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

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

Roger Zimmermann
Computer Science Department
National University of Singapore
Singapore 117590
rogerz@comp.nus.edu.sg

ABSTRACT

Location-based services are increasingly popular recently and efficiently supporting queries is a key challenge. Here, we present a novel design to process large numbers of location-based snapshot queries on MOVing objects in road Networks (MOVNet, for short). MOVNet's dual-index design utilizes an on-disk R-tree to store the network connectivities and an in-memory grid structure to maintain moving object position updates. A method to speedily compute the overlapping grid cells in the network relates these two indices and given an arbitrary edge in the space we analyze the minimum and maximum number of grid cells that are possibly affected. Based on the above features we propose algorithms to support mobile network distance range and k nearest neighbor queries. We demonstrate via theoretical analysis and experimental results that MOVNet yields excellent performance with various networks while scaling to 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 games, are popular in many 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 made possible by the usage of car-based or handheld GPS devices in metro cities results in two fundamental system requirements: distance computations within a (road) network and processing of moving Points of Interest (POIs).

An increasing number of applications require query processing of moving POIs based on an underlying network. For example, 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. Note that the mentioned examples require snapshot queries, rather than continuous monitoring (which is another class of applications).

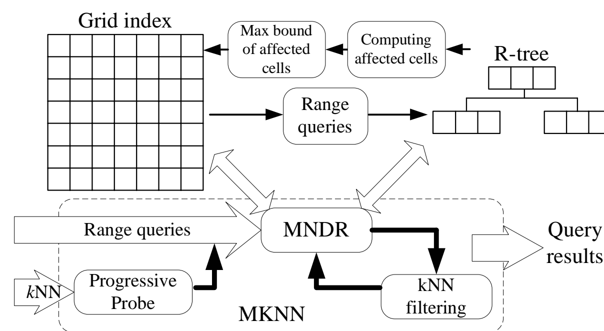

Figure 1: The index structures and query processing modules of MOVNet.

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. Two of the main challenges when supporting POI mobility on an underlying road network are to (a) efficiently manage object location updates and (b) provide fast network-distance computations. To address these issues we have designed a novel system to process location-based queries on MOVing objects in road Networks (MOVNet). The goal is to efficiently execute snapshot range and k nearest neighbor queries over moving POIs within a stationary road network. 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 snapshot query processing capabilities. For instance, when a user calls a service center to find a nearby taxi, the query is instantaneous and as soon as a taxi is dispatched to pick up the customer, the transaction is complete (i.e., the ordering phase). Figure 1 illustrates MOVNet's system infrastructure and components. To handle large networks, MOVNet utilizes an on-disk R*-tree [1] structure to store the necessary connectivity information. Efficient processing of moving object position updates is achieved with an in-memory grid index. An appealing 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. Based on the concept of *affected cells* that form the set of grid cells overlapping with a given edge we present algorithms to execute range as well as kNN queries. Analytical bounds on the minimum and maximum number of affected cells with an arbitrary network edge enable the pruning of the search space during mobile range query processing. In the mobile kNN query al-

*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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

gorithm, we utilize the concept of a *progressive probe* into the grid index to estimate the subspace containing the result set. The performance of our design has been verified vigorously through theoretical analysis and simulations. Our comparison with two state-of-the-art baseline algorithms demonstrate the superior performance of MOVNet.

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 our analysis in Section 6. Finally we conclude with Section 7.

2. RELATED WORK

Processing spatial queries in networks has been studied intensely. Papadias et al. [15] first presented a framework that integrates network and Euclidean information when processing network-based queries. The VN^3 method [11] improves upon this idea with a Voronoi-based approach to pre-compute the distances within and across subspaces. The goal was to avoid on-line distance computations in processing k Nearest Neighbor (kNN) queries. Huang et al. [8] addressed the same problem by proposing the $islands$ $approach$. It estimates the overhead of pre-computation and the trade-off between query and update performance for kNN queries. To cope with Continuous kNN (C-kNN) queries on stationary POIs in a network, Kolahdouzan et al. [10] proposed the Intersection Examination and Upper Bound Algorithm (IE/UBA) to compute the kNN objects of all nodes on a path and the $split$ $points$ between adjacent nodes whose nearest neighbors are different. Recently, Cho et al. [4] solved the same problem by introducing UNICONS that incorporates pre-computed kNN lists into Dijkstra's algorithm such that it outperforms the IE/UBA in dense networks.

