Location-based Query Processing on Moving Objects in Road Networks*

Haojun Wang
Computer Science Department
University of Southern California
Los Angeles, CA, USA 90089
haojunwa@usc.edu

Roger Zimmermann
School of Computing
National University of Singapore
Singapore 117543
rogerz@comp.nus.edu.sg

ABSTRACT

Location-based services are increasingly popular. In this paper, we present our novel design of a system to process location-based queries on MOVing objects in road Networks (MOVNet, for short) that efficiently handles a large number of spatial queries over moving objects in a stationary road network. MOVNet utilizes an on-disk R-tree to store the network connectivities and an in-memory grid index to maintain moving object position updates. In addition, a technique to speedily compute the overlapping grid cells in the network is proposed to relate these two indices. Moreover, given an arbitrary edge in the space, we analyze the minimum and maximum number of grid cells that are possibly affected. Finally, based on these features, we propose algorithms for mobile network distance range queries and nearest neighbor queries to support mobile location-based services. We demonstrate via theoretical analysis and experimental results that MOVNet yields excellent performance in various networks and with a very large number of moving objects.

1. INTRODUCTION

With the widespread use of GPS devices, more and more people are enjoying location-based services. Various applications, such as road-side assistance, highway patrol, and location-aware advertisement, are popular in many – especially urban – areas. This has intensified research interests to overcome the inherent challenges in designing scalable and efficient infrastructures to support very large numbers of users concurrently. The mobility that is made possible by the usage of car-based or handheld GPS devices in metro cities results in two fundamental system requirements: distance computations based on a (road) network and processing of moving Points of Interest (POIs).

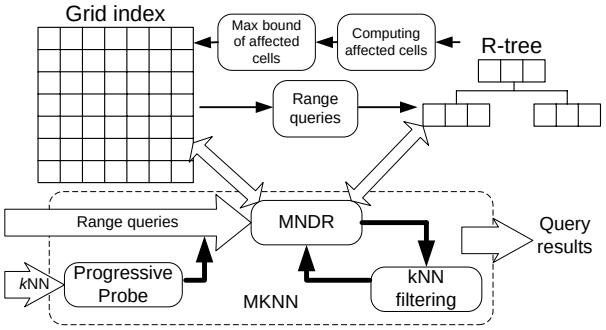


Figure 1: The index structures and query processing modules of MOVNet

There exist an increasing number of applications that require query processing of moving POIs based on an underlying network. For example, a visitor may desire to find the three nearest available taxis on the road and call for service. When a pedestrian calls for emergency assistance the call-center may want to locate all police cars within a five-mile distance and dispatch them to the call-originating location.

Spatial data processing is a very active research field. Some of the early work introduced spatial processing of stationary objects based on Euclidean distance metrics. More recent work incorporates POI mobility or network-distance processing, but often not both. Several techniques [3, 15, 21] have aimed at solving location-based (Continuous) k Nearest Neighbor (kNN/C-kNN) queries in spatial networks. These methods assume that the positions of POIs are fixed (e.g., gas stations or bus stops). While the ability to process moving POIs is challenging, it also enables new applications and in the most general case the query points and the POIs are interchangeable, *i.e.*, users or vehicles equipped with GPS devices are able to report their positions and hence themselves become POIs. Some prior work has focused on providing the functionality for moving POI processing [19, 26, 28]. Specifically, these techniques aim to continuously monitor a set of moving nearest neighbors. However, one of the limitations of these methods is their reliance on Euclidean distance measures, which can be imprecise, especially in dense road networks.

The main challenges when supporting POI mobility on an underlying road network is to (a) efficiently manage object location updates and (b) provide fast network-distance

*This research has been funded in part by NSF grants EEC-9529152 (IMSC ERC), CMS-0219463 (ITR), IIS-0534761, NUS AcRF grant WBS R-252-050-280-101/133 and equipment gifts from the Intel Corporation, Hewlett-Packard, Sun Microsystems and Raptor Networks Technology.

VLDB '07, September 23-28, 2007, Vienna, Austria.

computations. IMA/GMA is among the few existing solutions applicable to this environment, providing a method for C-kNN monitoring [20]. We present our novel design of a system to process location-based queries on MOVing objects in road Networks (MOVNet). MOVNet focuses on executing snapshot range and nearest neighbor queries over moving POIs in a stationary road network. Figure 1 illustrates the system infrastructure and components of MOVNet. Specifically, MOVNet combines an on-disk R-tree [8] structure to store the connectivity information of the road network with an in-memory grid index to efficiently process moving object position updates. A feature of MOVNet is the bi-directional mapping between the two structures that enables the retrieval of a minimal set of data for query processing. We introduce the concept of *affected cells* that form the set of grid cells overlapping with a given edge and provide an efficient algorithm to compute these cells. Furthermore, we propose algorithms to execute range as well as kNN queries with the MOVNet infrastructure. We provide analytical bounds on the minimum and maximum number of affected cells with an arbitrary edge in the network, which enables the pruning of the search space during the mobile range query processing. In the mobile kNN query algorithm, we utilize the concept of a *progressive probe* into the grid index to estimate the subspace containing the result set.

The performance of the MOVNet design has been verified by vigorous simulations and theoretical analysis and it features the following contributions:

- To the best of our knowledge, MOVNet is the first dual-index design that utilizes an R-tree to store network connectivity and a grid index to support location updates from moving POIs. We demonstrate that these two indices can be seamlessly correlated. The overhead during query processing is less than 5% of the CPU time in various tests while the method offers substantial flexibility and scalability.

- We propose mobile range query and nearest neighbor query algorithms in network distances to illustrate the functionality and performance of MOVNet. As comparisons, we devised two baseline algorithms. The simulation results consistently show that our algorithms outperform the baseline algorithms with various network densities.

- We demonstrate that the analytical properties of the minimum and maximum number of affected cells with an arbitrary edge can be effectively used to enhance the performance of the range query algorithm. Especially with long road segments (e.g., freeways), the CPU execution time is reduced by up to 50%. Additionally, our experimental study shows that up to 48% of the kNN query results are enclosed in the space resulted from the progressive probe. This significantly reduces the complexity of the query processing.

The remainder of this paper is organized as follows. Section 2 describes the related work. Section 3 discusses our assumptions and the dual-index design. In the following Section 4 we propose our mobile network distance range query and k nearest neighbor query algorithms. We present the theoretical analysis of our design in Section 5. We vigorously verify the performance of MOVNet and demonstrate that the results match with our analysis in Section 6. Finally we conclude with Section 7.

2. RELATED WORK

Processing spatial queries in the network has been intensively studied recently. Papadias et al. [21] first presented an architecture that integrates network and Euclidean information in processing network-based queries. Specifically, the idea of *Euclidean restriction* utilizes the property that for any two objects in the network, the network distance is at least the same as the Euclidean distance. In contrast, the *network expansion* method performs the query directly from the query point by expanding the nearby vertices in the order of the distances from the query point. As an improvement, the VN^3 method [15] was proposed as a Voronoi-based approach to pre-compute the distances within and across subspaces. The goal was to avoid on-line distance computation in processing kNN queries. Huang et al. [11] addressed the same problem by proposing the *islands approach* that estimates the overhead of pre-computation and the trade-off between query and update performance for kNN queries with various densities of POIs and networks. To cope with C-kNN queries on stationary POIs in the network, Kolahdouzan et al. [14] proposed the Intersection Examination and Upper Bound Algorithm (IE/UBA) to compute the kNN objects of all nodes on the path and the *split points* between adjacent nodes whose nearest neighbors are different. Lately, Cho et al. [3] solved the same problem by introducing UNICONS that incorporates the pre-computed kNN lists into Dijkstra's algorithm such that it outperforms the IE/UBA in dense networks.

In summary, these works hold the assumption that the POIs are static. However, a large number of the spatial applications require the capability to process moving POIs. This requirement raises the issue of managing a very large number of location updates of moving POIs in an index structure. To overcome this challenge, using predictions (i.e., the trajectory of moving objects) to presume the movement of objects has been used in R-tree-based structures (e.g., the *TPR-Tree* and its variants [23, 24]) and B-tree-based structures (e.g., the B^x tree [13]). As an alternative, STRIPE [22] introduces the idea of transforming the trajectories of objects in a D-dimensional space into points in a 2-D space. However, the assumption of being able to predict the trajectories of moving objects is not always realistic. If the prediction of the object movements fails (e.g., pedestrian strolling in a shopping mall), these approaches are inappropriate. Hence the Lazy-update R-tree (LUR-tree) [16] and its extension [17] modify the original R-tree design to support frequent updates with no restriction on object movement. Alternatively, by assuming moving objects in fixed networks, Frentzos proposed the Fixed Network R-tree (FNR-tree) [6], which has a 2-tier R-tree infrastructure, where a forest of 1-D R-tree indexing the moving objects in road segments on top of a 2-D R-tree indexing the network structure.

