Processing of Continuous Location-Based Range Queries on Moving Objects in Road Networks

Haojun Wang, *Member*, *IEEE*, and Roger Zimmermann, *Senior Member*, *IEEE*

Abstract—With the proliferation of mobile devices, an increasing number of urban users subscribe to location-based services. This trend has led to significant research interest in techniques that address two fundamental requirements: road network-based distance computation and the capability to process moving objects as points of interests. However, there exist few techniques that support both requirements simultaneously. To address these challenges, we propose a novel approach to process continuous range queries. We build on our previous work of an infrastructure that supports location-based snapshot queries on MOVing objects in road Networks (MOVNet). We introduce several significant features to enable continuous queries. The dual index structure that we proposed for MOVNet has been appropriately modified. We further appoint a number of *connecting vertices* in each cell and precompute the distances among them to expedite query processing. Most importantly, to alleviate the effects of frequent object updates, we introduce a *Shortest-Distance-based Tree* (SD-Tree). We illustrate that the network connectivity and distance information can be preserved and reused by the SD-Tree when the query point location is updated; hence, reducing the continuous query update cost. Our experimental results demonstrate that our method yields excellent performance with a very large number of moving objects.

Index Terms—Spatial databases and GIS, location-dependent and sensitive.

1 Introduction

Recently, more and more mobile users are willing to subscribe to location-based services, such as road-side assistance, location-based games, and location-sharing social networks, in many—especially urban—areas. This raised enormous research interest in designing novel and scalable location-based services in highly mobile environments.

A number of recently proposed techniques incorporate Point of Interests (POI) mobility or network-distance processing, but often not both. The main challenges when supporting POI mobility on an underlying road network are to 1) efficiently manage object location updates and 2) provide fast network-distance computations. To cope with these challenges, we proposed a novel infrastructure that supports location-based query processing on MOVing objects in road Networks (MOVNet) [26]. MOVNet is a centralized solution that combines an on-disk R*-tree [1] structure to store the connectivity information of the road network with an in-memory grid index to efficiently process moving object position updates. The inherent inspiration of using such a dual-index structure is based on the fact that the R-tree structure has been widely studied for efficiently handling large-size stationary spatial data sets and the grid index has been verified for suitably managing dynamic spatial data. A feature of MOVNet is the bidirectional mapping between the two indices that enables the retrieval of a minimal set of data for query processing. Based on this dual-index structure, we proposed algorithms to efficiently execute snapshot range queries as well as snapshot kNN queries.

The continuous query is one of the most complicated query types in location-based services due to its expensive consumption of memory and computational resources. However, it provides a prolonged perspective on the change of object movements, and hence, it is well suited for monitoring purposes. In this paper, we significantly extend the functionality of MOVNet to support continuous range query processing. Specifically, we introduce the concept of *connecting vertices* in each grid cell. We demonstrate that with a precomputing step a corresponding distance table can be created in each grid cell to speed up the network distance expansion during query processing. We also present our design of a *Shortest-Distance-based tree* (SD-tree, for short) that preserves the network connectivity and distance information in continuous query processing. We propose a novel algorithm that rotates, truncates, and extends the edges of an SD-tree with regard to the query point movements. This algorithm avoids recomputing the network connectivity and distance information when the query point moves to a new position. Based on these supporting techniques, we propose an efficient method to process continuous range queries. The contributions of our work are as follows:

- We propose a precomputation step that uses our design of connecting vertices to record the distance information among these vertices to facilitate query

- *H. Wang is with the Department of Computer Science, University of Southern California, Los Angeles, CA 90089. E-mail: haojunwa@usc.edu.*
- *R. Zimmermann is with the School of Computing, Department of Computer Science, National University of Singapore, Computing 1, 13 Computing Drive, Singapore 117417. E-mail: rogerz@comp.nus.edu.sg.*

Manuscript received 5 Mar. 2009; revised 5 Oct. 2009; accepted 21 Mar. 2010; published online 7 Sept. 2010.
Recommended for acceptance by X. Zhou.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-2009-03-0110.
Digital Object Identifier no. 10.1109/TKDE.2010.171.

processing. Our simulation results show that such a technique works superbly when dealing with dense networks.

- We introduce a novel data structure, the SD-tree, to preserve the network connectivity and distance information for the duration of a query. With the SD-tree, we can often infer network connectivity and distance information even when the query point moves to a new position. We use this data structure to monitor the space affected by a query, and update the result space with regard to the query point movements. We illustrate that such a technique facilitates the continuous query processing.
- We propose a Continuous Mobile Network-Distance-based Range query algorithm (C-MNDR) and we verify the performance of this algorithm in MOVNet through vigorous experiments. We illustrate that C-MNDR is very efficient in supporting continuous range query processing with a large number of moving POIs in metro road networks.

The rest of this paper is organized as follows: The related work is described in Section 2. In Section 3, we introduce our design of connect vertex, precomputing component, the structure of SD-tree, and the details of C-MNDR. The experimental validation of our design is presented in Section 4. Finally, we discuss the conclusions and future work in Section 5.

2 Related Work

The processing of spatial queries on stationary POIs with network distances has been intensively studied in recent years. Papadias et al. [19] first presented an architecture with disk-based data storage that integrates network and euclidean distance information in processing network-based range and kNN queries. Specifically, their technique is based on the ideas of *euclidean restriction* and *network expansion*. The euclidean restriction stipulates that for two arbitrary objects in a network, the network distance is equal or greater than the euclidean distance between them. This can be utilized as a lower bound in the distance computation. In contrast, the network expansion method performs the queries incrementally from the query point by expanding the nearby vertices in the order of their distances from the query point. As an improvement, the VN^3 approach [15] was proposed that partitions a large network into a set of small Voronoi regions. The goal was to avoid online distance computation in processing kNN queries by precomputing the distances within and across Voronoi regions. Moreover, Huang et al. [11] addressed the same problem by introducing the *islands approach* that estimates the overhead of precomputation and the trade-off between query and update performance for kNN queries with varying densities of POIs and networks. Recently, Samet et al. [22] proposed a novel technique that is based on precomputing and distance encoding. Specifically, the shortest paths between all possible vertices in the network are collected and the query processing for kNN objects is simplified to a search in encoded subspace.

To cope with Continuous kNN (C-kNN) queries over stationary POIs in the network, Kolahdouzan and Shahabi [14] proposed the Intersection Examination and Upper Bound Algorithm (IE/UBA). This can be regarded as the counterpart of VN^3 in C-kNN query processing, 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 and Chung [3] solved the same problem by introducing UNICONS that incorporates the precomputed kNN lists into Dijkstra's algorithm. The simulation results showed that it outperforms the IE/UBA approach when dealing with dense networks. Additionally, Hu et al. [8] proposed a tree-based structure to record the graph topology of the road network. In Hu's approach, the traditional network expansion technique is replaced by precomputed tree path exploration to reduce the processing cost for kNN queries.

All these works hold the assumption that the POIs are static. Therefore, the idea of precomputing distances between POIs and vertices is widely used and proves to be efficient. On the other hand, these methods cannot be applied to a dynamic environment where the POIs are constantly moving.

A large number of the spatial applications require the capability to process moving POIs. This requirement raises the issue of managing location updates of moving POIs in an index structure. Although tree-based index structures (e.g., R-tree and its variants [7], [1]) have been widely used in managing stationary spatial data, they suffer from expensive node reconstruction overhead when dealing with location updates. To overcome this challenge, using the trajectory of moving POIs to presume the movement of objects has been used (e.g., the *TPR-Tree* and its variants [25], [21] and the B^x tree [13]). In general, these methods assume that the movement of POIs can be represented as a linear function of time. Changing the velocity vector of moving POIs consequently invokes an update of the movement function. As an alternative, STRIPE [20] introduces the idea of transforming the trajectories of objects in D-dimensional space into points in $2D$ 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. Recently, grid-based index structures have raised intensive interest due to their simplicity and efficiency in managing moving objects. For instance, Xiong et al. proposed LUGrid [28], an update-tolerant on-disk grid index, that outperforms the LUR-tree in terms of update and query costs. Based on this fact, most of the recent works leverage either an in-memory grid index [4], [6], [17], [29] or an on-disk grid index [27] for spatio-temporal processing. In similar spirit, MOVNet utilizes an in-memory grid index to manage the location updates of moving POIs.

There exist considerable research results addressing the issue of spatial query processing on moving POIs with euclidean distances. Most of these works rely on a grid index for maintaining position updates. 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 and kNN queries. Hu et al. [9] 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 [16] and SEA-CNN [27] were introduced as centralized solutions with the idea of *shared execution* to process continuous range and kNN queries over moving POIs. Yu et al. [29] proposed an algorithm for monitoring C-kNN queries over 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. [17] presented a solution (CPM) for C-kNN queries 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, such as to improve the system throughput. However, the above techniques only consider the euclidean distance computation, which makes them unsuitable for applications where the network-based distance computation is required.