The above group of algorithms makes the assumption that the POIs are static. When POIs are dynamic, the key challenge lies in the large number of location updates that must be managed with an appropriate indexing structure. Movement predictions (i.e., the trajectory of moving objects) have been used with R-tree-based structures (e.g., the *TPR*-Tree* [16]). However, these tree-based indices suffer from excessive node reconstruction costs when performing location updates. Therefore, grid-based structures have raised considerable interest due to their simplicity and efficiency in indexing moving objects. Much of the recent work leverages either an in-memory grid index [5, 13, 19] or an on-disk grid index [7, 18]. Following this trend, our design of MOVNet utilizes an in-memory grid index to manage the location updates of moving POIs.

A number of grid-index based methods have been proposed to process location-based services on moving POIs with Euclidean distances. For instance, Chon et al. [5] first presented an algorithm based on the trajectory of moving POIs overlapping with grid cells to solve snapshot range and kNN queries. SINA [12] and SEA-CNN [18] were introduced as centralized solutions with the idea of shared execution to process continuous range and kNN queries on moving POIs. Yu et al. [19] 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. [13] 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 system throughput. However, the above techniques are limited to Euclidean distance computations.

For environments where POIs are dynamic and distances are based on network paths only a few techniques exist. Jensen et al. [9] described an abstract distributed infrastructure for handling location updates of moving POIs in a network in conjunction with a kNN query algorithm. As a centralized alternative, S-GRID [7] was introduced as a means to process kNN queries. A pre-computed structure is maintained with regard to the spatial network data such as to improve the efficiency of query processing. Recently, Mouratidis et al. [14] 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 road networks. This work utilizes an in-memory data structure to store the network connectivity, therefore it is unsuitable for large-sized networks (e.g, metro cities). In contrast, MOVNet uses an on-disk R-tree structure that has a proven performance record for large-sized 2D data usage.

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

We define a road network (or network for short) as 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 $\mathbb{E} \subseteq \mathbb{V} \times \mathbb{V}$. For any network $G(\mathbb{E},\mathbb{V})$, each edge e is represented as $e(v_1, v_2)$, i.e., 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.

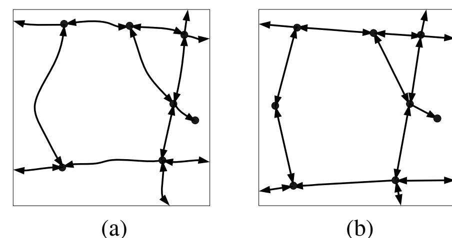

Figure 2: An example of a road network and its corresponding, linearized modeling graph.

The road network is transformed into a $modeling$ $graph$ during query processing. 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 a road segment exceeds a certain threshold so that the road segment is split into two pieces to preserve the curvature property. Although polylines can also be used to represent the edges, we use a set of line segments to represent an edge due to the nature of our data set. As a result, the modeling graph is a piecewise approximation of the network. For example, Figure 2(a) shows a small road network, and Figure 2(b) illustrates 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 known 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 $q \in \mathbb{M}$ is a moving

object issuing a location-based spatial query at different times. Currently our design is focusing on snapshot range queries (e.g., “find all taxis within a three mile range 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 moving object positions conveys their locations as a function of time. This method is commonly used (see [18]) as it provides a good approximation of the moving object positions. Our primary goal is to reduce the evaluation cost during query processing. A spatial query submitted by a user at time t_1 is computed based on $loc_{t_0}(\mathbb{M})$. The system has the lastest snapshot of moving objects at t_0, where $t_0 \leq t_1$, $t_1 - t_0 < \Delta t$, and Δt is a fixed time interval; the result is valid until $t_0 + \Delta t$.

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

The distance between two moving objects depends on the length of edges and the connectivity of vertices as well as the current locations of the objects. We elaborate on our dual-index structure designed to facilitate distance computations in the following section.

3.2 Dual-Index Structure Design

To record the connectivity and coordinates of vertices in stationary networks, MOVNet utilizes an on-disk R*-tree, a data structure 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 array to store the coordinates of vertices in the graph. For each vertex, the array maintains a list recording its outgoing edges. To quickly locate a vertex in the array, MOVNet manages a hash table to map the coordinate of a vertex into its index in the vertex array.

A memory-based grid index is used to manage the locations of moving objects [19]. Without loss of generality, we assume that the service space is a square. We can partition the space into a regular grid of cells with a size of $l \times l$. We use $c(column, row)$ to denote a specific cell in the grid index (assuming the cells are ordered from the lower 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 $c(\lfloor \frac{x_m}{l} \rfloor, \lfloor \frac{y_m}{l} \rfloor)$. 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 of 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.