In general, these tree-based indices suffer from node reconstruction when performing location updates. Therefore, grid-based structures have raised intensive interest due to their simplicity and efficiency in indexing moving objects. Specifically, Xiong et al. proposed LUGrid [27], an update-tolerant on-disk grid index, that outperforms the LUR-tree in terms of update and query costs. Based on this fact,

much of the recent work leverage either an in-memory grid index [4, 7, 19, 28] or an on-disk grid index [26]. Another notable technique was proposed by Hu et al. [9], who introduced the concept of *distance signature* that specifically focuses on indexing the network distance between objects by partitioning the distances into categories to prune the search space. However, this work assumes a network with long edges, which does not apply to our data sets. Consequently, our design of MOVNet utilizes an in-memory grid index to manage the location updates of moving POIs.

Backed by a grid index, many methods have been proposed to process location-based services on moving POIs with Euclidean distances. For instance, Chon et al. [4] first presented an algorithm based on the trajectory of moving POIs overlapping with the grid cells to solve snapshot range queries and kNN queries. By introducing computation capabilities on the mobile client side, MobiEyes [7] proposed a distributed infrastructure to process mobile range queries on dynamic POIs. Similarly, Hu et al. [10] proposed a generic framework to handle continuous queries by introducing the concept of *safe region* through which the location updates from the mobile clients can be further reduced. In contrast, SINA [18] and SEA-CNN [26] were introduced as centralized solutions with the idea of *shared execution* to process continuous range and kNN queries on moving POIs. Yu et al. [28] proposed an algorithm (referred to as YPK-CNN) for monitoring C-kNN queries on moving objects by defining a search region based on the maximum distance between the query point and the current locations of previous kNNs. As an enhancement, Mouratidis et al. [19] presented a solution (CPM) that defines a *conceptual partitioning* of the space by organizing grid cells into rectangles. Location updates are handled only when objects fall within the vicinity of queries hence improving the system throughput. However, the above techniques do not consider network distance computation, which makes them unsuitable for applications where the network-based distance computation is required.

Method	Query type	Distance	POI type
VN^3	kNN	Network	static
IE/UBA	C-kNN	Network	static
UNICONS	C-kNN	Network	static
MobiEyes	C-range	Euclidean	dynamic
YPK-CNN	C-kNN	Euclidean	dynamic
SEA-CNN	C-kNN	Euclidean	dynamic
CPM	C-kNN	Euclidean	dynamic
IMA/GMA	C-kNN	Network	dynamic
MOVNet	kNN, range	Network	dynamic

Table 1: Properties of several query processing methods

Jensen et al. [12] addressed the challenge of query processing on moving POIs in the network. Specifically, this work described an abstract infrastructure on handling location updates of moving POIs in the network and proposed a kNN query algorithm. This work is fundamentally different from MOVNet due to the system assumptions. MOVNet adopts a centralized infrastructure with periodical location updates from moving POIs, while their method assumes that the mobile client is willing to participate in the kNN query processing. Moreover, Mouratidis et al. [20] addressed the issue of processing C-kNN queries in road networks by proposing two algorithms (namely, IMA/GMA) that handle arbitrary object and query movement patterns in the road network. This work utilizes an in-memory data structure to store the network connectivity, therefore it is undesirable to use it for large-size networks (e.g, metro cities) due to the memory requirement. Instead, MOVNet uses an on-disk R-tree structure that has already been intensively studied for large-size 2-D data usage. Although MOVNet is not aimed at continuous query processing in its current form, we believe that a large number of location-based services only require efficient snapshot query processing capabilities. For example, when a user calls the service center to find the nearest taxi, the customer service prefers to locate the taxi as soon as possible and return to the user. The sooner the request is answered, the sooner more incoming calls will be served. In this case, the system throughput is far more important than monitoring the movement of available taxis.

In summary, Table 1 lists the properties of query processing methods described above compared to MOVNet.

3. SYSTEM DESIGN

In this section, we describe our data modeling of the road network, the data structures of indices, and a cell overlapping algorithm that relates the R-tree and the grid index in MOVNet.

3.1 Network Modeling and Assumptions

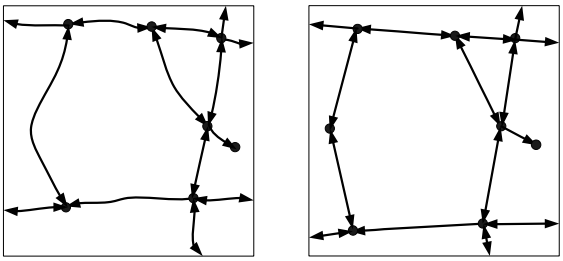
MOVNet focuses on processing moving objects in a road network; we define the road network as follows:

DEFINITION 1. *A road network (or network for short) is a directional weighted graph G consisting of a set of edges (i.e., road segments)* $\mathbb{E}$*, and a set of vertices (i.e., intersections, dead ends)* $\mathbb{V}$*, where* $\mathbb{E} \subseteq \mathbb{V} \times \mathbb{V}$.

DEFINITION 2. *For any network G(*$\mathbb{E}$*,*$\mathbb{V}$*), each edge e is represented as* $e(v_1, v_2)$*, which means it is connected to two vertices* v_1*,* v_2*, where* v_1 *and* v_2 *are the starting and ending vertex, respectively. Let* $v_1 \neq v_2$*. Each edge e is associated with a length, given by a function length(e) :* $\mathbb{E} \rightarrow \mathbb{R}^+$*, where* $\mathbb{R}^+$ *is the set of positive real numbers.*

A network in MOVNet is transformed into a *modeling graph* in memory. Specifically, graph vertices represent the following three cases: (i) the intersections of the network, (ii) the dead end of a road segment, and (iii) the points where the curvature of the road segment exceeds a certain threshold so that the road segment is split into two to preserve the curvature property. The modeling graph is a piecewise approximation of the network. Figure 2(a) shows an example road network, and Figure 2(b) demonstrates the corresponding modeling graph.

There are different objects (e.g., cars, taxis, and pedestrians) moving along the road segments in a network. These objects are identified as the set of *moving objects* $\mathbb{M}$. A moving object $m \in \mathbb{M}$ is a POI located in the network. The location of m at time t is defined as $loc_t(m) = (x_m, y_m)$, where x_m and y_m are the x and y coordinates of m at time t, respectively. A query point q is a moving object $\subset \mathbb{M}$ issuing a location-based spatial query at any time. Currently our design is focused on snapshot range queries (e.g., "finding all taxis within a three mile range from my current location"), and kNN queries (e.g., "finding the 3 nearest police



(a) (b)

Figure 2: An example of road network and its modeling graph.

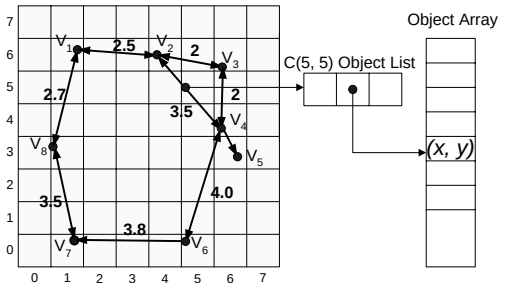


Figure 3: An example network indexed by the grid index and its data storage.

vehicles from my current location"). Note that these queries are processed with network distances. For simplicity, we use the term distance to refer to the network distance in the following sections.

MOVNet assumes that periodic sampling of the positions of moving objects is used to represent their locations as a function of time. This method is also used in [26, 28]. It provides a good approximation on the positions of moving objects. A spatial query submitted by a user at time t_1 is computed based on $loc_{t_0}(\mathbb{M})$, where the system has the last snapshot of moving objects at t_0 with $t_0 \leq t_1$, $t_1 - t_0 < \Delta t$, Δt is a fixed time interval, and the result is valid until $t_0 + \Delta t$.