For environments where POIs are dynamic and distances are based on network paths only, a few techniques exist. Jensen et al. [12] addressed the challenge of query processing on moving POIs in a network. Specifically, this work described an abstract infrastructure for handling location updates of moving POIs in a network and proposed a kNN query algorithm. This work is fundamentally different from MOVNet due to its system assumptions. MOVNet adopts a centralized infrastructure with periodic location updates from moving POIs, while Jensen's method assumes that the mobile client is willing to participate in the kNN query processing. As a centralized alternative, S-GRID [10] was introduced as a means to process snapshot kNN queries. A precomputed structure is maintained with regard to the spatial network data, such as to improve the efficiency of query processing. Moreover, Mouratidis et al. [18] 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 a 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 requirements. Instead, MOVNet uses an on-disk R-tree structure that has a proven performance for large size 2D data usage. Additionally, both S-GRID and IMA/GMA focus on kNN type of queries only, which inherently uses different query execution strategies. Recently, Stojanovic et al. [23] proposed an approach that monitors continuous range queries on both static and moving objects. However, the technique assumes that the query point is willing to report its destination in advance to the server; hence, the route that the query point moves on is recorded on the server and the POIs that move on the route become the candidate result set. In contrast, our technique does not require the knowledge of the query point's destination during query processing; hence, it can be applied to applications which require more general mobility scenarios.

3 System Design

We start by presenting our design of the index structures and precomputing components to support continuous query processing. After that we present our continuous range query algorithm (i.e., C-MNDR).

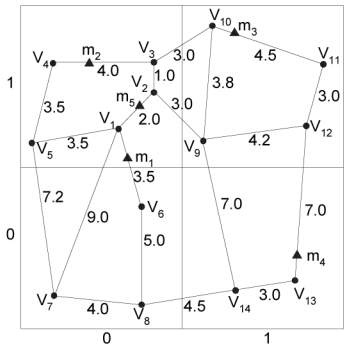

Fig. 1. An example network indexed by a 2×2 grid index.

3.1 Data Structure Design

To cope with continuous query processing, we introduce a dual-index structure, which leverages our index design of MOVNet for snapshot query processing [26]. First, an on-disk R*-tree [1] stores the stationary network data. Specifically, the edges of the network are stored as MBRs bounded by their vertices. During query processing, when the edges are retrieved from disk, a corresponding directed modeling graph is constructed in memory. Second, a memory-based grid index is used to store the locations of moving POIs. We partition the space into a regular grid of $l \times l$ cells. 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 is positioned at $loc_t(m) = (x_m, y_m)$; 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. We assume that when an object sends an updated position to MOVNet, the server invokes a map-matching procedure to locate the object on a road segment. For each element in the object array, we record the coordinates of the moving object and the edge on which the moving object is located (i.e., the starting and ending vertices of the edge). We also create a *queryPoint* flag indicating if the moving object belongs to a query point. A query point q is a moving object $\in \mathbb{M}$ issuing a location-based spatial query at any time. Currently, our design focuses on continuous range query processing. 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 unless explicitly denoted as different metrics.

We observe that, for a dense network, most of its edges are short. For each grid cell, there are only several edges that cross the boundary of the cell. Thus, those vertices connected by these edges act as the entrances and exits of the cells in the network. Based on this observation, we introduce the concept of *connecting vertices* to efficiently support query processing in dense networks. A connecting vertex has at least one outgoing edge that crosses the boundary of its enclosing cell. For example, Fig. 1 shows a dense network that overlaps with a 2×2 grid index. There

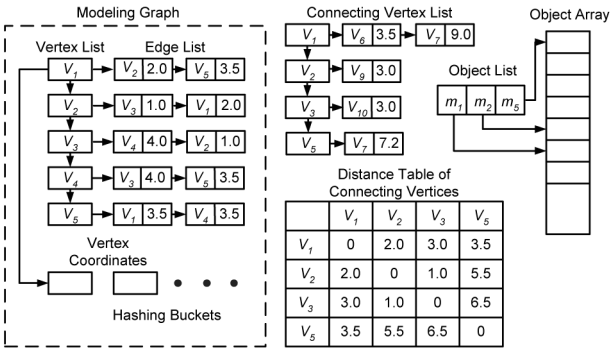

Fig. 2. An example of the data structure of C-MNDR.

are four connecting vertices in $c(0,1)$: v_1, v_2, v_3, and v_5. The outgoing edge of a connecting vertex connects with its *paired connecting vertex*. A connecting vertex has at least one corresponding paired connecting vertex. For instance, the connecting vertex v_1 has two paired connecting vertices, v_6 and v_7. As we shall also see, there are three POIs in $c(0,1)$: m_1, m_2, and m_5.

Fig. 2 shows an example of the data structures that are used in continuous query processing for $c(0,1)$ of Fig. 1. Specifically, an in-memory grid index is superimposed over the service space. Each cell has an *object list*, which records the identifiers of enclosed objects. The identifiers point to the *object array* in memory that stores the coordinates of objects. Additionally, when a cell overlaps with a query, the enclosed edges are retrieved from the R*-tree by using a stationary range query whose range is the area of the cell (e.g., $c(0,1)$). Correspondingly, a *modeling graph* is created on-the-fly and stored in memory. Note that the modeling graph only records the edges that are fully enclosed in the cell. For instance, the *edge list* of v_2 records $e(v_2, v_1)$ and $e(v_2, v_3)$. Moreover, for each cell in the grid index, we precompute the following data set: First, we record the set of the connecting vertices and their paired connecting vertices in the *connect vertex list*. Second, we create a *distance table* of the connecting vertices in each cell. It records the pair-wise distance between all connecting vertices in the same cell. For example, for connecting vertex v_1 in $c(0,1)$, the distances to other connecting vertices in the same cell are $dist(v_1, v_2) = 2.0$, $dist(v_1, v_3) = 3.0$, and $dist(v_1, v_5) = 3.5$, respectively.

The connecting vertex list together with the distance table is able to provide a more precise estimation on the edges and cells that are affected by a query. In the following section, we describe how to use our data structures to process continuous queries.

3.2 Continuous Range Query Algorithm

We now detail the efficient processing of continuous network-distance-based range queries. We first define a continuous network-distance-based range query: given a query point q, a value d, a network G, and a set of moving objects $\mathbb{M}$, a continuous network-distance-based range query retrieves all POIs of $\mathbb{M}$ that are within distance d along the edges of the network from q during a time interval $[t_1, t_2]$. The query can be represented as $continuousRangeQuery_t(q, d)$: $loc_t(q) \times loc_t(\mathbb{M}) \rightarrow \{m_i, i = 1, \ldots, n\}, \forall m_i, \ dist_t(q, m_i) \leq d$, $t \in [t_1, t_2]$. In the following sections, we start by describing our design of computing initial query results. More importantly, we introduce the concept of SD-tree to monitor the edges that are affected by a continuous query. When the query point moves, we present a novel algorithm to rotate, truncate, and expand the SD-tree to obtain the updated set of affected cells and the distances of vertices with regard to the query point movement. By using such a technique, continuous query processing can be accomplished in an incremental manner, which significantly reduces the query cost. Finally, we outline a complete procedure for continuous range query processing.

3.2.1 Initial Result Computation in C-MNDR

The first step to process a continuous range query is to obtain the initial query result set. Later, we monitor the change of POI locations to update the result set. Our design of initial range query result processing is based on a cell-based network expansion approach. Algorithm 1 details the procedure. When a moving object q submits a continuous range query request with a range constraint d, we first locate the cell where q is located (Line 2) and retrieve the edges in the cell to create a modeling graph (Line 4). After that we insert the query point as the starting point into the modeling graph (Line 5). Next, Dijkstra's algorithm is invoked to compute the distance of each vertex in the graph from the query point (Line 6). We set the range constraint d in the distance computation so that for any vertex whose distance is larger than d, the algorithm stops to expand to other vertices. We also maintain a minimum priority queue $CVToExpand$ to store connecting vertices based on their distance values from the query point (Lines 7-9). Note that if the distance of a connecting vertex is out of the query range, it will not be inserted into $CVToExpand$. Moreover, $resultCellSet$ stores the set of cells that possibly have objects in the query range.

Algorithm 1. Compute-init-cont-rangeQuery (q, d)

1: /*q is the query object */
2: /*d is the range */
3: $resultObjs = \phi$, $resultCellSet = \phi$
4: $c =$ Locate-cell(q)
5: $resultCellSet = resultCellSet \cup c$
6: $G =$ Create-graph($tree, c$)
7: Add-vertex-into-graph(G, q)
8: $S =$ Compute-distance(G, q, d)
9: **for** each connecting vertex v in S where $dist(q, v) < d$ **do**
10: $CVToExpand = CVToExpand \cup v$
11: **end for**
12: **while** $CVToExpand$!= NULL **do**
13: $v =$ De-queue($CVToExpand$)
14: **for** each paired vertex v' of v **do**
15: $resultCellSet = resultCellSet \cup cellOverlapping(e(v, v'), d - dist(q, v))$
16: **if** $dist(q, v) + length(v, v') < d$ AND $dist(q, v) + length(v, v') < dist(q, v')$ **then**
17: $c' =$ Locate-cell(v')
18: **if** G' in c' == null **then**
19: $G' =$ Create-graph($tree, c'$)
20: **end if**

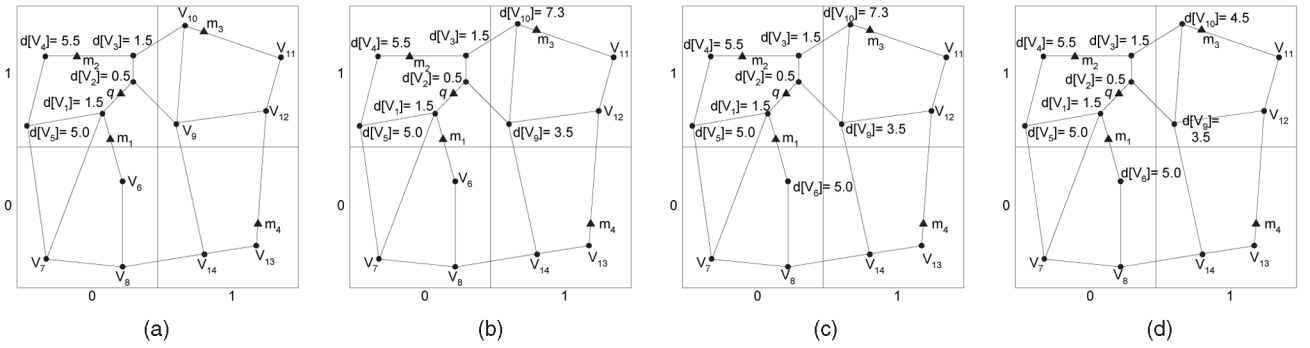

Fig. 3. The distances of vertices in an example of initial continuous range query processing.

