International Journal of Geo-Information

MDPI

*Article*

# The SSP-Tree: A Method for Distributed Processing of Range Monitoring Queries in Road Networks

**HaRim Jung and Ung-Mo Kim ***

College of Information and Communication Engineering, Sungkyunkwan University, Suwon 440-746, Korea; harim3826@gmail.com

* Correspondence: ukim@skku.edu; Tel.: +82-31-290-7118

Received: 11 August 2017; Accepted: 24 October 2017; Published: 26 October 2017

**Abstract:** This paper addresses the problem of processing range monitoring queries, each of which continuously retrieves moving objects that are currently located within a given query range. In particular, this paper focuses on processing range monitoring queries in the road network, where movements of the objects are constrained by a predefined set of paths. One of the most important challenges of processing range monitoring queries is how to minimize the wireless communication cost and the server computation cost, both of which are heavily dependent on the amount of location-update stream generated by moving objects. The traditional centralized methods for range monitoring queries assume that moving objects periodically send location-updates to the server. However, when the number of moving objects becomes increasingly large, such an assumption may no longer be acceptable because the amount of location-update stream becomes enormous. Recently, some distributed methods have been proposed, where moving objects utilize their available computational capabilities for sending location-updates to the server only when necessary. Unfortunately, the existing distributed methods only deal with the objects moving in Euclidean space, and thus they cannot be extended to processing range monitoring queries over the objects moving along the road network. In this paper, we propose the distributed method for processing range monitoring queries in the road network. To utilize the computational capabilities of moving objects, we introduce the concept of vicinity region. A vicinity region, assigned to each moving object $o$, makes $o$ monitor whether or not it should be included in the results of nearby queries. The proposed method includes (i) a new spatial index structure, called the Segment-based Space Partitioning tree (SSP-tree) whose role is to efficiently search the appropriate vicinity regions for moving objects based on their heterogeneous computational capabilities and (ii) the details of the communication strategy between the server and moving objects, which significantly reduce the wireless communication cost as well as the server computation cost. Through simulations, we verify the effectiveness for processing range monitoring queries over a large number of moving objects (up to 100,000) in the road network (modeled as an undirected graph).

**Keywords:** spatial databases; location-based services; road networks; range monitoring queries; mobile devices; energy efficiency

## 1. Introduction

The proliferation of handheld computing devices equipped with positioning systems has led to the rapid growth of *Location-Based Services* (*LBSs*) [1]. In this paper, we study the problem of processing *range monitoring queries*. A range monitoring query $q = (q.p, q.d)$, issued over a set of moving objects $O$, (i) retrieves a subset of moving objects $\acute{O}$ ($\subseteq O$) that are located within a query distance $q.d$ from a query point $q.p$; and (ii) continuously updates $\acute{O}$ as the moving objects change their location. Range monitoring queries often play an important role for supporting LBSs. For example, let us consider the following scenarios. A gas station owner (i.e., client) wants to send promotional

coupons to all cars (i.e., moving objects) currently located near her gas station; a child safety service provider (i.e., client) wants to monitor the potential dangerous areas to alert parents when their children (i.e., moving objects) enter these areas; a traffic management department wants to monitor the traffic conditions of the main highways in a city. In such scenarios, the functionality of monitoring moving objects that are currently located within a region of interest is highly required.

A large number of methods for processing range monitoring queries were proposed [2–14], which can be broadly classified into two categories according to the mobility of query points; one deals with static query points (e.g., facilities such as the gas station mentioned above) [2–7], whereas the other deals with moving query points (e.g., free taxis looking for the nearby passengers) [8–14]. Our proposed method belongs to the former category. Most existing methods for processing range monitoring queries are highly centralized in the sense that moving objects periodically send location-updates to the server, and that the server carries out all the computations for processing range monitoring queries [1]. Therefore, their focus is developing algorithms for efficiently process the queries at the server side. However, when the number of moving objects becomes very large, they may suffer from a severe communication bottleneck as well as overwhelming server workload due to a huge amount of location-update stream generated by the moving objects.