We define the distance between objects and vertices in the network as follows:

DEFINITION 3. *The distance function of two moving objects m_1 and m_2 at time t is $dist_t(m_1, m_2)$: $loc_t(m_1) \times loc_t(m_2) \rightarrow \mathbb{R}^+$. $dist_t(m_1, m_2)$ denotes the shortest path from m_1 to m_2 in the metric of the network distance at time t. For simplicity, we denote $dist(m_1, m_2)$ as the distance function of m_1 and m_2 at the current time.*

DEFINITION 4. *The distance function of an edge $e(v_1, v_2)$ and a moving object m at time t is defined as $dist_t(e, m)$: $loc(v_1) \times loc_t(m) \rightarrow \mathbb{R}^+$. $dist_t(e, m)$ denotes the shortest path from v_1 to m in the metric of the network distance at time t. For simplicity, we denote $dist(e, m)$ as the distance function of $e(v_1, v_2)$ and m at the current time.*

The distance between two moving objects highly depends on the length of edges and the connectivity of vertices as well as the current locations of the objects. To efficiently manage both stationary network connectivity and dynamic object position updates, MOVNet utilizes a dual-index structure. First, an on-disk R-tree stores the stationary network data. Second, an in-memory grid index supports the position updates of moving objects. Since MOVNet uses two indices for managing the data, it is important to design efficient means to relate these data during query processing. We propose a cell overlapping algorithm that quickly computes the overlapping cells given an arbitrary edge. We elaborate on our design of the index structures in the following section.

3.2 Indexing Structure Design

To record the connectivity and coordinates of vertices in stationary networks, MOVNet utilizes an on-disk R-tree, which has been intensively studied for handling very large 2-D spatial data. Once the edges are retrieved from disk, a corresponding modeling graph is constructed in memory using the following structure. We use a vertex list to store the coordinates of vertices in the graph. For each vertex, a linked list records the edges going out from it. To quickly locate a vertex in the list, MOVNet manages a hash table to map the coordinate of a vertex into its index in the vertex list.

A memory-based grid index is used to manage the locations of moving objects [28]. Without loss of generality, we assume the service space is a square. We can partition the space into a regular grid of cells with a size of $c \times c$. We use $c(column, row)$ to denote a specific cell in the grid index (assuming the cells are ordered from the bottom left corner of the space). At time t, a moving object m has $loc_t(m) = (x_m, y_m)$, therefore it overlaps with cell $c(\lfloor \frac{x_m}{c} \rfloor, \lfloor \frac{y_m}{c} \rfloor)$. Each cell maintains an object list containing the identifiers of enclosed objects. The objects' coordinates are stored in an object array, and the object identifier is the index into this array. Figure 3 shows a part of the network of Figure 2(b) that is managed by a grid index with 8×8 cells. An example object on $e(v_2, v_4)$ is enclosed by $c(5, 5)$. Accordingly, the object list of $c(5, 5)$ records the object identifier and hence we can retrieve the coordinate of the object from the object array.

Cells that overlap with underlying road segments contain the actual data of moving object positions. In general, if the service space is an urban area with a dense network, moving objects can be regarded as uniformly distributed in the grid with a certain granularity.

Given a set of grid cells, retrieving the underlying network can be performed by range queries into the R-tree structure. It is highly desirable to have an algorithm so that for an arbitrary edge, we are able to find overlapping cells very quickly. For this purpose, we devise an incremental algorithm. As we shall see, the CPU time used for computing overlapping cells consumes less than 5% of the query processing time with various settings. It indicates that this method is well suited for online computing. Before we describe our design of the cell overlapping algorithm, we define the concept of *affected cells* as follows:

DEFINITION 5. *Assume that the service space is managed by a grid-based index. We define the set of cells $\{c_1, c_2, ... c_n\}$, where an edge $e(v_1, v_2)$ consecutively overlaps from v_1 to v_2, as the set of affected cells of e.*

For instance, in Figure 3, the affected cells of $e(v_1, v_2)$ are $\{c(1, 6), c(2, 6), c(3, 6), c(4, 6)\}$.

Given an edge $e(v_1, v_2)$, the coordinates of vertex v_1 and v_2 are (x_{v1}, y_{v1}) and (x_{v2}, y_{v2}), respectively. The set of

affected cells of e can be computed with Algorithm 1.

Algorithm 1 Compute-affectedcells (e, c)

```
1: /* e is the edge */
2: /* c is the side length of a cell */
3: m = (y_v2 − y_v1)/(x_v2 − x_v1), b = y_v1 - m · x_v1
4: startX = ⌊ x_v1/c ⌋, startY = ⌊ y_v1/c ⌋
5: endX = ⌊ x_v2/c ⌋, endY = ⌊ y_v2/c ⌋
6: cellList = φ
7: while startX ≠ endX do
8:    if endX > startX then
9:       nextX = startX + 1
10:   else
11:      nextX = startX - 1
12:   end if
13:   nextY = ⌊ (m × nextX × c + b)/c ⌋
14:   for i = startY to nextY do
15:      cellList = cellList ∪ c(startX, i)
16:   end for
17:   startX = nextX, startY = nextY
18: end while
19: for i = startY to endY do
20:   cellList = cellList ∪ c(endX, i)
21: end for
22: return cellList
```

We use straight line segments to represent edges in the network. Therefore, for any edge $e(v_1, v_2)$ in the network, it can be described by a first degree polynomial function in the form of $y = m \cdot x + b$ with $x \in [x_{v1}, x_{v2}]$. Algorithm 1 first captures the gradient m and the y-intercept b of an edge (Line 3). After that, it computes the cells overlapping with the starting and ending vertex of the edge, respectively (Lines 4 - 5). The algorithm follows a step-forward approach where in each step, it moves one cell on the x-axis from the cell overlapping with the starting vertex and calculates the affected cells along the y-axis (Lines 7 - 18). Finally, it terminates once it reaches the cell overlapping with the ending vertex (Lines 19 - 21).

The complexity of Algorithm 1 is linear in the length of the edge. Our experimental evaluation confirms that the CPU cost of Algorithm 1 is very small. More importantly, by introducing Algorithm 1, MOVNet creates tactics to bi-directionally map underlying networks and moving object positions. We present our query design in the following sections showing the flexibility and scalability of this dual-index approach.

4. QUERY DESIGN

In this section, we first describe our design of a mobile range query algorithm. Next, we present the minimum and maximum bounds on the number of grid cells that can overlap with an arbitrary edge. Furthermore, the maximum bound can be used to prune the search space during query processing. Finally, we propose a mobile kNN query algorithm by introducing the concept of a progressive probe and leveraging our range query algorithm.

4.1 Range Query Algorithm

Given a query point q, a value d, a network G and a set of moving objects $\mathbb{M}$, a location-based network distance range query retrieves all POIs of $\mathbb{M}$ that are within the distance d from q at time t. By using the definitions of Section 3, the query can be represented as $rangeQuery_t(q, d)$: $loc_t(q) \times loc_t(\mathbb{M}) \rightarrow \{m_i, i = 1, ..., n\}, \forall\ m_i, dist_t(q, m_i) \leq d$.

We propose a Mobile Network Distance Range query algorithm (MNDR) to facilitate the query processing. It executes the following steps.

First, we know from *Euclidean distance restriction* [21] that for an object m in a network, $dist(q, m)$ is always larger than or equal to the Euclidean distance of q to m. Therefore, if we perform a Euclidean range query with q as the center and d as the radius, the enclosed sub-network G' resulting from the query contains all moving objects within the distance d from q. The advantage of performing this operation as the first step in MNDR results from the fact that only the network data in MOVNet is stored on disk. Once these data are retrieved from the R-tree and the corresponding modeling graph is created, we are able to perform the latter steps in memory. As a comparison, we implemented a baseline algorithm in the simulation based on the idea of *network expansion* [21]. The results show that our approach performs better with a wide range of settings.

Second, the starting vertex of an edge $e(v_1, v_2)$ has the property that if $dist(q, v_1) > d$, the affected cells of the edge are not required to be examined because any moving object on e has a distance greater than d from q. Hence for each vertex in the modeling graph from the first step, MNDR leverages Dijkstra's algorithm [5] to compute the distance from q. In addition, our algorithm avoids unnecessary processing on any edge with a distance from the query point greater than d because no object on these edges will be in the result set.

Finally, for each edge whose starting vertex has a distance $\leq d$, MNDR generates the list of affected cells by using Algorithm 1 and retrieves the corresponding moving objects from the grid index.

Algorithm 2 details MNDR. When MOVNet receives a range query request from a query object q, it first executes a Euclidean distance range query based on the position of q and the range d, see Line 5. After the set of edges and vertices is retrieved, a correlative modeling graph is built in memory (Line 6) that includes all network connectivity information needed for the query. Next, our algorithm adds q as the starting vertex into the modeling graph (Lines 7 - 8). Subsequently, MNDR invokes a modified Dijkstra's algorithm to compute the distance of each vertex from q. Note that we add a constraint d in the distance computing so that any edge $e(v_1, v_2)$ with $dist(q, e) > d$ will not be processed. This constraint prunes the search space. Once the distance computation terminates, the set S (Line 9) consists of the list of vertices whose distances from q are no longer than d and the constituting shortest paths. The algorithm further generates the set of grid cells overlapping with edges in S as shown in Lines 10 - 14. Next, the system extracts moving objects from grid cells in *cellSet*. Note that this step is executed after we obtained the complete set of overlapping grid cells, hence we avoid to retrieve the objects from the grid index repeatedly. Lines 16 - 22 describe a final filtering step that is performed on the retrieved moving objects, ensuring that for every object m, $dist(q, m) \leq d$.