21: $dist(q, v') = dist(q, v) + length(v, v')$
22: **for** each connecting vertex v'' in c' **do**
23: **if** $dist(q, v'') < d$ AND the paired vertex of $v'' != v$ **then**
24: $CVToExpand = CVToExpand \cup v''$
25: **end if**
26: **end for**
27: Compute-distance(G', v', $d - dist(q, v')$)
28: **end if**
29: **end for**
30: **end while**
31: $resultObjs =$ Retrieve-objects($resultCellSet$)
32: return $resultObjs$

Next, we start to expand the network into other cells via connecting vertices. We start by extracting the first connecting vertex v from $CVToExpand$ (Line 11). For each outgoing edge of the connecting vertex, we compute the overlapping cells (Line 13). After that the paired vertex v' of v is selected and the overlapping cell c' for v' is located. Next, we use the distance table to determine the distances of other connecting vertices in c'. When a connecting vertex is in the query range, it will be appended to $CVToExpand$ (Line 22).

The algorithm continues to expand the network and compute the distances of vertices via connecting vertices until $CVToExpand$ becomes empty (Line 10). At that moment, we have discovered all the cells that are affected by the range query. The next phase is to retrieve moving objects in $resultCellSet$ from the grid index to constitute the result set (Line 29). Based on the distance information of vertices that we collected in previous steps, we can compute the distance of these objects and insert the ones that are within the range d into $resultObjs$.

To illustrate the algorithm with an example, let us assume that the system is processing a network as shown in Fig. 1. A moving object m_5 with $dist(m_5, v_2) = 0.5$ submits a continuous range query where the range $d = 7.5$. After we finish executing Line 9 in Algorithm 1, the distance of each vertex in $c(0,1)$ is shown in Fig. 3a. Additionally, $CVToExpand$ contains $\langle (v_2 = 0.5), (v_1 = 1.5), (v_3 = 1.5), (v_5 = 5) \rangle$. Next, v_2 is de-queued from $CVToExpand$. It has a paired connecting vertex v_9, which is located in $c(1,1)$. We insert $c(1,1)$ into $resultCellSet$. With the distance information stored in the connecting vertex list, we determine that $dist(q, v_9) = 3.5 < d$. Therefore, we retrieve the edges in $c(1,1)$ and create the corresponding modeling graph in $c(1,1)$. Additionally, the connecting vertex list of $c(1,1)$ indicates that the connecting vertices in $c(1,1)$ are v_9, v_{10}, and v_{12}. Based on the values stored in the distance table of $c(1,1)$, we are able to conclude that $dist(q, v_{10}) = 7.3$ and $dist(q, v_{12}) = 7.7$ (which is out of range). We set the condition in Line 21 in Algorithm 1 to avoid the expansion to loop; hence, our expansion on v_9 will not move back to v_2. Additionally, v_9 also connects with v_{14}, which is on a path that leads to $c(1,0)$. Consequently, we insert v_9 and v_{10} into $CVToExpand$. Now, $CVToExpand$ enqueues the following items: $\langle (v_1 = 1.5), (v_3 = 1.5), (v_9 = 3.5), (v_5 = 5), (v_{10} = 7.3) \rangle$. The distance of each vertex within the query range in $c(1,1)$ is shown in Fig. 3b.

Next, we de-queue v_1 from $CVToExpand$. The paired connecting vertex v_7 is out of range while v_6 has $dist(q, v_6) = 5.0$. Therefore, we insert $c(0,0)$ into $resultCellSet$. On the other side, other connecting vertices in $c(0,0)$ are out of range based on the values in the distance table. Therefore, no connecting vertex is inserted into $CVToExpand$. When the expansion finishes with $c(0,0)$, the distance of each vertex within the query range is shown in Fig. 3c.

Now, we de-queue v_3 from $CVToExpand$. The paired connecting vertex is v_{10}. Although v_{10} was expanded and we recorded $dist(q, v_{10}) = 7.3$, there exists a shorter route from q via v_3 where $dist(q, v_{10}) = 4.5$. This case is recognized by our algorithm in Line 14. We update the distance of v_{10}. Additionally, following the path from v_{10}, other connecting vertices in $c(1,1)$ have $dist(q, v_9) = 8.3$ and $dist(q, v_{12}) = 12.0$, which are both out of range. Therefore, no connecting vertex is inserted into $CVToExpand$. Next, the distance of each vertex in $c(1,1)$ is computed again. However, when the distance computation finishes, no other vertex distances in $c(1,1)$ are shortened. The distance of vertices in $c(1,1)$ is presented in Fig. 3d.

At this moment, there are three connecting vertices left in $CVToExpand : \langle (v_9 = 3.5), (v_{10} = 4.5), (v_5 = 5) \rangle$. We de-queue v_9 first. It has two paired vertices, v_2 and v_{14}. Since $dist(q, v_2) = 0.5$ is already recorded and it is shorter than the path via v_9, we do not expand on v_9 at this moment. Additionally, v_{14} is out of range. We only insert $c(1,0)$ into $resultCellSet$. Next, v_{10} is de-queued and its paired vertex is v_3. However, $dist(q, v_3) = 1.5$ is already recorded, which indicates that we have already found a shorter route. Finally, we de-queue v_5. The paired vertex is v_7 and it is out of range;

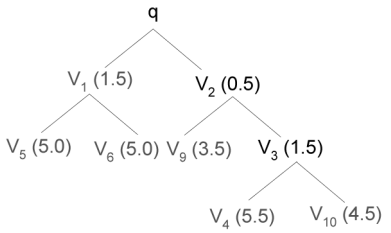

Fig. 4. An example of the SD-tree.

hence, there is no expansion needed. Since there is no vertex left in $CVToExpand$, we finish searching the affected cells with $resultCellSet = \langle c(0,1), c(1,1), c(0,0), c(1,0)\rangle$.

Finally, the moving objects in $resultCellSet$ are retrieved and the distance from q is computed based on the distances of the starting and ending vertices of the edge where the object is located (Line 29) to constitute the result set. In this example, $resultObjs$ contains three objects when the algorithm terminates: m_1, m_2, and m_3.

3.2.2 Monitoring Object Updates in C-MNDR

The initial query result is only valid for the first update cycle. We need to continuously monitor the change of the query result. However, the initial query result processing creates several data sets that are useful for subsequent processing steps. First, $resultCellSet$ represents a snapshot of the area in the service space that is affected by the query. Second, the cell-based network expansion procedure computes the distances of vertices that are within the query range. Based on this information, we introduce the concept of a *SD-tree* to facilitate the query update processing.

Definition 1. *A Shortest-Distance-based Tree is a tree structure that consists of a set of vertices that are within the query range d from the location of the current query point q. The query point is the root element in the tree. Branches from the query point to a specific vertex represent the shortest path between them.*

For instance, after we finish computing the distances of vertices in Fig. 3d, we construct a SD-tree for the query as shown in Fig. 4. Specifically, starting from the query point, we record every vertex that is within the query range in the SD-tree. As we can see from the SD-tree, $dist(q, v_{10}) = 4.5$, and the shortest path is $q \rightarrow v_2 \rightarrow v_3 \rightarrow v_{10}$. Note that the SD-tree does not record every edge as a branch whose starting and ending vertices fall both within the range, such as $e(v_9, v_{10})$, because these edges do not belong to any shortest path. Additionally, we define two types of vertices in the SD-tree.

Definition 2. *A **complete** vertex in the SD-tree indicates that each edge in the network starting from the vertex is recorded in the SD-tree. In contrast, a vertex is called **partial** in an SD-tree if some of the edges starting from the vertex are not recorded.*

For example, v_2 is a complete vertex, which is shown in black in Fig. 4. On the other hand, v_5 is connected with v_1, v_4, and v_7. Since v_7 is out of range, it is not recorded in the SD-tree. Additionally, v_5 and v_4 are not connected by a branch in the SD-tree because each of them follows a different shortest path from q. Therefore, v_5 is a partial vertex, which is shown in gray (red) in Fig. 4.