One of the key challenges of processing range monitoring queries is how to minimize the wireless communication cost and the server computation cost, both of which heavily depend on the amount of location-update stream. As the amount of location-update stream is increased, the wireless communication cost and the server computation cost are also increased accordingly. Recently, some distributed methods, which aim to reduce the amount of location-update stream by utilizing the computational capabilities of moving objects, have been proposed [2,3]. In the distributed methods, the server assigns each moving object $o$ several queries, and $o$ locally monitors its movement against these queries. Only when $o$ affects the current result of any of the assigned queries, does it send a location-update to the server to let the server update the result of the corresponding query. As such, in the distributed methods, moving objects no longer need to periodically send location-updates to the server, and thus the amount of location-update stream can be reduced. The additional benefit of the distributed methods is that the moving objects can save energy consumption by reducing the number of wireless message transmissions (i.e., the number of location-updates sent to the server). Please note that a wireless message transmission is an energy expensive operation.

Unfortunately, the existing distributed methods only deal with the objects moving freely in Euclidean space. In most real-life scenarios of LBSs, the objects are allowed to move only on a pre-defined set of paths specified by the underlying road network. In Euclidean space, the distance between a moving object $o$ and a query point $q.p$ is defined as the length of the straight-line connecting them, whereas in the road network, the distance between $o$ and $q.p$ is defined as the total length of the shortest path connecting them. Therefore, the existing distributed methods cannot support range monitoring queries in the road network. For example, let us consider a range monitoring query $q = (q.p, q.d)$, where $q.p$ is shown in Figure 1 and $q.d = 10$. As shown in Figure 1a, if $q$ is issued over a set of moving object $O = \{o_1, o_2, o_3, o_4, o_5, o_6\}$ in Euclidean space, its current result is $\{o_3, o_4, o_5, o_6\}$, where the red circle in the figure represents the *query range* of $q$. On the other hand, as shown in Figure 1b, if $q$ is issued over $O$ in the road network, its current result is $\{o_3, o_6\}$, where a set of red *line segments* in the figure represents the query range of $q$. (Note: Each number in brackets in Figure 1b indicates the distance between $o_{i\,(1 \le i \le 6)} \in O$ and $q.p$ in the road network.) We refer to each line segment that belongs to the query range in the road network as the *query segment*.

In this paper, we propose the distributed method for processing range monitoring queries in the road network. To utilize the computational capabilities of moving objects, we introduce the concept of *vicinity region*. Given a moving object $o$, $o$'s vicinity region, denoted by $VR(o)$, is a rectangular region, which *contains* (i) the point of $o$'s current location and (ii) a number of query segments. By letting the server assign $o$ (i) $VR(o)$ and (ii) query segments *inside* $VR(o)$, $o$ can monitor by itself whether it affects the results of nearby queries while it is moving. The moving object $o$ sends a location-update to the

server whenever (i) it leaves $VR(o)$ or (ii) it affects the result of some nearby query $q$. In the former case, the server assigns $o$ a new vicinity region together with new query segments, while in the latter case, the server updates the result of $q$ accordingly.