To illustrate our MNDR algorithm with an example, let us assume that the system is processing a network as shown in Figure 3, where the side length of cells is 1.0 unit. A query object q with $dist(q, v2) = 1.0$ submits a range query with a

Algorithm 2 Mobile Network Distance Range Query (q, d)

```
1: /* q is the query object */
2: /* d is the distance */
3: result = φ
4: /* Finding the set of edges E', AND vertices V' overlapped by the circle with center point q, and radius d */
5: (E', V') = Euclidean-range(q, d)
6: G = Create-modeling-graph(E', V')
7: e(v1, v2) = Find-edge(q, E')
8: q = Add-vertex(G, q, e(n1, n2))
9: S = Compute-distance(G, q, d)
10: for each vertex v in S do
11:    for each edge e(v, v') outgoing from v do
12:       cellSet = cellSet ∪ cellOverlapping(e(v, v'), d − dist(q, v))
13:    end for
14: end for
15: result = Retrieve-objects(cellSet, G)
16: for each object m in result do
17:    e(v1, v2) = Find-edge(m, E')
18:    dist(q, m) = min(dist(q, v1) + dist(v1, m), dist(q, v2) + dist(v2, m))
19:    if dist(q, m) > d then
20:       result = result - m
21:    end if
22: end for
23: return result
```

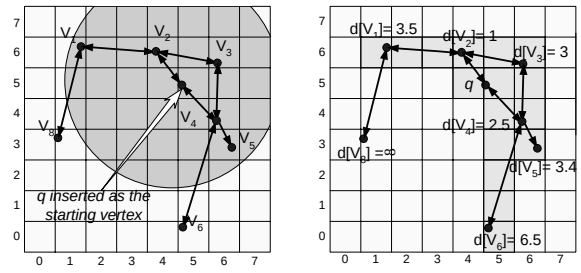


Figure 4: Mobile Network Distance Range query example.

range $d = 3.5$. MOVNet first launches a Euclidean distance range query with q as the center and d as the radius. Consequently, edges overlapping with the shadowed area will be retrieved from the R-tree index and a corresponding modeling graph is built as shown in Figure 4(a). Note that q is inserted as the starting vertex into the modeling graph. Next, Dijkstra's algorithm is invoked. The algorithm runs in a greedy network expansion manner to compute the shortest path from the starting vertex (i.e., q) to other vertices with the distance constraint d. When Dijkstra's algorithm finishes, the distance of each vertex from q is shown in Figure 4(b). Note that v_8 is not processed because $dist(q, v_1) = 3.5$. There is no object in $e(v_1, v_8)$ that will be in the result set. In addition, $S = \langle (v_2, 1), (v_4, 2.5), (v_3, 3), (v_5, 3.4) \rangle$. Based on this information, MNDR computes *cellSet* in Lines 10 - 14, shown as the shadowed cells in Figure 4(b). After that, the moving objects in *cellSet* are retrieved from the grid index to constitute the result set. However, several steps are required to ensure that the distance of each moving object is within range d (Lines 16 - 22). First, some cells might overlap with several edges. For instance, $c(6, 6)$ overlaps with $e(v_2, v_3)$ and $e(v_3, v_4)$. Hence for each object in the result set, MNDR determines which edge the object is located on (Line 17). Second, some objects may be reachable by more than one path from the query point. Our system will only consider the shortest path and examine the path against the range d (Line 18). For example, moving objects on edge $e(v_3, v_4)$ have two paths from q ($q \rightarrow v_2 \rightarrow v_3$, $q \rightarrow v_4$). Our algorithm will compute the distance of each object via each path, and only use the shortest one. Finally, once the distance from q to the object is determined, MNDR confirms that the distance is $\leq d$. For instance, for any object m retrieved from $c(5, 0)$, $dist(q, m) > 3.5$, thus the algorithm removes these objects in Lines 19 - 21.

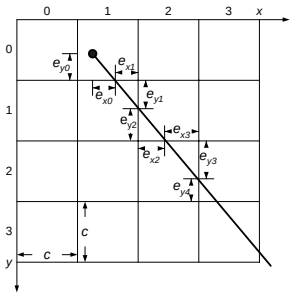


Figure 5: Computing the length of edges with regard to the number of grid cells

The MNDR algorithm is able to compute the exact result set for mobile network distance range queries. However, when computing *cellSet* in Algorithm 2, some cells can be further pruned before the system extracts the corresponding moving objects from the grid index. For instance, in the example illustrated above, every cell overlapping $e(v_4, v_6)$ is in *cellSet*. Clearly, some of the cells should be pruned because their distances from $q > d$. This optimization can be achieved by using the geometric properties introduced in the following section.

4.2 The Minimum and Maximum Number of Cells Overlapping with an Edge

We present an important geometric property that relates an arbitrary edge with the grid cells it overlaps. Since the edge is represented as a straight line segment, the relationship between the length of an edge $e(v_1, v_2)$ and the number of its affected cells can be described as follows.

LEMMA 1. *Assume that the service space is managed by a grid-based index with a cell size of $c \times c$. For an edge $e(v_1, v_2)$ with a set of affected cells $\{c_1, c_2, \ldots. c_n\}$, the maximum length of e is $\sqrt{2} \times c \times n$. The minimum length of e is*

$$\begin{cases} 0 & 1 \leq n \leq 2 \\ \sqrt{\left\lfloor \frac{n-3}{2} \right\rfloor^2 + \left\lfloor \frac{n-2}{2} \right\rfloor^2} \cdot c & n \geq 3 \end{cases}$$

PROOF. Without loss of generality let us consider an edge e in the service space that overlaps with grid cells as shown in Figure 5. Assume the number of affected cells for e is n. Therefore, for $0 \leq e_{xi} \leq c$, $0 \leq e_{yi} \leq c$, we have

$$length(e) = \sqrt{\sum_{i=0}^{n-1} e_{xi}^2 + \sum_{i=0}^{n-1} e_{yi}^2} \quad (1)$$

We observe that, when $e_{xi} = e_{yi} = c$, where $0 \le i \le n-1$, we have the maximum length of e when substituting e_{xi} and e_{yi} in (1)

$$length_{max}(e) = \sqrt{n^2 \cdot c^2 + n^2 \cdot c^2} = \sqrt{2} \cdot c \cdot n$$

To compute the minimum length of e, we observe from Figure 5 that $e_{y1} + e_{y2} = e_{x2} + e_{x3} = c$, and so on, which can be summarized as

$$\begin{cases} e_{x(2j)} + e_{x(2j+1)} = c & 1 \le j \le \lfloor \frac{n-3}{2} \rfloor \\ e_{y(2k-1)} + e_{y(2k)} = c & 1 \le k \le \lfloor \frac{n-2}{2} \rfloor \end{cases} \quad (2)$$

For simplicity, we use E_{xj} to refer to $e_{x(2j)} + e_{x(2j+1)}$ and E_{yk} to refer to $e_{y(2k-1)} + e_{y(2k)}$ from here on.

When $n = 1$, the minimum length of $e = \sqrt{e_{x0}^2 + e_{y0}^2} = 0$, where $e_{x0} = e_{y0} = 0$. Similarly, when $n = 2$, the minimum length of $e = 0$, where $e_{x0} = e_{y0} = e_{x1} = e_{y1} = 0$.