Furthermore, for an object m that is located on an edge $e(v_1, v_2)$, if either v_1 or v_2 is a vertex in the SD-tree, then m may be in the result set. In other words, the set of partial vertices, complete vertices, and the SD-tree branches constitute a monitoring area centered at the query point. During each query update cycle, moving objects that either enter or leave this area should be updated in the result set. Since MOVNet uses periodic sampling on moving object positions, there are a number of object updates received and stored in an object update buffer during each cycle. At the beginning of each new cycle, MOVNet invokes a procedure to process these object updates. Specifically, we distinguish two types of the objects updates: query point updates and nonquery-point updates.

Let us assume that the query point experiences no update during a cycle. In this case, the monitoring area of the SD-tree remains the same. Hence, our goal is to observe how the objects in $resultCellSet$ change. Specifically, there are two cases that affect the result set: First, an object m moves onto an edge whose starting or ending vertex is in the SD-tree, and hence, m might be in the result set. We need to compute the distance of m to the query point to confirm whether its distance is within the range. Second, if an object $m \in resultObjs$ moves to an edge whose starting and ending vertices are not in the SD-tree, m will be removed from the result set.

We now consider the case when the query point updates its location during a cycle. If the query point moves to a position that is still within the monitoring area of the SD-tree, we are able to update the connectivity and distance information of some vertices based on the SD-tree. On the other hand, if the query point moves out of the SD-tree monitoring area, we will have to invoke Algorithm 1 again to obtain the query result. For instance, if the query point in Fig. 3d moves to $e(v_{11}, v_{12})$ within one update interval, there is no information we can use directly from the SD-tree. Hence, a new range query will be issued. Note that this is unlikely to happen, unless the query object moves very fast.

If the query point moves to a new position that is within the range from the original location, there are two cases: First, the query point moves to an edge that is recorded as a branch in the SD-tree. Second, the query point moves to an edge that is not recorded as a branch in the SD-tree; however, both the starting and ending vertices of the edge are recorded in the SD-tree. We will show in detail that the second case only requires a few more steps than the first case to update the distance information of vertices.

Specifically, when the query point moves to an edge that is recorded in the SD-tree, there are three steps in updating the SD-tree.

- **Step 1. Tree rotation and distance update.** We move the query point to the branch where it is now located. We rotate the tree to place the query point to be the root element in the SD-tree. After that we update the distances of vertices in the SD-tree based on the updated branch structure.
- **Step 2. Tree truncation.** We remove vertices and their children that become out-of-range after the distance updates. We mark their parent vertices as partial vertices.

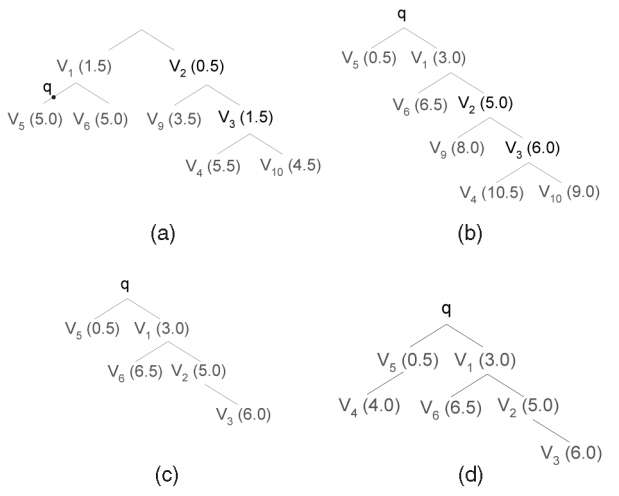

Fig. 5. The update of SD-tree when the query point moves to an edge that is recorded in the original SD-tree.

- **Step 3. Tree expansion.** For a partial vertex, if its updated distance is shortened, some of its child vertices might be able to find a shorter path. Therefore, we need to expand on these shortened-distance partial vertices. Moreover, if the updated distance for a partial vertex becomes longer, we do not need to expand the network from the vertex. This is because that during the processing in previous steps (e.g., the initial query result processing), we have already ensured that the routes that are not recorded from the partial vertex are either out of range or nonoptimal. Additionally, a complete vertex does not require any expansion because its connectivity is completely recorded in the SD-tree.

Let us assume that q in Fig. 4 moves to $e(v_5, v_1)$ where $dist(v_5, q) = 0.5$ as shown in Fig. 5a. We first process **Step 1** to place q to be the root element again. We also recompute the distance for each vertex correspondingly as displayed in Fig. 5b. Next, we execute **Step 2** and remove v_9, v_4, and v_{10} that are out of range. Their parent vertices also become partial vertices (Fig. 5c). Note that when a vertex is removed, the overlapping cells of its outgoing edges will be removed from $resultCellSet$, except for the one that connects with its parent vertex in the SD-tree. Finally, we execute **Step 3**. Only the distance of the partial vertex v_5 is shortened from 5.0 to 0.5 when we compare the distance information to earlier data as shown in Fig. 4. Therefore, we need to invoke the network expansion from v_5 with an initial distance of 0.5. In case v_5 has any children, we remove these child vertices before the expansion. This step can be regarded as a similar expansion algorithm as we described in Section 3.2.1. Once the expansion finishes, the SD-tree has been transformed into the one shown in Fig. 5d. Specifically, we found a shortest path $q \rightarrow v_5 \rightarrow v_4$ where $dist(q, v_4) = 4.0$. Additionally, $resultCellSet = \langle c(0,1), c(1,1), c(0,0)\rangle$. This example shows that the SD-tree helps to reconstruct most of the connectivity and distance information when the query point moves; hence, it significantly improves the query processing efficiency.

Lemma 1. *Let m be a moving object on an edge whose starting vertex is recorded in the SD-tree. After SD-tree update, $dist(q, m)$ in the SD-tree still records the shortest path from q to m.*

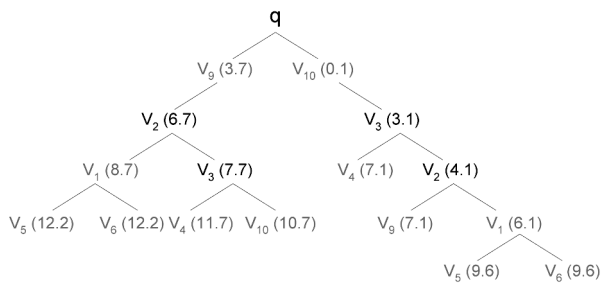

Fig. 6. The transformation of the SD-tree when q moves to an edge that is not recorded in the original SD-tree.

Proof. Assume that q is updated to be on an edge $e(v_1, v_2)$. From the SD-tree update procedure, we know that if there is a partial vertex v on the path of $q \rightarrow v_1 \rightarrow \cdots v \rightarrow \cdots \rightarrow m$ and $dist(q, v)$ gets shortened, then a tree expansion will be invoked on v. Let v be the first partial vertex that gets shortened on the path of $q \rightarrow v_1 \rightarrow \cdots v \rightarrow \cdots \rightarrow m$ (i.e., $dist(q, v_1)$ becomes longer), the updated path from v to m can be guaranteed to be the shortest in the tree expansion. Based on triangle inequality, if there exists a shorter path of from q to m, it implies that either there exists a shorter path of $q \rightarrow v_2 \rightarrow \cdots \rightarrow m$ or a shorter path from q to v. For the first case, we know that $dist(q, v_2)$ becomes shorter during an update; hence, a network expansion will be invoked on v_2, which is guaranteed to find such a shortest path. For the second case, it implies that there is a partial vertex whose distance becomes shorter during an update. However, this contradicts the fact that v is the first partial vertex gets shortened on the path of $q \rightarrow v_1 \rightarrow \cdots v \rightarrow \cdots \rightarrow m$, hence such a path does not exist. □

If the query point moves to an edge that is not recorded as a branch in the SD-tree (e.g., q moves to $e(v_9, v_{10})$ where $dist(v_{10}, q) = 0.1$), we are not able to execute **Step 1** directly. Recall that a dense network is highly connected. From our experiments, we found that, for example, the SD-tree records about 50 percent of the edges that are within the range for the data set of networks in Los Angeles County (LA). It is highly desirable to have an algorithm to support processing query point updates in this case.

We observe that for a query point q located on $e(v_9, v_{10})$, the shortest path of a vertex is via either v_9 or v_{10}. Therefore, we first use v_9 as the root element to rotate the SD-tree. Next, we insert the updated SD-tree as a child node of q connected by $e(q, v_9)$ (i.e., the subtree starting from the left child of q in Fig. 6). After that we use v_{10} as the root element to rotate the original SD-tree. The rotated SD-tree is added as another subtree starting as the right child node of q connected by $e(q, v_{10})$ in Fig. 6.