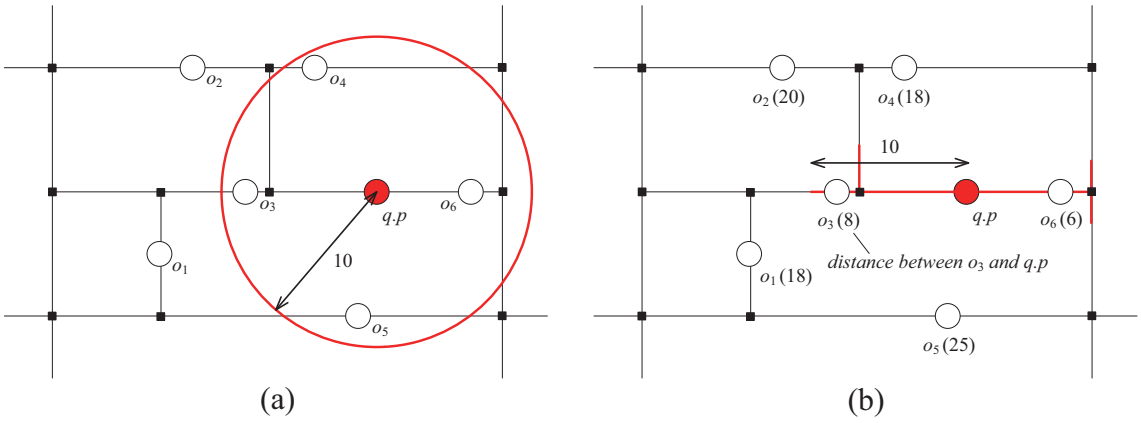


**Figure 1.** Difference between Euclidean space and the road network. (**a**) The query range in Euclidean space; (**b**) The query range in the road network.

One critical problem is how to determine the suitable size of a vicinity region $VR(o)$ for each moving object $o$. If $VR(o)$ is too small, $o$ needs to frequently send a location-update to the server for receiving a new vicinity region. On the other hand, if $VR(o)$ is too large, $o$ needs to monitor a large number of queries, which imposes a computationally intensive burden on $o$. In general, a handheld computing device carried by $o$ executes multiple applications, and thus if a single LBS application occupies substantial computational resources, the service quality of the other applications may deteriorate drastically. With this problem in mind, we propose a new spatial index structure, called the *Segment-based Space Partitioning tree* (*SSP-tree*). The role of the SSP-tree to efficiently search the appropriate vicinity regions for moving objects based on their *heterogeneous* computational capabilities. We also describe the details of the communication strategy between the server and moving objects for cooperative processing of range monitoring queries in the road network.

In summary, we propose (i) the concept of vicinity region; (ii) the SSP-tree for vicinity region search; and (iii) the vicinity region based communication strategy between the server and moving objects for distributed processing of static range monitoring queries. Through simulations, we verify the effectiveness of the proposed method for processing static range monitoring queries in terms of the wireless communication cost and the server computation cost.

## 2. Problem Statement and System Overview

### 2.1. Background and Problem Statement

In this paper, we address the problem of processing range monitoring queries in the road network. The road network is modeled as an undirected graph $G = (V, E)$, where $V = \{v_1, v_2 \ldots, v_{|V|}\}$ is a set of vertices and $E(\subseteq V \times V) = \{v_i v_j | 1 \leq i \leq |E|, 1 \leq j \leq |E|, i \neq j\}$ is a set of edges. A vertex $v \in V$ corresponds to a road intersection or dead-end. On the other hand, an edge $v_i v_j \in E$ corresponds to a road segment, which connects two vertices $v_i$ and $v_j$. For convenience of notation, we sometimes indicate an edge $v_i v_j$ as $e$. We can assume that each road segment of the road network is a straight line because a curved road segment can be transformed into a set of straight lines by adding extra vertices and edges to $G$. Therefore, the length of an edge $v_i v_j$ can be the Euclidean distance between its two endpoints $v_i$ and $v_j$. Hereafter, we use $dist_E(\cdot, \cdot)$ to denote the Euclidean distance between any two points (including the vertices) in the road network $G$.

**Definition 1.** *Given two vertices $v_a$ and $v_b$ in the road network $G = (V, E)$, where $v_a v_b \notin E$, a **path** from $v_a$ to $v_b$, denoted by $P(v_a, v_b)$, is a sequence of vertices $(v_{p_1}, v_{p_2} \ldots, v_{p_k})$ such that $v_{p_1} = v_a$, $v_{p_k} = v_b$, and for each consecutive pair of vertices $(v_{p_i}, v_{p_{i+1}})$ for all $1 \leq i < k$, the condition: $v_{p_i} v_{p_{i+1}} \in E$ holds. Then, the **path length** of $P(v_a, v_b)$ is calculated as:*

$$L(P(v_a, v_b)) = \sum_{i=1}^{k-1} dist_E(v_{p_i}, v_{p_{i+1}}), \tag{1}$$

*where $v_{p_1} = v_a$ and $v_{p_k} = v_b$.*

**Definition 2.** *Given two vertices $v_a$ and $v_b$ in the road network $G = (V, E)$, where $v_a v_b \notin E$, there can exist more than one path from $v_a$ to $v_b$. Then, the **shortest path** from $v_a$ to $v_b$, denoted by $SP(v_a, v_b)$, is defined as:*

$$SP(v_a, v_b) = \underset{P(v_a, v_b) \in Pset(v_a, v_b)}{\arg\min} L(P(v_a, v_b)), \tag{2}$$