Given a set of grid cells, retrieving the underlying network can be transformed into range queries on the R-tree. It is highly desirable to have an algorithm so that for an arbitrary edge, we are able to find the set of overlapping cells very quickly. Although this is similar to the line rasterization algorithm (e.g., Bresenham's Algorithm [2]), it is noteworthy to point out that these existing algorithms only obtain an approximation of the overlapping cells (or

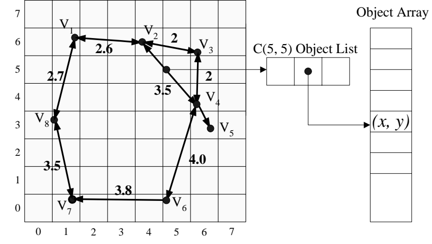

Figure 3: An example network indexed by the grid index and its data storage.

pixels, in that case). In contrast, our goal is to compute the complete set of overlapping cells. Therefore, we devise an incremental algorithm.

First, let us assume that the service space is managed by a grid-based index. We define the set of cells $\{c_1, c_2, \ldots, c_n\}$, which are consecutively overlapped from $v1$ to $v2$ by an edge $e(v1, v2)$, 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 vertices 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: /* l is the side length of a cell */
3: $m = \frac{y_{v2} - y_{v1}}{x_{v2} - x_{v1}}$, $b = y_{v1}$ - $m \cdot x_{v1}$
4: $startX = \lfloor \frac{x_{v1}}{l} \rfloor$, $startY = \lfloor \frac{y_{v1}}{l} \rfloor$
5: $endX = \lfloor \frac{x_{v2}}{l} \rfloor$, $endY = \lfloor \frac{y_{v2}}{l} \rfloor$
6: $cellList = \phi$
7: **while** $startX \neq endX$ **do**
8: **if** $endX > startX$ **then**
9: $nextX = startX + 1$
10: **else**
11: $nextX = startX$ - 1
12: **end if**
13: $nextY = \lfloor \frac{m \times nextX \times c + b}{c} \rfloor$
14: **for** $i = startY$ to $nextY$ **do**
15: $cellList = cellList \cup$ c($startX$, i)
16: **end for**
17: $startX = nextX$, $startY = nextY$
18: **end while**
19: **for** $i = startY$ to $endY$ **do**
20: $cellList = cellList \cup$ c($endX$, i)
21: **end for**
22: return $cellList$

We use straight line segments to represent edges in the network. Therefore, any edge $e(v_1, v_2)$ 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 results show that the CPU time used for comput-

ing overlapping cells consumes less than 5% of the query processing time with various settings. This indicates that our method is well suited for online computing. More importantly, by introducing Algorithm 1, MOVNet creates a means 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. Then, the maximum bound is 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. First, we know from the *Euclidean distance restriction* [15] property that the distance $dist(q, m)$ for object m in a network is always larger than or equal to the Euclidean distance d of q to m. We observe that only the network data in MOVNet is stored on disk. Therefore, we first perform a Euclidean range query with q as the center and d as the radius to retrieve the network from the R-tree and to create the corresponding modeling graph. After that, we are able to perform the later steps efficiently in memory. 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 during this first pass because any moving object on e has a distance greater than d from q. Hence for each vertex in the modeling graph, MNDR leverages Dijkstra's algorithm [6] 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. 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. To illustrate the 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 range d = 3.5. MOVNet first invokes a Euclidean distance range query with q as the center and d as the radius (Line 5 of Algorithm 2). 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) (Line 6). Note that q is inserted as the starting vertex into the modeling graph (Line 8). Next, Dijkstra's algorithm is invoked (Line 9). We add a constraint d in the distance computation so that any edge $e(v_1, v_2)$ with $dist(q, e)$ $> d$ will not be processed, which avoids excessive computation on edges that are out of range. When Dijkstra's algorithm finishes, the distance of each vertex from q is shown in Figure 4(b). 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 $cellSet$ by using our cell overlapping algorithm in Lines 10 - 14, shown as the dark-grey cells in Figure 4(b). After that, the moving objects in $cellSet$ are retrieved from the grid

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

1: /* q is the query object */
2: /* d is the distance */
3: $result = \phi$
4: /* Finding the set of edges $\mathbb{E}'$, AND vertices $\mathbb{V}'$ overlapped by the circle with center point q, and radius d */
5: ($\mathbb{E}'$, $\mathbb{V}'$) = Euclidean-range(q, d)
6: G = Create-modeling-graph($\mathbb{E}'$, $\mathbb{V}'$)
7: e = Object-map-matching(q, $\mathbb{E}'$)
8: q = Add-vertex-into-graph(G, q, e)
9: S = Compute-distance(G, q, d)
10: **for** each vertex v in S **do**
11: **for** each edge e outgoing from v **do**
12: $cellSet = cellSet \cup cellOverlapping(e, d - dist(q, v))$
13: **end for**
14: **end for**
15: $result$ = Retrieve-objects($cellSet$, G)
16: **for** each object m in $result$ **do**
17: $e(v_1, v_2)$ = Object-map-matching(m, $\mathbb{E}'$)
18: $dist(q, m) = min(dist(q, v_1) + dist(v_1, m), dist(q, v_2) + dist(v_2, m))$
19: **if** $dist(q, m) > d$ **then**
20: $result = result$ - m
21: **end if**
22: **end for**
23: return $result$