Fig. 6 shows that, for each vertex, there are two updated paths from q via the starting and ending vertex of the edge on which q is located, respectively. To obtain the shorter path for each vertex, we invoke a breadth first tree traversal. Let us start with v_9 in this example. There are two paths, $q \rightarrow v_9$ and $q \rightarrow v_{10} \rightarrow v_3 \rightarrow v_2 \rightarrow v_9$, that are recorded in the

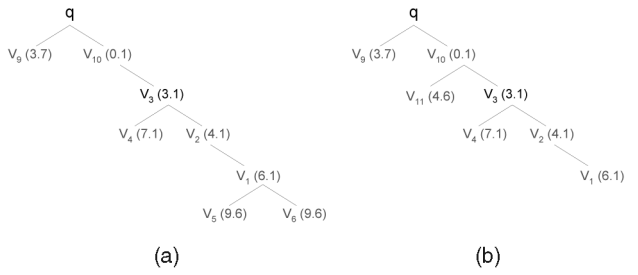

Fig. 7. The update of SD-tree when the query point moves to an edge that is not recorded in the original SD-tree. (a) Tree traversal to obtain the shortest path for vertices in the SD-tree. (b) The updated SD-tree with valid shortest distances on the vertices.

transformed SD-tree. Since the first path is shorter, we delete v_9 as the child of v_2. Additionally, we set v_2 to be a partial vertex. Next, v_{10} is examined. As we shall see, the path $q \rightarrow v_{10}$ has a shorter distance than the one of $q \rightarrow v_9 \rightarrow v_2 \rightarrow v_3 \rightarrow v_{10}$. Therefore, we keep the first path. When we examine v_2, we find that the path $q \rightarrow v_{10} \rightarrow v_3 \rightarrow v_2$ is shorter than the one of $q \rightarrow v_9 \rightarrow v_2$. Consequently, we remove v_2 as the child of v_9 as well as all children of v_2. The result of the removal operation is shown in Fig. 7a. As we shall see, the advantage of using a breadth first tree traversal is that we are able to find a duplicate vertex that has a longer distance at a higher tree level. By deleting the vertex and its children, we avoid exploring more vertices during the remaining tree traversal.

The traversal continues to process other vertices in the SD-tree. Since each vertex has only one instance in Fig. 7a, there is no operation during the remaining tree traversal. After that we can directly execute **Steps 2** and **3**, and the result is shown in Fig. 7b.

Algorithm 2 summarizes the procedure of maintaining the SD-tree with regard to query point updates.

Algorithm 2. Update-SD-tree(q, d)
1: **if** q moves to an edge $e(v_1, v_2)$that is not recorded *SD-tree* **then**
2: $subTree1$ = Rotate-tree(*SD-tree*, v_1)
3: $subTree2$ = Rotate-tree(*SD-tree*, v_2)
4: *SD-tree* $= \phi$
5: *SD-tree* = Add-child($subTree1$)
6: *SD-tree* = Add-child($subTree2$)
7: Determine-shortest-paths(*SD-tree*)
8: **else**
9: *SD-tree* = Rotate-tree(*SD-tree*, q)
10: **end if**
11: Remove-out-of-range-vertices(*SD-tree*)
12: **for** each vertex v whose distance is shortened **do**
13: Delete-child-vertices(v)
14: Expand-vertex(v, d)
15: **end for**

From an analytical perspective, the SD-tree is an unbalanced binary tree structure. When the query point requires no updates during the update cycle, then there is no cost associated with the SD-tree maintenance. Moreover, if the query point moves to a new location outside of the SD-tree monitoring area, then the SD-tree will have to be reconstructed at an additional cost of $O(n \log n)$. However, if the query point moves to a new location that is within range from the original location—in both cases, when the point moves to an edge that is recorded as a branch in the SD-tree and when it is not recorded—then the costs are the same. This is because in the latter case, MOVNet performs two tree rotations and a tree traversal before executing Steps 2 and 3. The complexity of the tree rotations is $O(\log n)$. Next, we update the distances of vertices in the rotated SD-tree, which requires $O(n)$ operations [24]. The following tree traversal step also costs $O(n)$. In contrast, if the query point moves to an edge that is recorded as a branch in the SD-tree, we only perform one tree rotation operation and update the distances of vertices afterward. This results in a total cost of also $O(n)$. At this point, both cases execute Steps 2 and 3 in our algorithm, which are eventually bounded by the cost of tree expansion (i.e., Dijkstra's algorithm).

3.2.3 Overview of the Continuous Range Query Processing

So far we presented the issues and solutions for initial query result processing, creating the SD-tree, dealing with object updates, and maintaining the SD-tree with regard to query point updates. In this section, we combine all the components and describe the complete procedure of C-MNDR to process a continuous network-distance-based range query (shown in Algorithm 3).

Algorithm 3. C-MNDR(q, d)
1: /*q is the query object */
2: /*d is the range */
3: Compute-init-cont-rangeQuery (q, d)
4: Build-SD-tree(q)
5: **for** each update cycle **do**
6: **if** query point position is updated **then**
7: **if** query point moves out of SD-tree **then**
8: Compute-init-cont-rangeQuery (q, d)
9: Build-SD-tree(q)
10: continue
11: **else**
12: $resultObjs = \phi$, $resultCellSet = \phi$
13: Update-SD-tree(q, d)
14: $resultObjs$ = Retrieve-objects($resultCellSet$)
15: **end if**
16: **else**
17: Update-CRange-resultSet()
18: **end if**
19: **end for**

When MOVNet receives a request from q for a continuous range query, it launches the initial query result processing (Line 3) as described in Section 3.2.1. Once the processing is finished, the corresponding SD-tree is created based on the connectivity and distance data (Line 4). Next, at the beginning of each update cycle, MOVNet first examines if the query point submits an update. If the query point moves to a position that is out of the SD-tree monitoring area, MOVNet again invokes the initial query result processing and constructs a new SD-tree afterward (Lines 7-11). On the other hand, if the query point moves to a position that is in the area enclosed by the SD-tree, we

utilize the current SD-tree to expedite the query processing (Lines 12-14). In case that the query point has no update, C-MNDR only monitors the change of objects in $resultCellSet$ to keep the result set current.

In summary, MOVNet processes continuous queries by using the connecting vertices to determine the set of cells and vertices that overlap with the query. MOVNet also uses the SD-tree to monitor the changes of the query along the time dimension. We have presented the algorithm to rotate, truncate, and extend the edges in the SD-tree with regard to object updates. Our simulation results indicate that the system performance of C-MNDR is much more efficient than executing the snapshot-based query processing at the beginning of each update cycle. Extensive results will be presented in Section 4.2.

4 Experimental Evaluation

In Section 4.1, we start by describing the data set used in our simulation and our simulator implementation. After that we discuss the performance of continuous query processing with MOVNet in Section 4.2.

4.1 Simulator Implementation

We obtained a real data set from TIGER/Line.[1] The LA County 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 bidirectional. Additionally, we used a network simulator [2] to generate the positions of 100,000 moving objects in the road network. The simulator assumes uniform distribution of the objects at initial time. After that each object follows the random walk model [5] with a maximum speed limit to move in the service space. With the random walk model, a node's movement is divided into a sequence of intervals called mobility epochs. Each epoch is a random variable representing a duration which is exponentially distributed. During each mobility epoch, a node moves with a constant speed and direction. When a node arrives at its destination, a new speed and direction are chosen. A node's velocity is a random variable uniformly distributed between $[0, v_{max}]$ and the direction is uniformly distributed over $[0, 2\pi]$. At each time stamp, a user-defined number of moving objects report their updated locations.

In our study of continuous range query processing, we adopted two algorithms to compare with C-MNDR. First, we leveraged our design of MNDR in [26]. Since MNDR mainly focuses on snapshot query processing, a repeated snapshot processing method (RS-MNDR) was used to accommodate the continuous query processing: at the beginning of each update cycle, we invoked MNDR from the query point as if to process a new snapshot range query. Second, we designed a baseline continuous range query processing algorithm (C-MNDR-BASE). It utilizes the same data structures as C-MNDR. At the beginning of each update cycle, if the query point moved to a new location, the baseline algorithm invoked a new initial query result processing step. Thus, the baseline algorithm did not use the SD-tree to manage the network connectivity with regard to the updated query point location. We studied the simulation results between the baseline algorithm and C-MNDR to obtain the performance improvement produced by the SD-tree during query processing.

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

TABLE 1
Continuous Query Processing Simulation Parameters

Parameter	Default	Value Range
Number of POIs	50K	20K - 100K
POI distribution	Uniform	Uniform
Radius (miles)	3	0.1 - 5
Number of cells per axis	400	100 - 500
Number of queries	2K	500 - 3K
Percent of updated objects	5	2 - 100
Update cycles	20	20
Object speed	0.4	0.4 - 2

We implemented a simulator in Java. The LA county data set was arranged into an R^*-tree index file in which we set the page size to 4 KB. The high-level functionality of our simulator is as follows: 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 uses a buffer for caching the disk pages read by MOVNet with a size of 10 pages. Next, a grid index is created in memory. At the beginning of each test, our simulator acquires the positions of objects from a text file that records the coordinates. It also reads from another file that contains the records of object updates of each cycle and stores them in a list in memory. The simulator then executes object updates at the beginning of each time stamp.

4.2 Continuous Query Processing Simulation Results

In this section, we present and discuss the simulation results of continuous query processing. Table 1 summarizes the parameters used in our simulations. For each experimental setting, we varied a single parameter and kept the remaining ones at their default values. We assume that objects are moving in the area of the LA county. The average speed for a moving object during each update cycle is 0.4 kilometers. For each continuous query, the simulator randomly picks a moving object and launches a query from its location. We monitored the change of objects for 20 update cycles. The simulator output the initial query result as well as the updated result set after each time stamp. The experiments measured the CPU time and the number of disk page accesses as the performance metrics of the query processing. For each experimental configuration, the simulator executed 50 iterations and reported the average result. The simulation was executed on a Linux server with 16 GB memory and a 3.0 GHz Xeon processor.