*where $Pset(v_a, v_b)$ is the set of all the paths from $v_a$ to $v_b$. The path length of $SP(v_a, v_b)$, i.e., $L(SP(v_a, v_b))$, is called the **shortest path length**.*

Consider an example of the road network $G$ in Figure 2a, which serves as a running example in the rest of this paper. In Figure 2a, $G$ consists of 10 vertices and 13 edges. (Note: The vertices and edges outside of the workspace are omitted.) Each number near to each edge in the figure indicates its length (i.e., the Euclidean distance between its two endpoints). From the figure, it can be easily seen that $SP(v_1, v_9) = (v_1, v_4, v_7, v_9)$ and $L(SP(v_1, v_9)) = 29$.

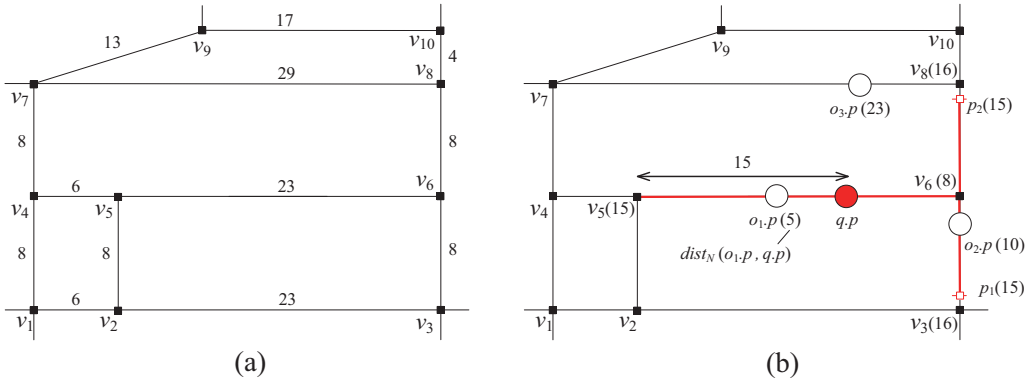


**Figure 2.** An example of the range monitoring query in the road network. (**a**) The road network $G$; (**b**) The range monitoring query in $G$.

Let $O = \{o_1, o_2, \ldots, o_{|O|}\}$ and $Q = \{q_1, q_2, \ldots, q_{|Q|}\}$ be a set of moving objects and a set of range monitoring queries, respectively. Each object $o \in O$ is constrained to move only along the edges in the road network $G$, and thus the point of $o$'s location, denoted by $o.p$, at a particular snapshot in time always lies on an edge in $G$. Each query $q \in Q$ is represented by a tuple $(q.p, q.d)$, where $q.p$ is a query point lying on an edge in $G$ and $q.d$ is a query distance.

**Definition 3.** *Given two points $p_a$ and $p_b$, where $p_a$ lies on an edge $v_{i_1} v_{j_1}$ and $p_b$ lies on an edge $v_{i_2} v_{j_2}$, the distance between $p_a$ and $p_b$ in the road network $G$, which is called the **network distance** (between $p_a$ and $p_b$) and denoted by $dist_N(p_a, p_b)$, is defined as:*

$$dist_N(p_a, p_b) = \begin{cases} dist_E(p_a, p_b) & \text{if } v_{i_1} = v_{i_2} \text{ and } v_{j_1} = v_{j_2} \text{ (i.e., } p_a \text{ and } p_b \text{ lie on the same edge);} \\ \min_{x \in \{i_1, j_1\}, y \in \{i_2, j_2\}} \left( dist_E(p_a, v_x) + L(SP(v_x, v_y)) + dist_E(v_y, p_b) \right) & \text{otherwise.} \end{cases} \tag{3}$$