index to constitute the result set. However, several post-processing steps are required to ensure that the distance of each moving object is within range d. 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) through a map matching process. Second, some objects may be reachable via more than one path from the query point. MOVNet 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 ($q \rightarrow v_2 \rightarrow v_3$, and $q \rightarrow v_4$). MNDR 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 $\leq d$. 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.

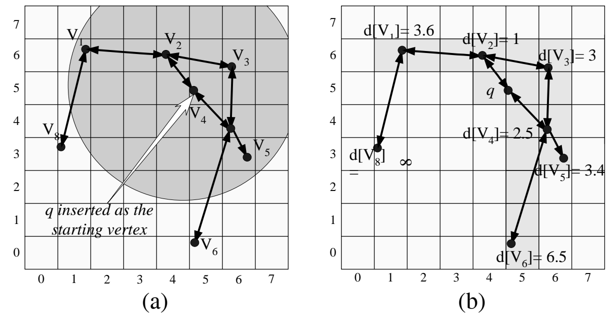

Figure 4: A Mobile Network Distance Range (MNDR) query example.

When we compute $cellSet$ in Algorithm 2 (Lines 10 -14), some cells can be further pruned before the system retrieves the moving

objects from the corresponding grid index. For instance, in the example illustrated above, $e(v_4, v_6)$ overlaps with six cells. Some of the cells can be pruned because their distances from $q > d$. This optimization can be achieved by using the geometric properties as described in the following section.

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

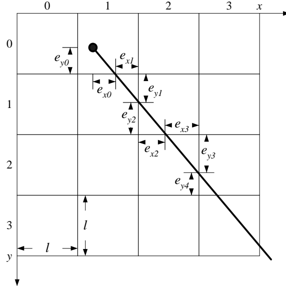

Figure 5: Computing the length of edges with regard to the number of grid cells

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 $l \times l$. 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 $\sqrt{2} \times l \times n$. The minimum length of e is*

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

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 that the number of affected cells for e is n. Therefore, for $0 \leq e_{xi} \leq l$, $0 \leq e_{yi} \leq l$, 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 $e_{xi} = e_{yi} = l$, where $0 \leq i \leq n-1$, we obtain the maximum length of e when substituting e_{xi} and e_{yi} in (1)

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

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

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

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 Eqn(2), the above equations can be transformed into

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

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

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

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 Eqn(2), we can conclude that $length_{min}(e) = \sqrt{\lfloor \frac{n-3}{2} \rfloor^2 + \lfloor \frac{n-2}{2} \rfloor^2} \cdot l$.

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

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 $l \times l$. For an edge $e(v_1, v_2)$, the maximum and minimum number of affected cells are $\lceil \frac{\sqrt{2} \cdot length(e)}{l} \rceil + 3$, and $\lfloor \frac{length(e)}{\sqrt{2} \cdot l} \rfloor$, respectively.*

PROOF. We know from Lemma 1 that, given an edge $e(v_1, v_2)$, $length(e) \leq \sqrt{2} \cdot l \cdot n$; hence we can directly deduce that $n \geq \lfloor \frac{length(e)}{\sqrt{2} \cdot l} \rfloor$.

Similarly, since $length(e) \geq \sqrt{\lfloor \frac{n-3}{2} \rfloor^2 + \lfloor \frac{n-2}{2} \rfloor^2} \cdot l$, it follows that $length(e) \geq \sqrt{2 \cdot \lfloor \frac{n-3}{2} \rfloor^2} \cdot l$, which leads us to conclude that
$n \leq \lceil \frac{\sqrt{2} \cdot length(e)}{l} \rceil + 3$. □

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 of 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 $e(v_4, v_6)$.

In summary, 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, a 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 $\forall\, m_j = \{\, \mathbb{M}$ - $m_i\, \}$, $dist_t(q, m_j) \geq dist_t(q, m_i)$.

To cope with this type of query, we propose 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. 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 subspace 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 to obtain an approximate area enclosing kNN objects within network distance. Our experimental study shows that in 30% to 48% of the test cases the actual number of 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.