For a continuous query, the total cost consists of the object update cost (i.e., updating the locations of objects in the grid index), the initial query result processing cost, and the query update cost. Note that the CPU time for MOVNet to process all continuous queries should be less than one update cycle to ensure the correctness of the query results. Otherwise, the query result would become invalid before the system finishes processing during each update cycle.

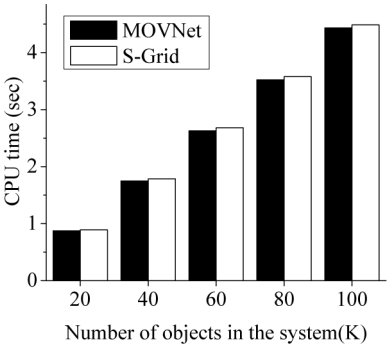

Fig. 8. The CPU time of the object update cost in MOVNet.

4.2.1 Object Update Cost in MOVNet

We first verified the object update costs in MOVNet compared to S-GRID [10]. To achieve a fair comparison, we implemented the *Vertex-Edge* component of S-GRID as an on-disk module. Edges of the network are indexed by an R^*-tree. Additionally, the precomputed results (e.g., the Cell-Border and Vertex-Border components) in S-GRID are stored in memory. We assume that at the beginning of each update period, 10 percent of the POIs submit their new positions. Fig. 8 shows that when there are 10,000 updates messages in one period, MOVNet is able to record these changes in about 4.5 seconds. Furthermore, MOVNet requires slightly less CPU time than that of S-GRID. Although both techniques include the map-matching procedure in the object update to record the edge where the object is located, S-GRID records an object in a cell if its nearest vertex on edge e belongs to this cell. Therefore, distance computation is performed during an object update in S-GRID. In contrast, MOVNet directly inserts the object into the cell which encloses it and hence simplifies the update procedure.

4.2.2 Connecting Vertex Distribution in MOVNet

Next, we focus our interests on studying the relationship between the number of connecting vertices and the cell size. Fig. 9a illustrates the statistics of the connecting vertex distribution. First, the number of connecting vertices grows with an increasing number of cells. Additionally, when MOVNet utilizes 500×500 cells, there are over 100K connecting vertices, which is more than three times the number when using 100×100 cells. In contrast, the number of connecting vertices is about 130K when MOVNet has $1{,}000 \times 1{,}000$ cells. This demonstrates that for relatively small cell sizes, the growth rate of the number of connecting vertices levels off. As we shall see in the following section, a very large number of connecting vertices can become the bottleneck in the system performance. In Fig. 9b, we illustrate the density of connecting vertices in MOVNet, which refers to the average number of connecting vertices in a cell. The results indicate that the density of the connecting vertices becomes less as the number of cells in the system increases.

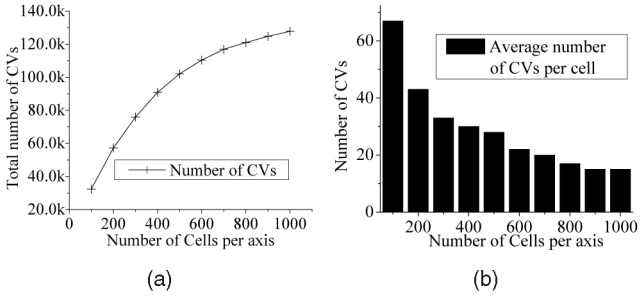

Fig. 9. The distribution of connecting vertices as a function of the number of cells. (a) Number of CVs in MOVNet. (b) Density of CVs in MOVNet.

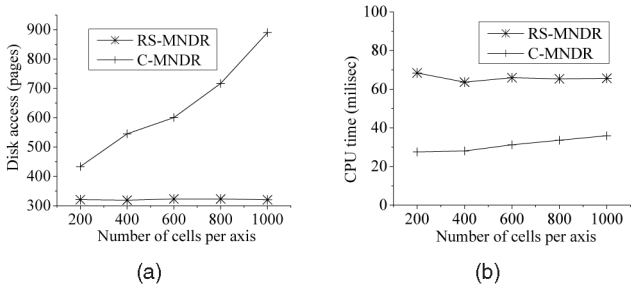

Fig. 10. The performance of initial query result processing in C-MNDR as a function of the number of cells. (a) CPU cost. (b) I/O cost.

4.2.3 Performance Study of C-MNDR

Initial query result processing. We verified the performance of the initial query result processing. Fig. 10a illustrates the effect of the number of cells. The results show that C-MNDR requires about half of the CPU time compared with the RS-MNDR algorithm. This is because with the help of distance tables, C-MNDR improves the efficiency in finding the set of network data in the range before retrieving objects from the grid index. Additionally, the CPU cost of C-MNDR increases with the increase of the number of cells. This suggests that with a larger number of cells, the number of connecting vertices becomes larger, which also increases the system cost. In contrast, Fig. 10b illustrates the page accesses of both algorithms. As we can see, the C-MNDR algorithm consumes more pages than RS-MNDR with various cell sizes. This can be explained by the fact that RS-MNDR uses the euclidean distance restriction as the first step when retrieving network data. Although this is a preliminary estimation in terms of the network that is within the range, it minimizes the I/O cost by performing just one range query. On the other hand, C-MNDR uses cell-based network retrieval, which results in a significant number of I/O operations, especially when there are a large number of cells in MOVNet.

Next, Fig. 11a illustrates the effect of the number of POIs. As we can see, C-MNDR consumes less than 50 percent of the CPU time compared with RS-MNDR with various numbers of POIs. Additionally, the output shows that the CPU time increases linearly with the number of POIs. The very small gradients of the C-MNDR as well as RS-MNDR output suggest that MOVNet is very scalable to support a very large number of POIs. Fig. 11b plots the disk accesses of both algorithms. The I/O cost of C-MNDR is about three times that of the RS-MNDR algorithm. Moreover, the I/O cost

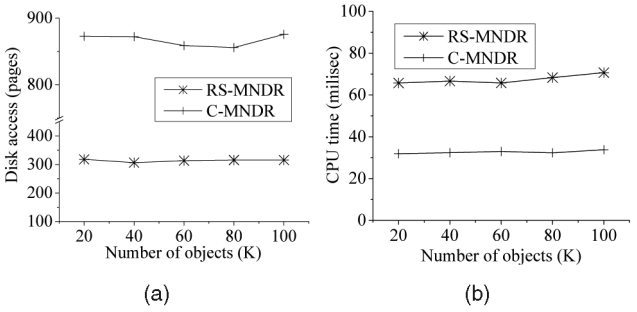

Fig. 11. The performance of initial query result processing in C-MNDR as a function of POIs. (a) CPU cost. (b) I/O cost.

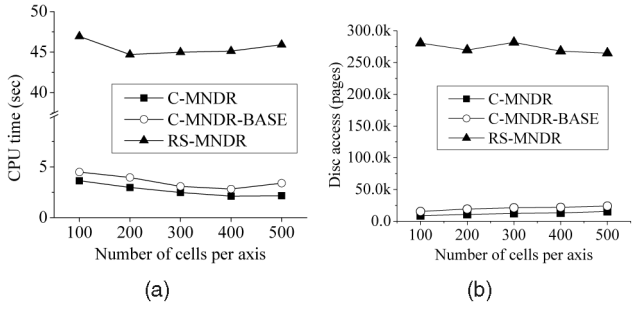

Fig. 13. The cost of query updates in C-MNDR as a function of the number of cells. (a) CPU cost. (b) I/O cost.

remains stable when varying the number of POIs in MOVNet. This is because POIs are managed by the in-memory grid index; hence, it has no effect on I/O operations.

Fig. 12a plots the CPU time of C-MNDR while changing the range. The CPU time quadratically increases with a larger range. When the range is 4 miles, C-MNDR costs 0.025 seconds. Processing a range of 10 miles requires 0.095 seconds by using C-MNDR compared with 0.257 seconds when using the RS-MNDR algorithm. Additionally, when the range is small, the CPU cost of C-MNDR and RS-MNDR is almost the same. With a larger range, C-MNDR becomes much faster than RS-MNDR, which indicates the advantage of using the connecting vertices and corresponding distance tables. Fig. 12b plots the corresponding page accesses. In contrast to the CPU output, C-MNDR requires more I/O operations than RS-MNDR. For instance, when the range is 10 miles, RS-MNDR consumes less than 1,100 page accesses while C-MNDR needs more than 2,200 page accesses. Moreover, the I/O cost grows quadratically in both algorithms, which shows the same characteristics as the CPU time.

In summary, we conclude that with the help of connecting vertices, the CPU cost of C-MNDR in initial query result processing is lower than that of RS-MNDR. On the other hand, RS-MNDR has the advantage of using the euclidean distance restriction to minimize the I/O cost.

