. A generic framework for monitoring continuous spatial queries over moving objects. In *SIGMOD*, 2005.

[11] C. S. Jensen, J. Kolář, T. B. Pedersen, and I. Timko. Nearest neighbor queries in road networks. In *GIS*, 2003.

[12] M. R. Kolahdouzan and C. Shahabi. Continuous K-nearest neighbor queries in spatial network databases. In *STDBM*, 2004.

[13] M. R. Kolahdouzan and C. Shahabi. Voronoi-based K nearest neighbor search for spatial network databases. In *VLDB*, 2004.

[14] N. Koudas, B. C. Ooi, K.-L. Tan, and R. Zhang. Approximate NN queries on streams with guaranteed error/performance bounds. In *VLDB*, 2004.

[15] M. F. Mokbel, X. Xiong, and W. G. Aref. SINA: Scalable incremental processing of continuous queries in spatio-temporal databases. In *SIGMOD*, 2004.

[16] K. Mouratidis, M. Hadjieleftheriou, and D. Papadias. Conceptual partitioning: An efficient method for continuous nearest neighbor monitoring. In *SIGMOD*, 2005.

[17] K. Mouratidis, D. Papadias, S. Bakiras, and Y. Tao. A threshold-based algorithm for continuous monitoring of k nearest neighbors. *IEEE TKDE*, 17(11):1451–1464, 2005.

[18] D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao. Query processing in spatial network databases. In *VLDB*, 2003.

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

[20] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *SIGMOD*, 1995.

[21] C. Shahabi, M. R. Kolahdouzan, and M. Sharifzadeh. A road network embedding technique for k-nearest neighbor search in moving object databases. In *GIS*, 2002.

[22] S. Shekhar and J. S. Yoo. Processing in-route nearest neighbor queries: a comparison of alternative approaches. In *GIS*, 2003.

[23] Z. Song and N. Roussopoulos. K-nearest neighbor search for moving query point. In *SSTD*, 2001.

[24] Y. Tao and D. Papadias. Spatial queries in dynamic environments. *ACM TODS*, 28(2):101–139, 2003.

[25] X. Xiong, M. F. Mokbel, and W. G. Aref. SEA-CNN: Scalable processing of continuous k-nearest neighbor queries in spatio-temporal databases. In *ICDE*, 2005.

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

[27] J. Zhang, M. Zhu, D. Papadias, Y. Tao, and D. L. Lee. Location-based spatial queries. In *SIGMOD*, 2003.

[28] B. Zheng and D. L. Lee. Semantic caching in location-dependent query processing. In *SSTD*, 2001.