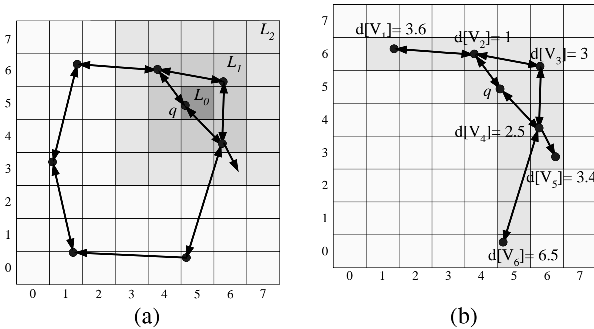

Figure 6: A Mobile Network Distance k-NN (MKNN) query example.

We define that cells in the grid index are grouped into levels centered at $c(\lfloor \frac{x_q}{l} \rfloor, \lfloor \frac{y_q}{l} \rfloor)$, where q is a moving object submitting a mobile kNN query and l is the side length of a grid cell. The first level L_0 is the single cell $c(\lfloor \frac{x_q}{l} \rfloor, \lfloor \frac{y_q}{l} \rfloor)$ and cells at the next level are the surrounding cells of L_0, and so on. Formally, cells in levels 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 $\lfloor \frac{x_q}{l} \rfloor - i \leq x_1 \leq \lfloor \frac{x_q}{l} \rfloor + i$, $y_1 = \lfloor \frac{y_q}{l} \rfloor + i$, $x_2 = \lfloor \frac{x_q}{l} \rfloor - i$, $\lfloor \frac{y_q}{l} \rfloor - i + 1 \leq y_2 \leq \lfloor \frac{y_q}{l} \rfloor + i - 1$, $x_3 = \lfloor \frac{x_q}{l} \rfloor + i$, $\lfloor \frac{y_q}{l} \rfloor - i + 1 \leq y_3 \leq \lfloor \frac{y_q}{l} \rfloor + i - 1$, and $\lfloor \frac{x_q}{l} \rfloor - i \leq x_4 \leq \lfloor \frac{x_q}{l} \rfloor + i$, $y_4 = \lfloor \frac{y_q}{l} \rfloor - i$. 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 at 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 searches 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. Because the R-tree is on secondary storage in MOVNet and the number of disk I/Os should be minimized, 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.

We also introduce the following data structures: $candidateObjs$ 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 $resultObjs$ 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 $unvisitedVertices$ (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 $candidateObjs$ (Lines 21 - 24). After that, we examine two possible cases: First, if there are less than k objects in $resultObjs$, MKNN de-queues objects from $candidateObjs$ into $resultObjs$ (Lines 25 - 27). Second, if the distance of the kth result object is greater than the distance of the first element of $candidateObjs$, the kth result object will be de-queued and inserted into $candidateObjs$. Next, $candidateObjs$ de-queues an object and inserts it into $resultObjs$ (Lines 28 - 30).

The algorithm terminates when $resultObjs$ contains k POIs and the distance of the kth result object is less than the distance of the minimum vertex in $univsitedVertices$ (Lines 17 - 20). 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 32). 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. COMPLEXITY ANALYSIS

We present our theoretical analysis of MOVNet in the following

Algorithm 3 Mobile Network Distance kNN Query (q, k)

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

sections. We assume that the network and moving objects are uniformly distributed in one unit square space (i.e., for an object m, $0 \leq x_m < 1$, and $0 \leq y_m < 1$). This is an optimal simplification, which is similar to previous studies [13, 19]. A grid index with $l \times l$ side length manages the moving object location updates. The total number of edges and moving objects in the network are E and M, respectively.

5.1 Analysis of MNDR

For a MNDR query with a range d, let us assume the query covers an area of $4d^2$. 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 $O(d^2E)$ 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 $O(d^2E)$ 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 $O(d^2E \cdot lg(d^2E))$. 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, $O(d^2E)$ 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 $O(d^3E)$. 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 $(2d+l)^2/l^2$ [17]. For each cell, we can assume that there are l^2M objects. Hence the number of moving objects retrieved in the final step is $O((2d+l)^2M)$. To sum up, the cost of MNDR can be represented as $O(d^2Elg(d^2E) + (2d+l)^2M)$.

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 we assume that POIs are uniformly distributed, the subspace containing kNN objects has a size of $\frac{k}{M}$. Therefore, MKNN needs to scan $\frac{k}{M \cdot l^2}$ cells to find the kth object and return a subspace. The subsequent steps in MKNN 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. Hence the cost in these operations can be summarized as $O(\frac{kE}{M} \cdot (2 + lg\frac{kE}{M} + \sqrt{\frac{kE}{M}}) + (\sqrt{\frac{kE}{M}} + l)^2 \cdot M)$. 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 can be simplified as $O(\frac{k}{Ml^2} + \frac{kE}{M} \cdot \sqrt{\frac{kE}{M}} + (\sqrt{\frac{kE}{M}} + l)^2 \cdot M)$.