When n is ≥ 3 and odd, we have

$$\begin{cases} \sum_{i=0}^{n-1} e_{xi}^2 = (e_{x0} + e_{x1} + \sum_{j=1}^{\lfloor \frac{n-3}{2} \rfloor} E_{xj} + e_{x(n-1)})^2 \\ \sum_{i=0}^{n-1} e_{yi}^2 = (e_{y0} + \sum_{k=1}^{\lfloor \frac{n-2}{2} \rfloor} E_{yk} + e_{y(n-2)} + e_{y(n-1)})^2 \end{cases}$$

Using the properties in (2), the above equations can be transformed into

$$\begin{cases} \sum_{i=0}^{n-1} e_{xi}^2 = (e_{x0} + e_{x1} + (\lfloor \frac{n-3}{2} \rfloor) \cdot c + e_{x(n-1)})^2 \\ \sum_{i=0}^{n-1} e_{yi}^2 = (e_{y0} + (\lfloor \frac{n-2}{2} \rfloor) \cdot c) + e_{y(n-2)} + e_{y(n-1)})^2 \end{cases}$$

Substituting the corresponding parts of (1) with the above equations, we can conclude that, if $e_{x0} = e_{x1} = e_{x(n-1)} = e_{y0} = e_{y(n-2)} = e_{y(n-1)} = 0$,

$$length_{min}(e) = \sqrt{\lfloor \frac{n-3}{2} \rfloor^2 + \lfloor \frac{n-2}{2} \rfloor^2} \cdot c$$

Similarly, when n is ≥ 3 and even, we have

$$\begin{cases} \sum_{i=0}^{n-1} e_{xi}^2 = (e_{x0} + e_{x1} + \sum_{j=1}^{\lfloor \frac{n-3}{2} \rfloor} E_{xj} + e_{x(n-2)} + e_{x(n-1)})^2 \\ \sum_{i=0}^{n-1} e_{yi}^2 = (e_{y0} + \sum_{k=1}^{\lfloor \frac{n-2}{2} \rfloor} E_{yk} + e_{y(n-1)})^2 \end{cases}$$

Using the same properties as shown in (2), we can conclude that $length_{min}(e) = \sqrt{\lfloor \frac{n-3}{2} \rfloor^2 + \lfloor \frac{n-2}{2} \rfloor^2} \cdot$ c.

We have proved that when $n \ge 3$, in both even and odd cases, $length_{min}(e) = \sqrt{\lfloor \frac{n-3}{2} \rfloor^2 + \lfloor \frac{n-2}{2} \rfloor^2} \cdot$ c. □

Lemma 1 states the minimum and maximum bounds of the length of an edge given a fixed number of cells. We further deduce from Lemma 1 the maximum and minimum number of affected cells with regard to an arbitrary edge.

COROLLARY 1. *Assume that the service space is managed by a grid-based index with a cell size of $c \times c$. For an edge $e(v_1, v_2)$, the maximum and minimum number of affected cells are $\lceil \frac{\sqrt{2} \cdot length(e)}{c} \rceil + 3$, and $\lfloor \frac{length(e)}{\sqrt{2} \cdot c} \rfloor$, respectively.*

PROOF. Lemma 1 tells us that, given an edge $e(v_1, v_2)$, $length(e) \le \sqrt{2} \cdot c \cdot n$; hence we can directly deduce that $n \ge \lfloor \frac{length(e)}{\sqrt{2} \cdot c} \rfloor$.

Similarly, since $length(e) \ge \sqrt{\lfloor \frac{n-3}{2} \rfloor^2 + \lfloor \frac{n-2}{2} \rfloor^2} \cdot c$, hence $length(e) \ge \sqrt{2 \cdot \lfloor \frac{n-3}{2} \rfloor^2} \cdot c$, which leads us to conclude that $n \le \lceil \frac{\sqrt{2} \cdot length(e)}{c} \rceil + 3$. □

We utilize the property of the maximum number of affected cells in Corollary 1 to prune the search space. Let us assume that MNDR generates the list of cells overlapping with an edge $e(v_1, v_2)$ and there are n_1 affected cells. By using Corollary 1 we deduce that a range $d - dist(q, v_1)$ is only able to overlap with at most n_2 cells, where $n_2 < n_1$. Therefore MNDR will only record the first n_2 cells on $e(v_1, v_2)$ into *cellSet*. As an example consult Figure 4(b). We know that $dist(q, v_4) = 2.5$, therefore we only need to record cells on $e(v_4, v_6)$ within a range of 3.5 - 2.5 = 1.0 from v_4. Using the maximum bound on the number of affected cells, MNDR records the first 5 cells on $e(v_4, v_6)$ starting from v_4, even though there are 6 cells overlapping with $e(v_4, v_6)$.

To summarize, Corollary 1 provides a precise range on how edges overlap with grid cells. Our simulation results indicate that this property offers substantial performance improvements when computing the affected cells over long edges (i.e., freeway segments).

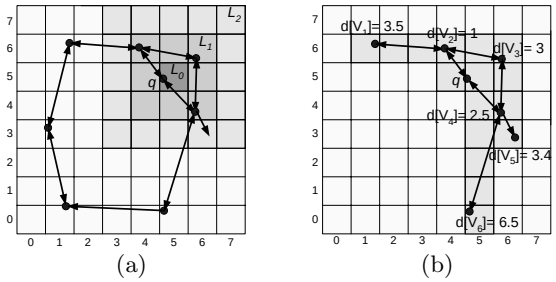
4.3 k Nearest Neighbor Query Algorithm

Given a query point q, a value k, a network G and a set of moving objects $\mathbb{M}$, a network-distance-based k nearest neighbor query retrieves the k objects of $\mathbb{M}$ that are closest to q according to the network distance at time t. Formally, the mobile kNN query is represented as $kNNQuery_t(q, k)$: $loc_t(q) \times loc_t(\mathbb{M}) \rightarrow \{m_i, i = 1, ..., k\}$, where $\forall\ m_j = \{\ \mathbb{M}$ - $m_i\ \}$, $dist_t(q, m_j) \ge dist_t(q, m_i)$.

We present a Mobile k Nearest Neighbor query algorithm (MKNN) leveraging our MNDR algorithm to efficiently compute the kNN POIs from the query point in the network. Moreover, we introduces the concept of *progressive probe* to estimate the result space as the first step of MKNN.

We observe that the grid index in MOVNet enables fine-grained space partitioning. Additionally, the grid index maintains an object list in each grid cell, which can be quickly accessed to retrieve the number of enclosed objects. Therefore, we begin by searching the surrounding area of the query point in the grid index and continuously enlarging the area until we are able find a sub-space that contains kNN POIs in terms of the Euclidean distance. We term this procedure a progressive probe. Note that in the progressive probe, we only retrieve the size of the object list from each cell, while the distance of each object from the query point is not computed because we aim at obtaining an approximate area enclosing kNN objects by the network distance. Our experimental study shows that in 30% to 48% of the test cases the actual kNN objects are bounded by our progressive probe. More importantly, the complexity of retrieving the object list size from each cell is $O(1)$, which is very efficient especially since our grid index is an in-memory structure. We detail the progressive probe next.

We define that cells in the grid index are grouped into levels centered at $c(\lfloor \frac{x_q}{c} \rfloor, \lfloor \frac{y_q}{c} \rfloor)$, where q is a moving object submitting a mobile kNN query. The first level L_0 is the single cell $c(\lfloor \frac{x_q}{c} \rfloor, \lfloor \frac{y_q}{c} \rfloor)$ and cells in the next level are the



(a) (b)

Figure 6: Mobile Network Distance k-NN query example.

surrounding cells of L_0, and so on. Formally, cells in the level L_i ($i \in \{1, 2, \cdots\}$) can be represented as $L_i = c(x_1, y_1) \cup c(x_2, y_2) \cup c(x_3, y_3) \cup c(x_4, y_4)$, where $\lfloor \frac{x_q}{c} \rfloor - i \leq x_1 \leq \lfloor \frac{x_q}{c} \rfloor + i$, $y_1 = \lfloor \frac{y_q}{c} \rfloor + i$, $x_2 = \lfloor \frac{x_q}{c} \rfloor - i$, $\lfloor \frac{y_q}{c} \rfloor - i + 1 \leq y_2 \leq \lfloor \frac{y_q}{c} \rfloor + i - 1$, $x_3 = \lfloor \frac{x_q}{c} \rfloor + i$, $\lfloor \frac{y_q}{c} \rfloor - i + 1 \leq y_3 \leq \lfloor \frac{y_q}{c} \rfloor + i - 1$, and $\lfloor \frac{x_q}{c} \rfloor - i \leq x_4 \leq \lfloor \frac{x_q}{c} \rfloor + i$, $y_4 = \lfloor \frac{y_q}{c} \rfloor - i$. By using the definition above, the progressive probe first retrieves the number of objects in L_0 via the grid index. If there are less than k objects in L_0, it continues to scan the number of objects in the next level of cells, and so on. Figure 6(a) illustrates an example of these steps. Assume the system is maintaining a network as shown in Figure 3 and a query object q in $c(5, 5)$ submits a nearest neighbor query with $k = 10$. The progressive probe first locates q in $c(5, 5)$, which becomes L_0. After that, the number of POIs in $c(5, 5)$ is retrieved from the grid index. If there are less than 10 POIs in L_0, the progressive probe sequentially search the next levels L_i, where $i \in \{1, 2, \cdots\}$, illustrated in the shadowed areas in Figure 6(a). Assuming that at least 10 POIs have been found after the scan in L_2, the probe stops and results in an estimated space for kNN objects in the network. Based on the fact that the R-tree is on secondary storage in MOVNet, MKNN utilizes this estimated area to launch a range query extracting the edges from the R-tree, instead of following a network expansion approach to retrieve a few edges at a time.