Given the road network $G$ in Figure 2a, Figure 2b shows the network distance between a query point $q.p$ and several points on $G$ (e.g., vertices and points of moving objects' location). For example, $dist_N(q.p, v_5) = dist_E(q.p, v_5)$ and $dist_N(q.p, o_1.p) = dist_E(q.p, o_1.p)$ because $q.p$, $v_5$, and $o_1.p$ lie on the same edge $v_5v_6$. On the other hand, $dist_N(q.p, v_3) = dist_E(q.p, v_6) + L(SP(v_6, v_3)) + dist_E(v_3, v_3)$ and $dist_N(q.p, o_3.p) = dist_E(q.p, v_6) + L(SP(v_6, v_8)) + dist_E(v_8, o_3.p)$. In Figure 2b, each number in brackets indicates the network distance between $q.p$ and each point. (Note: The network distance between $q.p$ and some vertices are omitted for brevity.)

**Definition 4.** *A **range monitoring query** $q = (q.p, q.d)$, issued over a set of moving objects $O$, in the road network $G$, continually retrieves a subset $\acute{O}$ ($\subseteq O$) of moving objects for which the condition: $\forall o \in \acute{O}, dist_N(q.p, o.p) \leq q.d$ holds.*

**Definition 5.** *Given a query $q = (q.p, q.d)$, the **query range** of $q$ in the road network $G$, denoted by $QR(q)$, is a set of all points on the edges (in $G$) reachable from $q.p$ within $q.d$. Formally, $QR(q) = \{p | dist_N(q.p, p) \leq q.d$ and $p$ is a point in $G\}$.*

From Definitions 4 and 5, we can immediately know that given a query $q = (q.p, q.d)$ and a moving object $o$ ($\in O$) in the road network $G$, if and only if the point of $o$'s current location is inside the query range of $q$, can $o$ be the current result of $q$ (i.e., $o.p \in QR(q) \Leftrightarrow dist_N(q.p, o.p) \leq q.d$). In the road network $G$, the query range $QR(q)$ of a query $q$ consists of a set of *line segments*.

**Definition 6.** *Given an edge $v_iv_j$ in the road network $G$, a **line segment**, denoted by $s[p_a, p_b]$, where $p_a$ and $p_b$ are points on $v_iv_j$, is a set of all points on $v_iv_j$ between $p_a$ and $p_b$, i.e., the portion of $v_iv_j$ between $p_a$ and $p_b$.*

For example, in Figure 2b, the query range consists of all red line segments (i.e., $s[v_5, v_6]$, $s[v_6, p_1]$, and $s[v_6, p_2]$), assuming $q.d = 15$. Please note that, by definition, an edge $v_iv_j$ in $G = (V, E)$ is also considered to be a line segment formed by its two endpoints $v_i$ and $v_j$ (i.e., $v_iv_j = s[v_i, v_j]$). In this paper, we refer to each line segment that belongs to the query range $QR(q)$ of a query $q$ as the *query segment* of $q$, which we denote by $qs[p_a, p_b]$ or $qs$ for short. Therefore, given the query range $QR(q)$ of a query $q$, each query segment $qs[p_a, p_b] \in QR(q)$ satisfies the condition: $\forall p \in qs[p_a, p_b]$, $p \in QR(q)$.

The primary goal of our work is to reduce the amount of location-update stream (generated by moving objects) while maintaining the correct results of range monitoring queries in the road network. To this end, we use the concept of vicinity region so that moving objects send location-updates to the server only when necessary. Given a moving object $o$, $o$'s vicinity region $VR(o)$ is a rectangular region, which contains (i) the point of $o$'s current location and (ii) some query segments. By assigning each moving object $o$ (i) $VR(o)$ and (ii) query segments inside $VR(o)$, $o$ can locally monitor whether it may affect the results of nearby queries based on the following two lemmas.