The equation above shows that the CPU cost of MKNN is proportional to k. The explanation is that with an increasing k, MKNN needs to search for a larger space to find the query result. Additionally, the CPU cost is inversely proportional to the number of objects. This is because 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. 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 a real dense road network. The result indicates that MOVNet achieves good throughput with a wide variety of data settings. In Section 6.1 we start by describing the data sets used in our simulation and our simulator implementation. Experimental results and the corresponding discussion are presented in Section 6.2.

6.1 Simulator Implementation

We obtained a real data set 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 miles. For simplicity, we assume that each road segment is bi-directional. The network data is indexed with an R*-Tree. Additionally, we used a network simulator [3] to generate the positions of 100,000 moving objects in the road network.

Existing work, such as IMA/GMA, only focus on C-kNN query processing. This method differs from the functionality of MOVNet that focuses on snapshot range query and kNN query processing. Therefore, we leveraged the concept of $network\ expansion$ [15] 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 for 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 result 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 simulator in Java. The simulation was executed on a workstation with 1 GB memory and a 3.0 GHz Xeon processor. We arranged the road segments of the LA county data set into a R*-tree index file, in which we set the page size to 4KB. Each road segment is stored in a MBR bounded by its starting and ending coordinates. To achieve fair comparison, our baseline algorithms also use this R*-tree index structure to speed up the edge retrieval during query processing. For each test case, our simulator creates a service space with the area equal to the LA county size. It then opens the R*-tree index file and use a buffer for caching the disk pages read by MOVNet with a size of 10 pages. Next, an in-memory grid index is created with the positions of the moving objects. To simplify the map-matching process, we assume that object locations always fall along the road segments. In the next step, the query generator randomly picks a moving object and launches a query from its location. Table 1 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) and the number of disk page access as the performance of the query processing. For each experimental configuration, the simulator executed 1,000 iterations and reported the average result.

6.2 Simulation Results

We are first interested in verifying the update costs from POIs in MOVNet. Since we use an in-memory grid index to handle these updates, there is no disk access on the R*-tree index file, which is on secondary storage. Therefore, we measured the CPU time of the update processing. Note that the update and query processing should be finished in one update cycle to ensure the correctness of the query results. We assume that at the beginning of each update period, 20% of the POIs submit their new positions. Figure 7 shows that when there are 20,000 updates messages in one period (i.e., 100,000 POIs), MOVNet is able to record these changes in about 200 miliseconds. Additionally, the update cost is proportional to the number of update messages. Therefore, it is possible to estimate the CPU time that is required to process updates.

Table 1: Simulation parameters

Parameter	Default	Value Range
Number of POIs	50K	10K - 100K
POI distribution	Uniform	Uniform
Number of NNs (k)	50	2 - 128
Radius (miles)	5	2 - 10
Number of cells per axis	1K	200 - 1,400

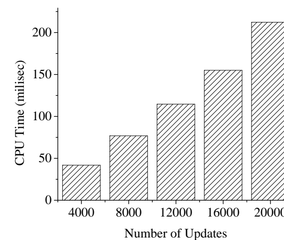

Figure 7: The CPU time of update cost as a function of POIs

We start to verify the performance of MNDR. Figure 8(a) illustrates the effect of the number of cells with the LA county data set. The results show that MNDR requires less than half of the CPU time compared with the baseline algorithm. Correspondingly, Figure 8(b) studies the page accesses of both algorithms. As we can see, the baseline algorithm consumes more than 3,000 page accesses with various cell sizes. As comparison, MNDR requires less than 100 page accesses during query processing. An important observation is that a small number of cells cause the CPU time of MNDR to degrade. On the other hand, the disk access of MNDR is stable with different cell sizes. This can be explained by the fact that a disk access only occurs when we retrieve the road segments from the R*-tree file. Since we use a fixed range in this test, the number of disk accesses is not affected by changing the cell size. However, a larger cell size will result in a larger number of POIs being retrieved from the grid index during query processing. Therefore, the CPU time expended in this portion is larger than with smaller cell sizes. Overall, we conclude that MOVNet scales very well with varying cell numbers. Note that with MNDR, the setting of 1,000 cells per axis achieves a stable and optimal performance, hence we set the default number of cells per axis to be 1,000 in our other tests.