Moreover, we introduce the following data structures in MKNN: *candidateObjects* and *unvisitedVertices*. These are minimum priority queues on the value of the distances from the query point. The set of candidate objects is retrieved from the grid index as possible objects in the final result set. The set of unvisited vertices is to be expanded when there are less than k objects found during query processing. Additionally, we manage *resultObjects* as a maximum priority queue in terms of the distance from the query point with a size of k.

Algorithm 3 elaborates on the MKNN algorithm. MKNN first executes the progressive probe in the grid index so that an approximate query result space is created. After that, MKNN uses this subspace as an initial range to invoke the MNDR module so that the corresponding edges are retrieved from the R-tree and the distance of each vertex from q is computed (Lines 8 - 12). Given the example of Figure 6(a), Figure 6(b) demonstrates the correlated modeling graph and the distance to each vertex.

Next, a vertex is de-queued from unvisited vertices (Line 15). For each outgoing edge from the vertex, the set of affected cells is computed and objects are retrieved from the corresponding grid cells and placed into the set of candidate objects (Lines 20 - 23). Lines 24 - 29 describe a filtering

Algorithm 3 Mobile Network Distance k Nearest Neighbor Query (q)

```
 1: /* q is the query object */
 2: /* k is the number of NN objects */
 3: /* c is the side length of cell */
 4: foundkObject = false
 5: visitedVertices = φ
 6: radius = Progressive-probe(q, k)
 7: while foundkObjects = false do
 8:    (E', V') = Euclidean-range(q_x, q_y, radius)
 9:    G = Create-modeling-graph(E', V')
10:    e(v_1, v_2) = Find-edge(q, E')
11:    q = Add-vertex(G, q_x, q_y, e(n_1, n_2))
12:    S = Compute-distance(G, q, ∞)
13:    unvisitedVertices = S - visitedVertices
14:    while unvisitedVertices != NULL do
15:       minVertex = De-queue(univisitedVertices)
16:       if resultObjects.size = k AND
          minVertex.distance ≥ kth resultObject.distance
          then
17:          foundKObjects = true
18:          break
19:       end if
20:       for each edge e(v, v') outgoing from UV do
21:          cellSet = cellSet ∪
             cellOverlapping(e(v, v'), d − dist(q, v))
22:       end for
23:       candidateObjects = candidateObjects ∪
          Retrieve-objects(cellSet, G)
24:       while resultObjects.size < k do
25:          De-queue(candidateObjects) to resultObjects
26:       end while
27:       while Peak(candidateObjects).distance ≤
          kth resultObject.distance do
28:          Swith(De-queue(candidateObjects),
             De-queue(resultObjects))
29:       end while
30:    end while
31:    radius = minVertex.distance
32: end while
33: return resultObjects
```

step consisting of two cases. First, if there are less than k objects in *resultObjects*, MKNN simply de-queues objects from *candidateObjects* into *resultObjects* (Lines 24 - 26). Second, if the distance of the kth result object is greater than the distance of the first element of *candidateObjects*, the kth result object will be de-queued and inserted into *candidateObjects*. Next, *candidateObjects* de-queues an object and inserts it into *resultObjects* (Lines 27 - 29).

The algorithm terminates when *resultObjects* contains k POIs and the distance of the kth result object is less than the distance of the minimum vertex in *univsitedVertices* (Lines 16 - 18). Otherwise, if the last vertex v in the modeling graph (i.e., the vertex with the longest distance to q) is visited and the distance of the kth result object is greater than $dist(q, v)$, MKNN will use $dist(q, v)$ as the radius to launch a range query in the R-tree as a new iteration of MKNN (Line 31). Although this step causes I/O operations as well as the overhead of creating a modeling graph again, MKNN maintains the set of visited vertices in each iteration to avoid visiting these vertices in future iterations

(Line 13). As our simulation results have verified, under various settings, MKNN requires no more than two iterations during query processing in more than 97% of the test cases. Therefore, this method significantly reduces the I/O cost and ensures high system throughput.

5. PERFORMANCE ANALYSIS

We present our theoretical analysis of MOVNet in the following sections. We assume that the network and moving objects are uniformly distributed in one unit square space, which is similar to previous studies [28, 19]. A grid index with $c \times c$ side length manages the moving object location updates. There are a total of E edges and M moving objects in the network.

5.1 Analysis of MNDR

For a MNDR query with a range d, let us assume the query covers an area of $2d \times 2d$. Although the Euclidean distance query in MNDR is in actuality performed within an area of πd^2, our assumption does not change the quality of our analysis. During the processing of MNDR, there are $O(4d^2E)$ edges retrieved from the on-disk R-tree in the Euclidean distance range query. The next step that creates a modeling graph is of complexity $O(4d^2E)$ since every edge will be recorded in the graph. Finding the edge where the query point is located can be achieved during the modeling graph construction. Additionally, inserting the query point into the modeling graph as the starting vertex requires only $O(1)$ operations. The running time of Dijkstra's algorithm to compute the distance of each vertex from the query point is $O(4d^2E \cdot lg(4d^2E))$. Next, MNDR calculates the cell set overlapping with the edges based on the distance information. Note that each edge is examined at most once during the course of this step. Therefore, $O(4d^2E)$ iterations are needed to calculate the overlapping cells. Moreover, since the length of the edge is bounded by d, the total complexity of this step is $O(4d^3E)$. Finally, MNDR retrieves the objects from the grid index and computes the result set. For a range query with a side length of $2d$, the number of overlapping grid cells is $(2d + c)^2/c^2$ [25]. For each cell, we can assume that there are c^2M objects. Hence the number of moving objects retrieved in the final step is $O((2d + c)^2M)$. To sum up, the cost of MNDR can be represented as $O(4d^2E(2 + lg(4d^2E) + d) + (2d + c)^2M)$.

We observe that the cost of MNDR is linear in the number of POIs. Similarly, the system throughput is proportional to the side length of cells (or inversely proportional to the number of cells). Additionally, both factors are lower-bounded by the cost of graph construction, Dijkstra's algorithm, and the overlapping cell computation. Finally, the CPU cost is a quadratic function of d, which means a larger range results in a serious increase in CPU cost.

5.2 Analysis of MKNN

For simplicity, let us assume that the progressive probe results in a subspace containing k nearest neighbor objects. In the case that MKNN needs to expand to a larger space with more iterations, it can be modeled with our cost model times a constant, which does not change the characteristic of our analysis.

Since POIs are uniformly distributed, the subspace containing kNN objects has a size of $\frac{k}{M}$. Therefore, MKNN needs to scan $\frac{k}{M \cdot c^2}$ cells to find the kth object and return a subspace. The subsequent steps that perform a Euclidean distance range query, construct the modeling graph, compute the overlapping cells, and retrieve objects are the same as the ones in MNDR. The cost in these operations can be summarized as $O(\frac{2kE}{M} + \frac{kE}{M} \cdot lg\frac{kE}{M} + \frac{kE}{M} \cdot \sqrt{\frac{kE}{M}} + (\sqrt{\frac{kE}{M}} + c)^2 \cdot M)$. The final step that filters objects from *candidateObjects* into *resultObjects* is bounded by the size of *resultObjects* (i.e., k). In summary, the cost of MKNN is $O(\frac{k}{Mc^2} + \frac{kE}{M} \cdot (2 + lg\frac{kE}{M} + \sqrt{\frac{kE}{M}}) + (\sqrt{\frac{kE}{M}} + c)^2 \cdot M + k)$.

The equation above shows that the CPU cost of MKNN is proportional to k. Furthermore, it is inversely proportional to the number of objects. The explanation is that with more POIs, the search space for finding the kth object becomes smaller, and vice versa. Finally, the system throughput as a function of the cell size is bounded by two factors: the cost from the progressive probe and the cost of retrieving objects from the grid index. A smaller cell size results in more overhead from the progressive probe. In contrast, increasing the size of cells implies that more objects are retrieved from the grid index. Therefore, it is important to investigate the optimal cell size in the simulations. We present our simulation results in the following section as an experimental verification of our theoretical analysis.

6. EXPERIMENTAL EVALUATION

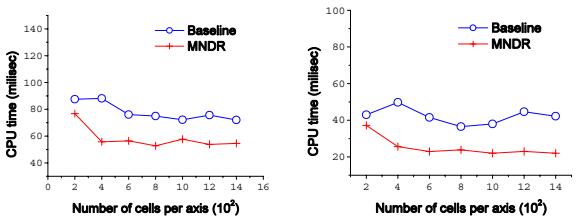
To evaluate the performance of MOVNet, we performed extensive simulations on two real data sets with various road segment densities. The results indicate that MOVNet achieves good throughput with a wide variety of input data.