**Lemma 1.** *Given a query $q = (q.p, q.d)$, a moving object $o \in O$, and one of the query segments $qs$ of $q$ in the road network $G$, let $o.\acute{p}$ and $o.p$ be the point of $o$'s last known location at time $\acute{t}$ and the point of $o$'s current location at time $t$ , respectively ($\acute{t} < t$). Suppose that $o.\acute{p} \notin qs$ and $o.p \in qs$, i.e., $o$ enters $qs$ from outside. Then, there exists a case such that $o$ affects the current result of $q$.*

**Proof.** To prove this lemma, it suffices to show that such a case exists. Without loss of generality, let us assume that $o.\acute{p} \notin QR(q)$, and thus $dist_N(q.p, o.\acute{p}) > q.d$, meaning that $o$ is not a result object at the last known time $\acute{t}$. From Definitions 4, 5, 6, and the description of the query segment, we know that $o.p \in QR(q)$, meaning that $o$ is a result object at the current time $t$. This immediately implies that $o$ becomes a new result object, and therefore $o$ affects the current result of $q$. □

**Lemma 2.** *Given the same setting and notation as Lemma 1, suppose that $o.\acute{p} \in qs$ and $o.p \notin qs$, i.e., $o$ leaves $qs$. Then, there also exists a case such that $o$ affects the current result of $q$.*

**Proof.** This lemma can be proved similarly as Lemma 1, and thus we omit the proof. □

Only when each moving object $o$ (i) leaves $VR(o)$ or (ii) enters or leaves any of the assigned query segments, does it send a location-update to the server in order to (i) receive a new vicinity region together with new query segments or (ii) let the server update the result of a nearby query $q$ if necessary. In this paper, we focus on how to efficiently search the appropriate vicinity regions for moving objects, and propose a new spatial index structure, namely the SSP-tree. We also present the details of the vicinity region-based communication strategy between the server and moving objects for processing range monitoring queries in the road network. To conclude Section 2.1, we summarize the frequently used notations in Table 1.

**Table 1.** Frequently used notation.

| Notation | Explanation |
|---|---|
| $G = (V, E)$ | A graph model of the road network ($V$: a set of vertices, $E$: a set of edges) |
| $v_i$ $(\in V)$ | A vertex in $G$ |
| $v_iv_j$ or $e$ $(\in E)$ | An edge in $G$ |
| $o$ | A moving object |
| $o.p$ | The point of $o$'s current location |
| $o.cap$ | $o$'s computational capability |
| $q = (q.p, q.d)$ | A range monitoring query ($q.p$: query point, $q.d$: query distance) |
| $QR(q)$ | The query range of $q$ |
| $qs[p_a, p_b]$ or $qs$ (of $q$) | A query segment (of $q$) |
| $dist_E(\cdot, \cdot)$ | The Euclidean distance between any two points in $G$ |
| $dist_N(\cdot, \cdot)$ | The network distance between any two points in $G$ |
| $N$ | A SSP-tree node or its corresponding subspace |
| $N.C$ | The count variable maintained in $N$ |
| $N.FL$ | The full list maintained in $N$ |
| $QRT$ | Query relevance table |
| $SRT$ | Segment relevance table |

### 2.2. System Overview

Figure 3 shows an overview of the system model. Similarly to the system model presented in the previous work [2–7], the system model we consider consists of three major components: moving objects, clients, and the central server.

- **Moving objects**: Each moving object $o$, which is identified by its unique identifier $oid$, is capable of sensing the point of its current location $o.p$ and has some available computational capability $o.cap$. We assume (i) that each moving object $o$ has a heterogeneous capability $o.cap$, which is measured by the maximum number of query segments it can process, and (ii) that $o.cap \geq \theta$, where $\theta$ is a system parameter indicating the minimum number of query segments $o$ should process; thus, a moving object with a more powerful capability can be assigned a larger vicinity region that contains a more number of query segments. There are two types of location-update messages sent from the moving objects to the server: RequestVR and UpdateResult. The former is for the purpose of receiving a new vicinity region, whereas the latter is to let the server update the result of a query (if necessary). For example, assuming the moving object $o_1$ in Figure 3 is assigned the vicinity region $VR(o_1)$ together with query segments $qs[v_6, p_1]$, $qs[v_6, p_2]$, and $qs[v_6, p_3]$, it sends the RequestVR message to the server because it leaves $VR(o_1)$. On the other hand, assuming the moving object $o_2$ in Figure 3 is assigned the vicinity region $VR(o_2)$ together with query segments $qs[v_1, v_4]$, $qs[v_2, v_5]$, and $qs[v_4, v_5]$, it sends the UpdateResult message to the server because it leaves $qs[v_1, v_4]$.
- **Clients**: Each client is able to issue multiple range monitoring queries over the moving objects, and continually receives the up-to-date results of these queries from the server via wireless or