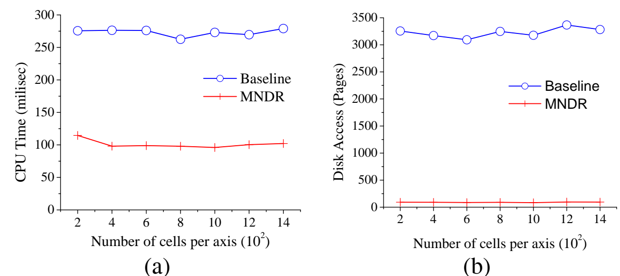

Figure 8: The performance of MNDR as a function of the number of cells

Next, Figure 9(a) illustrates the effect of the number of POIs on

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

the execution time of MNDR. 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 30% of that of the baseline algorithm. Additionally, the output shows that the CPU time increases linearly with the number of POIs, which follows our complexity analysis expectation. The very small gradient of the MNDR output suggests that MOVNet is very scalable to support a very large number of POIs. More importantly, with $100K$ POIs, the processing time for LA county is about 0.1 seconds. This demonstrates how efficiently MOVNet executes. Figure 9(b) plots the disk accesses of both algorithms. Similarly to the CPU time outputs, MNDR performs consistently much lower than the baseline algorithm.

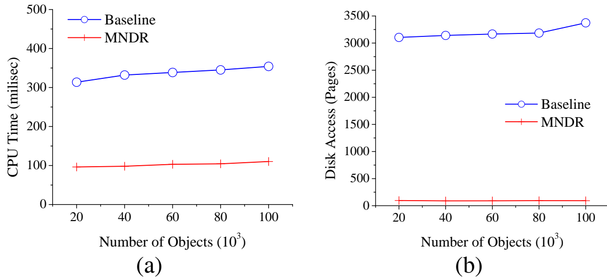

Figure 9: The performance of MNDR as a function of POIs

Figure 10(a) plots the CPU time (with logarithmic scale) versus the query range with the LA county set. The CPU time quadratically increases with a larger range. When the range is 4 miles, MNDR costs 0.076 seconds. Processing a range of 8 miles requires 0.2 seconds by using MNDR compared with 0.65 seconds when using the baseline algorithm. Additionally, MNDR always consumes about 40% of the CPU time compared with the baseline algorithm during query processing. Figure 10(b) plots the corresponding page accesses. The output corresponds to the CPU output, as well as our complexity analysis results. Assuming the road network is uniformly distributed, the number of edges grows quadratically with the increase of the range. Since these edges must be retrieved from the R-tree file during query processing, the performance of MNDR is deteriorating correspondingly.

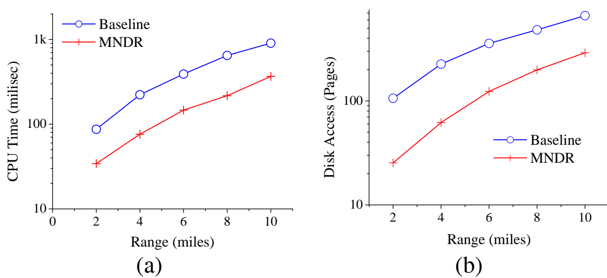

Figure 10: The performance of MNDR as a function of range

Next, we are interested in the performance improvement when using Corollary 1 in MNDR. Figure 11(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 about 10% when the range is 6.0 miles and less than 5% when the range is 2.0 miles. We belive this is largely due to the fact that the TIGER/Line data set consists of many very short road segments (0.1066 miles 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 query processing. However, the system improvement by 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 11(b) shows the results, with query ranges from 2.0 up to 10.0 miles. The results indicate that the improvement of system throughput by applying Corollary 1 is very noticable. Especially when the range is 6 miles, the system performance achieves a gain of over 30%. Hence we conclude that for a network with long road segments, it is very appealing to use Corollary 1 to prune the search space.

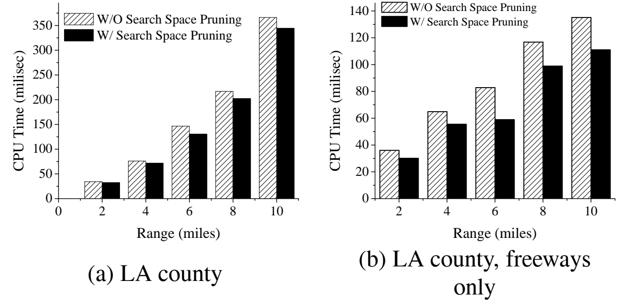

Figure 11: The CPU time improvement of using Corollary 1

Now we study the performance of MKNN. Figures 12(a) and (b) illustrate the CPU time and disk accesses of MKNN as a function of the number of cells, respectively. An observation is that the throughput of MKNN is relatively stable when the number of cells per axis exceeds 400. This is because when we use a fairly large cell size (e.g., 200 cells per axis), the grid index only provides a very coarse-grained space partition. Hence the progressive probe results in a space with lots of unnecessary road segments, which must be retrieved in later steps. However, the performance of MKNN becomes stable when we choose small cell sizes. Hence we set the default number of cells per axis to be 1,000 in our other MKNN tests. More importantly, MKNN consistently requires less CPU time as well as disk accesses than the baseline algorithm. In general, MKNN costs less than 50 milliseconds to process a kNN query with k = 50.