6.1 Simulator Implementation

We obtained two real data sets from TIGER/Line[1]. The Los Angeles County (LA) data set has 304,162 road segments distributed over an area of 4,752 square miles. The average length of road segments is 0.1066 mile. In contrast, the Santa Barbara County (SB) data set has 47,595 road segments in an area of 3,789 square miles. The average length of road segments is 0.2321 mile. For simplicity, we assume each road segment is bi-directional. The network data is indexed with an R*-Tree [1]. Additionally, we used a network simulator [2] to generate the positions of 100,000 moving objects in each of the road networks.

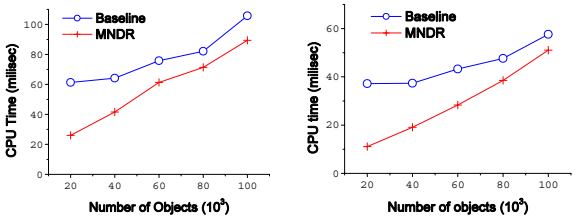
Since there is no existing system targeting the same functionality as MOVNet, we leveraged the concept of *network expansion* [21] to design baseline algorithms for performance comparisons in our simulations. The baseline algorithm for mobile range queries executes as follows. First we retrieve the edge where the query point is located. Next, the closest vertex to the query point is expanded and outgoing edges from this vertex are retrieved. The expansion stops once all vertices whose distances from the query point are less than d have been expanded. After that, for each expanded edge, the overlapping cells are computed and POIs in these cells are retrieved to constitute the result set. Based on the same idea, the baseline algorithm of mobile kNN queries has the following steps. First we locate the road segment on which the query point is moving and compute the affected cells. Next, the corresponding moving objects are retrieved from the affected cells. If there are less than k objects in the re-

[1]http://www.census.gov/geo/www/tiger/



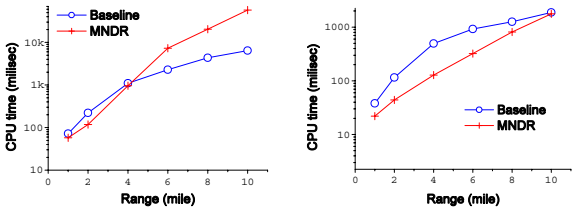
(a) LA county (b) SB county

Figure 7: The CPU time of MNDR as a function of the number of cells



(a) LA county (b) SB county

Figure 8: The CPU time of MNDR as a function of POIs



(a) LA county (b) SB county

Figure 9: The CPU time of MNDR as a function of range

sult set, or the distance from the query point to the closest vertex is less than the distance of the kth object from the query point, the closest vertex is expanded and the outgoing edges from this vertex are retrieved. Afterward, the set of affected cells on the outgoing edges is computed and the corresponding objects are retrieved from the grid index. The vertex expansion process stops when there are k objects in the result set and the distance from query point to the kth object is no greater than the distance from the query point to the closest un-expanded vertex.

We implemented a prototype simulator in Java. The simulation was executed on a workstation with 512 MB memory and a 2.4 GHz Pentium 4 processor. For each test case, our simulator creates a service space with the area equal to the county size. It then opens the R-tree index file of the road segments. Next, an in-memory grid index is created with the positions of the moving objects. Therefore each cell has a list of moving objects it encloses. In the next step, the query generator randomly picks a moving object and launches a query from its location. Table 2 summarizes the parameters used. In each experimental setting we varied a single parameter and kept the remaining ones at their default values. The experiments measured the CPU time (in milliseconds) as the performance of the query processing. For each experimental configuration, the simulator executed 1,000 iterations and reported the average result.

Parameter	Default	Value Range
Number of POIs	50K	10K - 100K
POI distribution	Uniform	Uniform
Number of NNs (k)	10	2 - 128
Radius (mile)	1	1 - 10
Number of cells per axis	1K	200 - 14K

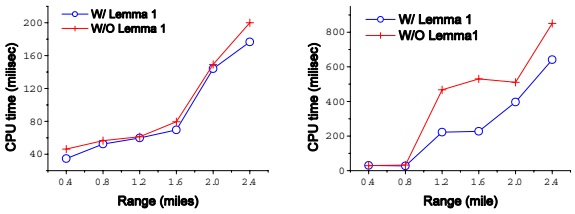
Table 2: Simulation parameters

6.2 Simulation Results

We begin with the study of the performance of MNDR. Figure 7(a) measures the effect of the number of cells in MNDR with the LA county data set. An important observation is that a small number of cells causes the system performance to degrade. On the other hand, there exists a threshold in the number of cells after which the system performance stabilizes. This result follows our theoretical expectation in Section 5. More importantly, MNDR consistently outperforms the baseline algorithm. Figure 7(b) studies the same metric with the SB county data. The CPU cost in SB county is only 50% as compared with the LA county data. This is because the network density in LA county is 5 times that of SB county. The result shows that with the SB county data, MNDR outperforms the baseline algorithm as well. Overall, we conclude that MOVNet scales very well with varying cell numbers.

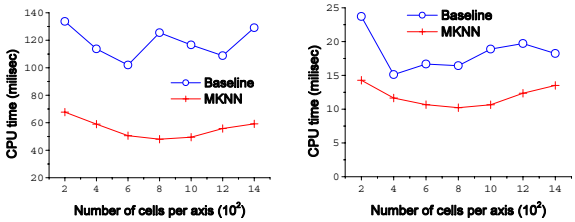
Next, Figure 8(a) illustrates the effect of the number of POIs on the running time of MNDR with the LA county data set. The gradient of the output shows that the CPU time increases linearly with the number of POIs. As we can see, MNDR outperforms the baseline algorithm with various numbers of POIs. In the case of $20K$ POIs, the CPU time of MNDR is about 40% of that of the baseline algorithm. As a comparison, Figure 8(b) plots the results with the same settings with the SB county data. MNDR performs even better, being twice as fast as the baseline algorithm in the case of $20K$ POIs. More importantly, with $100K$ POIs, the processing time in LA county is less than 0.1 second while in SB county is slightly over 40 milliseconds. This demonstrates how efficiently MOVNet executes.

Figure 9(a) plots the CPU time (in logarithmic scale) versus the query range with the LA county set. The CPU time quadratically increases with a larger range. It is noticeable that the performance of MNDR become worse than the baseline algorithm once the range exceeds 4 miles. This behavior is due to the high density of road segments in LA county so that a small portion of the network consists of many edges and the network distance grows much faster than the Euclidean distance. We examined the results, which show that the average Euclidean distance to POIs in the result set is 1.327 miles when the query range is 4 miles. In contrast, the average Euclidean distance is 3.478 miles with the 10-mile range setting. Hence there are a very large number of edges retrieved from the R-tree which will not be processed in the later steps of MNDR. For SB county, which is a much less dense network, MNDR requires less CPU time compared to the baseline algorithm. Although we expect that the performance of MNDR becomes worse than the baseline algorithm once the range exceeds 10 miles (which covers over 10% of the SB county area), our algorithm is a better choice under many practical settings. Efficiently handling large range



(a) LA county (b) LA county, freeways only

Figure 10: The CPU time improvement of using Corollary 1



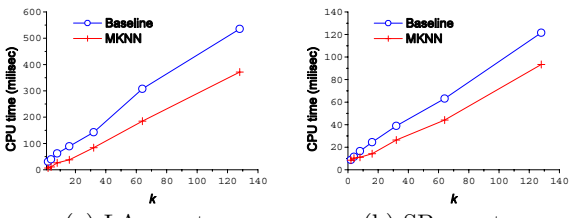
(a) LA county (b) SB county

Figure 11: The CPU time of MKNN as a function of the number of cells.

queries in very dense networks is one of our future directions to improve the MOVNet system.