Continuous query result processing. We now study the query update cost of C-MNDR. Figs. 13a and 13b show that the CPU and I/O costs of C-MNDR with regard to the number of cells, respectively. C-MNDR and the baseline algorithm consume as about 10 percent of the CPU time of RS-MNDR. Moreover, C-MNDR consistently requires only about 70 percent CPU time compared with the baseline algorithm. For 2,000 continuous queries in a service space covered by 400×400 cells, the query update cost is less than 3.5 seconds with C-MNDR. Finally, the savings in I/O cost are tremendous as C-MNDR only requires about 5 percent the cost of RS-MNDR. Although RS-MNDR consumes much fewer page accesses during the initial query result processing, the update cost is much higher than with C-MNDR. As a continuous query runs longer and longer, we shall see that the cumulative I/O cost of C-MNDR will become much lower than that of RS-MNDR.

Next, Fig. 14a demonstrates the CPU cost of C-MNDR compared with RS-MNDR as a function of the number of POIs. All algorithms incur larger costs as the number of POIs increases, which is caused by the retrieval of more objects from the grid index. The small gradient of these curves shows that MOVNet scales very well with regards to the growth of POIs. We attribute this feature to our design of using an in-memory grid index to manage the POIs. Additionally, C-MNDR consumes less than 10 percent of the CPU time of RS-MNDR. Compared with the baseline algorithm, C-MNDR on average saves 33 percent of the CPU time, which shows the benefits of using the SD-tree during query processing. Finally, the I/O costs of C-MNDR and the baseline algorithm are both only about 5 percent of that of RS-MNDR (Fig. 14b), which indicates a significantly improved system throughput when an incremental approach is used to process continuous queries.

Fig. 15 illustrates the effect of the range on query update processing. With a range of 5 miles, RS-MNDR requires over 100 seconds to process 2,000 queries in each update cycle.

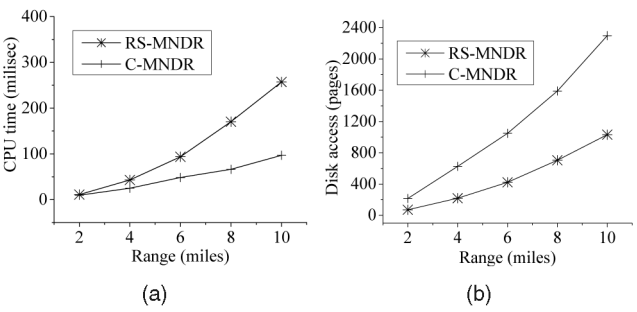

Fig. 12. The performance of initial query result processing in C-MNDR as a function of query range. (a) CPU cost. (b) I/O cost.

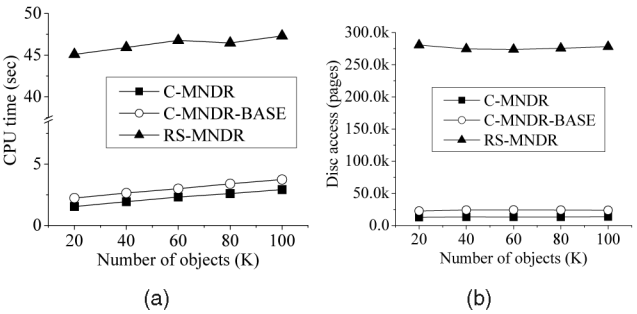

Fig. 14. The performance of query updates in C-MNDR as a function of the number of POIs. (a) CPU cost. (b) I/O cost.

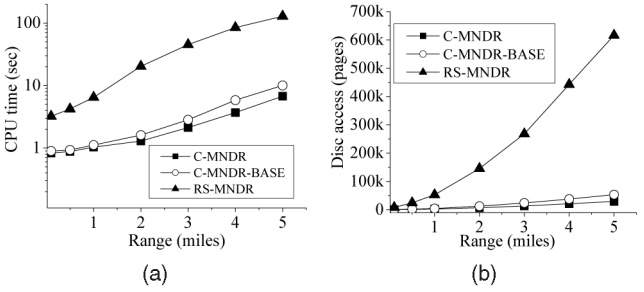

Fig. 15. The performance of query updates in C-MNDR as a function of range. (a) CPU cost. (b) I/O cost.

This might be unacceptable in many application scenarios. In contrast, C-MNDR necessitates only 6.7 seconds and the baseline algorithm consumes 10 seconds. Correspondingly, the I/O cost of C-MNDR is about 5 percent of that of RS-MNDR with a 5 mile range. The results demonstrate that our design is well suited for large range queries. Additionally, C-MNDR saves about 25 percent of the CPU time and 40 percent of the page accesses as compared with C-MNDR-BASE. Clearly, such an improvement reflects the benefits of adopting the SD-tree in query processing.

Fig. 16 plots the effect of the number of queries on the query updates of continuous range query processing. The CPU cost and the I/O cost grow proportionally with an increasing number of queries in MOVNet. The small gradient of C-MNDR compared with RS-MNDR for both CPU and I/O costs indicates that C-MNDR is able to support a very large number of queries at the same time. The more queries exist in MOVNet, the higher a performance improvement can be achieved by using C-MNDR to process continuous queries. Specifically, C-MNDR can support 3,000 queries in 3.6 seconds and 19K page accesses during each cycle when processing the object updates to obtain the updated result set. In contrast, RS-MNDR requires 69 seconds and over 400K page accesses. Additionally, C-MNDR saves about 35 percent of the CPU time as well as 45 percent of the page accesses compared with the baseline algorithm with 3,000 queries.

We also demonstrate the system performance with various update rates (from 10 percent up to 100 percent) of the moving objects in Figs. 17a and 17b. In RS-MNDR, every query needs to refresh the query result by invoking the complete procedure of MNDR. Therefore, the update

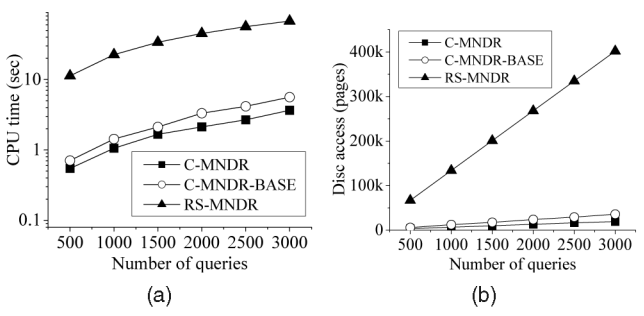

Fig. 16. The performance of query updates in C-MNDR as a function of number of queries. (a) CPU cost. (b) I/O cost.

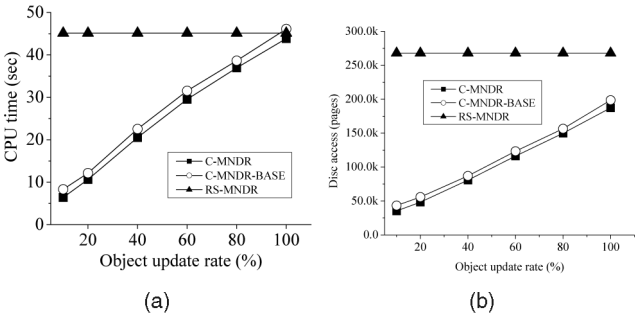

Fig. 17. The performance of query updates in C-MNDR as a function of the percentage of object updates. (a) CPU cost. (b) I/O cost.

rate of the moving objects does not affect the performance. On the other hand, with more objects reporting their location updates in one update cycle, C-MNDR requires more CPU time and page accesses. Specifically, the relationship between these two factors is linear. As is shown, when the object updates in the system reach 100 percent, the CPU cost of the MNDR baseline algorithm is even greater than that of RS-MNDR. This is because under such a setting, the baseline algorithm requires not only the initial query processing step, but also the SD-tree creation step. Hence, the baseline algorithm consumes more CPU time than RS-MNDR. Moreover, the CPU time of C-MNDR is only about 70 percent of the value of the baseline algorithm.

In Figs. 18a and 18b, we study the impact of object speed and observe the system throughput. We changed the average object speed in the service space from 0.4 miles per time stamp to 2 miles per time stamp. As shown, both the CPU and I/O costs increase gradually as the average speed steps up. Additionally, when the average speed of moving objects reaches 2 miles per time stamp, C-MNDR only consumes about 10 percent of the CPU time and disk accesses of RS-MNDR.

To summarize the performance of C-MNDR, we executed an sample set of 2,000 continuous range queries running over 10 update cycles in Fig. 19. The CPU and I/O costs at time stamp 0 in Fig. 19 represent the initial query result processing. As we can see, C-MNDR requires less CPU time but more page accesses during this step. However, after one update cycle, the total cost of CPU time and page accesses of C-MNDR becomes less than that of RS-MNDR. This is due to our incremental approach for

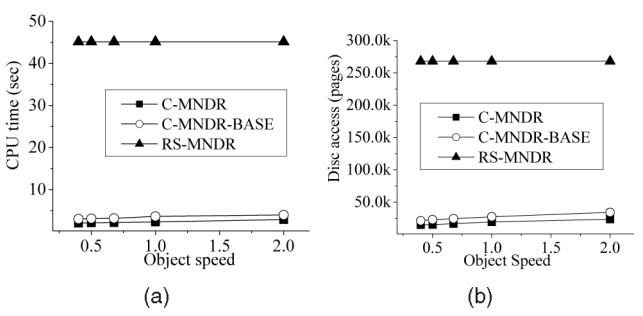

Fig. 18. The performance of query updates in C-MNDR as a function of the object speed. (a) CPU cost. (b) I/O cost.