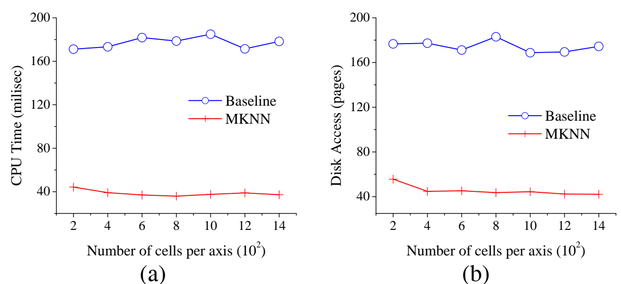

Figure 12: The performance of MKNN as a function of the number of cells.

Figure 13(a) plots the performance of MKNN with regard to k. The CPU time grows proportionally with k. More importantly, MKNN outperforms the baseline algorithm. The growth of the CPU time in MKNN is much slower than that of the baseline algorithm as a function of k. MKNN costs less than 80% of the CPU time of the baseline algorithm where k = 128. Figure 13(b) shows

the disk accesses of both algorithms. The gradient of MKNN output is very small, which suggests that with the increase of k, the progressive probe in MKNN significantly avoids excessive I/O operations on the R-tree. Finally, when k = 128, the CPU time in LA county is less than 0.5 seconds. This clearly shows that MOVNet can support a very large value of k.

Figure 14(a) illustrates the CPU time of MKNN as a function of the number of POIs. The result shows that the CPU time is inversely proportional to the number of POIs, which is what we expect from the theoretical analysis. With a larger number of POIs, the performance of MKNN becomes better. This characteristic ensures that MOVNet is very applicable for use in metro areas. When there are $100K$ POIs in the service area, processing a kNN query with k = 50 requires only 26 milliseconds. Another 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 4.23 up to 5.80. Figure 14(b) shows the disk access counts with regard to both MKNN and the baseline algorithm, which correlates with our CPU time measurement result.

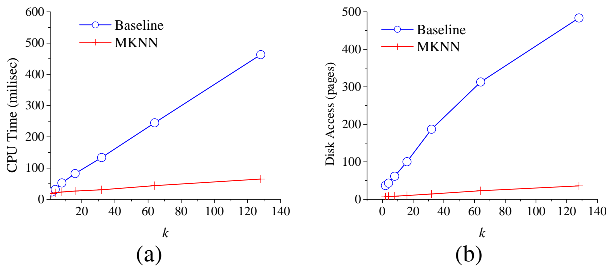

Figure 13: The performance of MKNN as a function of k.

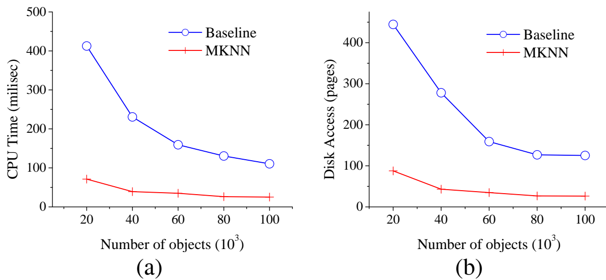

Figure 14: The performance of MKNN as a function of POIs.

Based on our simulation results and analysis, we conclude that the performance of MOVNet scales very well with various settings. It consumes less CPU time in all test cases. The performance difference between MOVNet and the baseline algorithms is much more distinguishable with regard to disk page access.

7. CONCLUSIONS

Location-based services have generated growing interest in the research community. This paper presents an infrastructure aimed to process 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 a real dense road network.

We plan to extend our work in several directions. First, our study currently assumes a static network. However, incorporating some dynamic network updates, such as the real-time traffic information, will be critical for many location-based services, especially those for metro usages. We would like to extend our work to support a dynamic underlying network. Additionally, 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] J. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4(1):25–30, 1965.

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

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

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

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

[7] X. Huang, C. S. Jensen, H. Lu, and S. Saltenis. S-GRID: A Versatile Approach to Efficient Query Processing in Spatial Networks. In *SSTD*, 2007.

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

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

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

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

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

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

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

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

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

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

[18] 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.

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