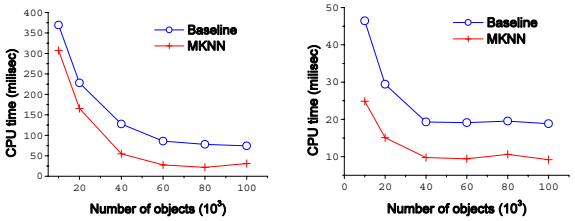
Next, we are interested in the performance improvement when using Corollary 1 in MNDR. Figure 10(a) plots the CPU time when using Corollary 1 to prune the search space in MNDR compared to not using it when handling the LA county data. The performance improvement of using Corollary 1 is slightly over 10% when the range is 2.4 mile and less than 5% when the range is 1.2 mile. We belive this is largely due to the fact that the TIGER/Line data set consists of many very short road segments (0.1066 mile on average). There are only a few cells that overlap with each edge, which implies that there is little chance to prune some cells during the query processing. However, the system improvement of using Corollary 1 is substantial when it is applied to large road segments. To illustrate this fact, we extracted the freeway segments in LA county (the average length of road segments is 2.7127 miles) and performed the simulation on just this network. Figure 10(b) shows the results, with query ranges from 0.15 up to 0.9 time of the average edge length. Clearly, as the range increases, the improvement of system throughput of applying Corollary 1 is very noticable once the range is over 0.8 mile, with a gain of over 50% where the range is 1.6 miles. Hence we conclude that for a network with long road segments, it is very appealing to use Corollary 1 to prune the search space.

Now we study the performance of MKNN. Figures 11(a) and (b) illustrate the performance of MKNN as a function of the number of cells in LA county and SB county, respectively. With both data sets, MKNN steadily requires less CPU time than the baseline algorithm. In denser networks, such as the LA county data set, the performance improvement is larger. Another observation is that the CPU time of MKNN follows our theoretical expectation. A very small number of cells increases the overhead of retrieving POIs from the grid index while a very large number of cells hinders the performance of the progressive probe. In order to



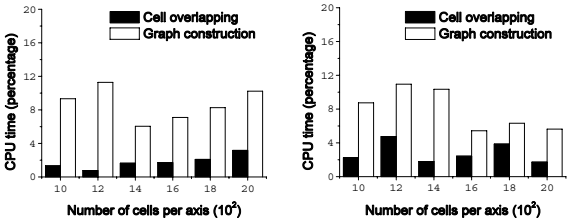
(a) LA county (b) SB county

Figure 12: The CPU time of MKNN as a function of k.



(a) LA county (b) SB county

Figure 13: The CPU time of MKNN as a function of POIs.



(a) MNDR (b) MKNN

Figure 14: The portion of CPU time used in Algorithm 1 and graph construction.

achieve a stable setting, we fixed the number of cells per axis to be 1,000 in the subsequent simulations.

Figures 12(a) and (b) study the performance of MKNN with regard to k in LA county and SB county, respectively. The CPU time grows proportionally with k. More importantly, MKNN outperforms the baseline algorithm. Especially with the increase of k, the progressive probe in MKNN significantly avoids excessive I/O operations on the R-tree. Thus the growth of the CPU time in MKNN is much slower than that of the baseline algorithm as a function of k. In LA county MKNN costs less than 70% of the CPU time of the baseline algorithm while in SB county it is lightly more than 75% where $k = 128$. Finally, when $k = 128$, the CPU time in LA county is less than 0.6 second while it is about 0.12 second in SB county. This clearly shows that MOVNet can support a large value for k.

Figure 13(a) illustrates the performance of MKNN as a function of POIs in LA county. The result shows that the CPU time is inversely proportional to the number of POIs, which is what we expect from the theoretical analysis. An important observation is that MKNN has better system throughput than the baseline algorithm with varying numbers of POIs. The improvement ranges from a factor of 1.2 up to 3.57. Similarly, Figure 13(b) shows that MKNN consistently performs better than the baseline algorithm with a wide range of POIs.

Finally, we study the overhead of Algorithm 1 and the modeling graph construction in both MNDR and MKNN. Figure 14 plots the percentage of CPU time used in these two parts as a function of the number of cells in MNDR and MKNN, respectively. It shows that the cost of computing overlapping cells is less than 5% while the cost of constructing the modeling graph is no more than 12% of the CPU time with varying numbers of cells. Due to the space limitation, we have not plotted the CPU time with regard to other parameters. However, the results exhibit similar characteristics. Therefore, we note that the overhead of these two online computation parts is a very small portion during the query processing.

7. CONCLUSIONS

Location-based services have generated growing interest in the research community. This paper presents an infrastructure aimed at processing location-based services with moving objects in road networks. We propose a cell overlapping algorithm that quickly relates the underlying network and moving objects in memory. Based on the infrastructure of MOVNet, we present two novel algorithms for processing snapshot range queries and kNN queries, respectively. The experimental evaluation suggests that MOVNet is highly efficient in processing these queries with various networks.

We plan to extend our work in several directions. First, our study shows that the system performance deteriorates quadratically with increasing query range in MNDR. Addressing this issue is critical to enable the usage in large cities with very dense networks. Second, we find that usually the network data is not uniformly distributed in the space. Therefore, for some grid cells, the object lists are always empty because no edge overlaps with these cells. Recent techniques, such as presented in [28], have proposed a hierarchical structure to manage skewed data in the grid index in terms of the Euclidean distance. However, there is no existing work to design a grid index managing moving objects with the constraint of underlying networks. We plan to study the modeling of different network distributions and design an improved grid index data structure to reduce the memory requirement. Finally, continuous queries are the most sophisticated query type in location-based services. Although they consume much more computation and memory resources than snapshot queries, they offer an extended view of POI movements and become appealing for monitoring purposes. This functionality is very useful in a number of places, such as 911 call-centers. We are planning to extend the functionality of MOVNet to support continuous queries.

8. REFERENCES

[1] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *SIGMOD Conference*, 1990.

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

[3] H.-J. Cho and C.-W. Chung. An efficient and scalable approach to cNN Queries in a Road Network. In *VLDB*, 2005.

[4] H. D. Chon, D. Agrawal, and A. E. Abbadi. Range and kNN query processing for moving objects in grid model. *MONET*, 8(4), 2003.

[5] E. Dijkstra. A note on two problems in connection with graphs. *Numeriche Mathematik*, 1, 1959.

[6] E. Frentzos. Indexing objects moving on fixed networks. In *SSTD*, 2003.

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

[8] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD Conference*, 1984.

[9] H. Hu, D. L. Lee, and V. C. S. Lee. Distance Indexing on Road Networks. In *VLDB*, 2006.

[10] H. Hu, J. Xu, and D. L. Lee. A Generic Framework for Monitoring Continuous Spatial Queries over Moving Objects. In *SIGMOD Conference*, 2005.

[11] X. Huang, C. S. Jensen, and S. Saltenis. The Islands Approach to Nearest Neighbor Querying in Spatial Networks. In *SSTD*, 2005.

[12] C. S. Jensen, J. Kolárvr, T. B. Pedersen, and I. Timko. Nearest Neighbor Queries in Road Networks. In *ACM GIS*, 2003.

[13] C. S. Jensen, D. Lin, and B. C. Ooi. Query and Update Efficient B+-Tree Based Indexing of Moving Objects. In *VLDB*, 2004.

[14] M. R. Kolahdouzan and C. Shahabi. Continuous K-Nearest Neighbor Queries in Spatial Network Databases. In *STDBM*, 2004.

[15] M. R. Kolahdouzan and C. Shahabi. Voronoi-Based K Nearest Neighbor Search for Spatial Network Databases. In *VLDB*, 2004.

[16] D. Kwon, S. Lee, and S. Lee. Indexing the Current Positions of Moving Objects Using the Lazy Update R-tree. In *MDM*, 2002.

[17] M.-L. Lee, W. Hsu, C. S. Jensen, B. Cui, and K. L. Teo. Supporting Frequent Updates in R-Trees: A Bottom-Up Approach. In *VLDB*, 2003.

[18] M. F. Mokbel, X. Xiong, and W. G. Aref. SINA: Scalable Incremental Processing of Continuous Queries in Spatio-temporal Databases. In *SIGMOD Conference*, 2004.

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

[20] K. Mouratidis, M. L. Yiu, D. Papadias, and N. Mamoulis. Continuous nearest neighbor monitoring in road networks. In *VLDB*, 2006.

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

[22] J. M. Patel, Y. Chen, and V. P. Chakka. STRIPES: An Efficient Index for Predicted Trajectories. In *SIGMOD Conference*, 2004.

[23] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the Positions of Continuously Moving Objects. In *SIGMOD Conference*, 2000.

[24] Y. Tao, D. Papadias, and J. Sun. The TPR*-Tree: An Optimized Spatio-Temporal Access Method for Predictive Queries. In *VLDB*, 2003.

[25] H. Wang, R. Zimmermann, and W.-S. Ku. ASPEN: An Adaptive Spatial Peer-to-Peer Network. In *ACM GIS*, 2005.

[26] 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.

[27] X. Xiong, M. F. Mokbel, and W. G. Aref. LUGrid: Update-tolerant Grid-based Indexing for Moving Objects. In *MDM*, 2006.

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