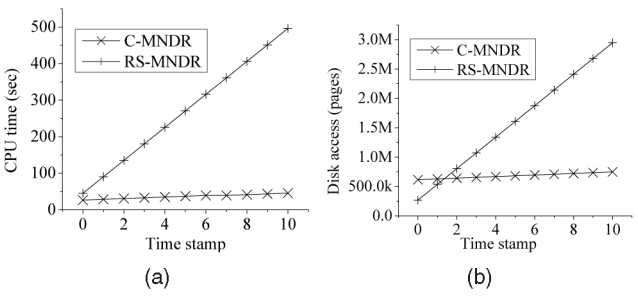

Fig. 19. The overall performance of C-MNDR. (a) CPU cost. (b) I/O cost.

continuous query processing, which saves a significant fraction of the recomputations required to update network connectivity and distance information, as well as the page accesses to retrieve road segments from the underlying R-tree. As time progresses and the queries continue to execute, C-MNDR increasingly and significantly outperforms RS-MNDR. Therefore, we conclude that C-MNDR achieves its objectives and provides a very efficient means of processing continuous range queries.

5 Conclusions

Location-based services have generated growing interest in both the academia and commercial enterprizes. Specifically, supporting mobile location-based queries in highly dynamic environments remains a big challenge. In this study, we addressed the problem of supporting continuous queries on moving objects in dense networks. Specifically, we designed a dual-index structure with a precomputing component for fast distance computation on moving objects. We also introduced the SD-tree structure that assists query update processing over time. Our experimental results demonstrate that our design is very efficient and scalable in coping with various numbers of moving objects in dense networks.

In the future, we plan to extend our work in several directions. First, we would like to study the distribution of moving objects in metro areas, which would provides us a more realistic simulation environment. Second, at its current stage, MOVNet assumes a stationary network. It would be preferable to also study traffic conditions and incorporate traffic events into the network data so that the system will be well suited for use in metro areas.

Acknowledgments

This research has been funded in part by US National Science Foundation (NSF) grants EEC-9529152 (IMSC ERC), CMS-0219463 (ITR), and IIS-0534761; and NUS AcRF grant WBS R-252-050-280-101/133.

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," *Proc. ACM SIGMOD*, 1990.

[2] T. Brinkhoff, "A Framework for Generating Network-Based Moving Objects," *GeoInformatica*, vol. 6, no. 2, pp. 153-180, 2002.

[3] H.-J. Cho and C.-W. Chung, "An Efficient and Scalable Approach to CNN Queries in a Road Network," *Proc. Int'l Conf. Very Large Data Bases (VLDB)*, 2005. Q1

[4] H.D. Chon, D. Agrawal, and A.E. Abbadi, "Range and kNN Query Processing for Moving Objects in Grid Model," *Mobile Networks and Applications*, vol. 8, no. 4, pp. 401-412, 2003.

[5] W. Feller, *An Introduction to Probability Theory and Its Applications*, vol. 1. Wiley, 1968.

[6] B. Gedik and L. Liu, "MobiEyes: Distributed Processing of Continuously Moving Queries on Moving Objects in a Mobile System," *Proc. Int'l Conf. Extending Database Technology (EDBT)*, 2004.

[7] A. Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching," *Proc. ACM SIGMOD*, 1984.

[8] H. Hu, D.L. Lee, and J. Xu, "Fast Nearest Neighbor Search on Road Networks," *Proc. Int'l Conf. Extending Database Technology (EDBT)*, 2006.

[9] H. Hu, J. Xu, and D.L. Lee, "A Generic Framework for Monitoring Continuous Spatial Queries over Moving Objects," *Proc. ACM SIGMOD*, 2005.

[10] X. Huang, C.S. Jensen, H. Lu, and S. Saltenis, "S-GRID: A Versatile Approach to Efficient Query Processing in Spatial Networks," *Proc. Int'l Symp. Spatial and Temporal Databases (SSTD)*, 2007.

[11] X. Huang, C.S. Jensen, and S. Saltenis, "The Islands Approach to Nearest Neighbor Querying in Spatial Networks," *Proc. Int'l Symp. Spatial and Temporal Databases (SSTD)*, 2005.

[12] C.S. Jensen, J. Kolárvr, T.B. Pedersen, and I. Timko, "Nearest Neighbor Queries in Road Networks," *Proc. ACM Int'l Symp. Advances in Geographic Information Systems (GIS)*, 2003.

[13] C.S. Jensen, D. Lin, and B.C. Ooi, "Query and Update Efficient B+-Tree Based Indexing of Moving Objects," *Proc. Int'l Conf. Very Large Data Bases (VLDB)*, 2004.

[14] M.R. Kolahdouzan and C. Shahabi, "Continuous K-Nearest Neighbor Queries in Spatial Network Databases," *Proc. Workshop Spatio-Temporal Database Management (STDBM)*, 2004.

[15] M.R. Kolahdouzan and C. Shahabi, "Voronoi-Based K-Nearest Neighbor Search for Spatial Network Databases," *Proc. Int'l Conf. Very Large Data Bases (VLDB)*, 2004.

[16] M.F. Mokbel, X. Xiong, and W.G. Aref, "SINA: Scalable Incremental Processing of Continuous Queries in Spatio-Temporal Databases," *Proc. ACM SIGMOD*, 2004.

[17] K. Mouratidis, M. Hadjieleftheriou, and D. Papadias, "Conceptual Partitioning: An Efficient Method for Continuous Nearest Neighbor Monitoring," *Proc. ACM SIGMOD*, 2005.

[18] K. Mouratidis, M.L. Yiu, D. Papadias, and N. Mamoulis, "Continuous Nearest Neighbor Monitoring in Road Networks," *Proc. Int'l Conf. Very Large Data Bases (VLDB)*, 2006.

[19] D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao, "Query Processing in Spatial Network Databases," *Proc. Int'l Conf. Very Large Data Bases (VLDB)*, 2003.

[20] J.M. Patel, Y. Chen, and V.P. Chakka, "STRIPES: An Efficient Index for Predicted Trajectories," *Proc. ACM SIGMOD*, 2004.

[21] S. Saltenis, C.S. Jensen, S.T. Leutenegger, and M.A. Lopez, "Indexing the Positions of Continuously Moving Objects," *Proc. ACM SIGMOD*, 2000.

[22] H. Samet, J. Sankaranarayanan, and H. Alborzi, "Scalable Network Distance Browsing in Spatial Databases," *Proc. ACM SIGMOD*, 2008.

[23] D. Stojanovic, A.N. Papadopoulos, B. Predic, S. Djordjevic-Kajan, and A. Nanopoulos, "Continuous Range Monitoring of Mobile Objects in Road Networks," *Data and Knowledge Eng.*, vol. 64, no. 1, pp. 77-100, 2008.

[24] J.A. Storer, *An Introduction to Data Structures and Algorithms*. Birkhauser Boston, 2001.

[25] Y. Tao, D. Papadias, and J. Sun, "The TPR*-Tree: An Optimized Spatio-Temporal Access Method for Predictive Queries," *Proc. Int'l Conf. Very Large Data Bases (VLDB)*, 2003.

[26] H. Wang and R. Zimmermann, "Snapshot Location-Based Query Processing on Moving Objects in Road Networks," *Proc. ACM Int'l Symp. Advances in Geographic Information Systems (GIS)*, 2008.

[27] X. Xiong, M.F. Mokbel, and W.G. Aref, "SEA-CNN: Scalable Processing of Continuous K-Nearest Neighbor Queries in Spatio-Temporal Databases," *Proc. Int'l Conf. Data Eng. (ICDE)*, 2005.

[28] X. Xiong, M.F. Mokbel, and W.G. Aref, "LUGrid: Update-Tolerant Grid-Based Indexing for Moving Objects," *Proc. Int'l Conf. Mobile Data Management (MDM)*, 2006.

[29] X. Yu, K.Q. Pu, and N. Koudas, "Monitoring K-Nearest Neighbor Queries over Moving Objects," *Proc. Int'l Conf. Data Eng. (ICDE)* 2005.

Haojun Wang received the BS degree in information engineering from the Shanghai Jiao Tong University, China, in 1998, and the MS and PhD degrees in computer science from the University of Southern California, Los Angeles, in 2006 and 2009, respectively. His research interests include spatial data management, peer-to-peer systems, and mobile location-based systems. He is a member of the IEEE.

Roger Zimmermann (S'93-M'99-SM'07) received the informatik ingenieur HTL degree from the Höhere Technische Lehranstalt in Brugg-Windisch, Aargau, Switzerland, in 1986, and the MS and PhD degrees in computer science from the University of Southern California, Los Angeles, in 1994 and 1998, respectively. He is currently an associate professor in the Department of Computer Science at the National University of Singapore (NUS), where he is also an investigator in the Interactive and Digital Media Institute (IDMI). Prior to joining NUS, he held the position of research area director in the Integrated Media Systems Center (IMSC) at the University of Southern California (USC). His research interests are in the areas of distributed and peer-to-peer systems, collaborative environments, streaming media architectures, georeferenced video search, and mobile location-based services. He has coauthored a book, four patents, and more than a 100 conference publications, journal articles, and book chapters in the areas of multimedia and information management. He is an associate editor of the *ACM Computers in Entertainment* magazine and the *ACM Transactions on Multimedia Computing, Communications and Applications* journal. He is a member of the ACM. He is a senior member of the IEEE.