high-speed wired connections. Each query $q$, issued by a client, is identified by its unique identifier $qid$, and its query point $q.p$ is assumed to be static; thus, the movement of $q.p$ can be treated as a deletion of the old query followed by an insertion of a new query.

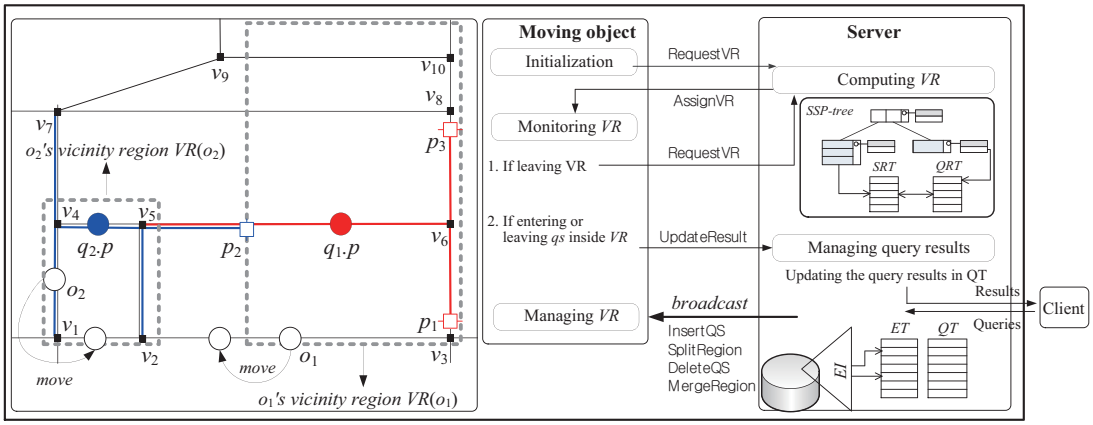


**Figure 3.** System overview.

- **Central server**: The server acts as an intermediary between moving objects and clients, i.e., moving objects and clients do not communicate directly, but indirectly through the server. In addition to the SSP-tree, the server maintains the following basic memory-resident data structures, which are commonly used in the existing methods for the road network [13,14].

  - ***Edge Index*** ($EI$): EI is the PMR-quadtree built on the edges in the road network $G = (V, E)$. Each leaf node of EI stores the identifiers of the edges it intersects. Given a query $q$, EI is used to identify the edge $e$ (i.e., $v_iv_j \in E$), where $q.p$ resides. Specifically, EI is traversed down to the leaf node that contains $q.p$, and $e$ is identified among the edges stored in this leaf node.
  - ***Edge Table*** ($ET$): ET is a table hashed on the identifier of each edge $e$. ET stores for $e$: (i) its endpoints (i.e., $v_i$ and $v_j$); (ii) its length (i.e., $dist_N(v_i, v_j) = dist_E(v_i, v_j)$) and (iii) the sets of edges adjacent to each of its endpoints. ET is used to maintain the connectivity information of the road network $G$.
  - ***Query Table*** ($QT$): QT is a table hashed on the identifier $qid$ of each query $q$. QT stores for $q$: (i) its query point $q.p$; (ii) its query distance $q.d$; (iii) a set of its query segments; and (iv) its current result. QT is used to maintain the information of the registered queries.

  As the intermediary between moving objects and clients in the system, the server performs the following three main tasks.

  - **Query registration**: When a new query $q$ is issued (or $q$ is terminated) by a client, the server inserts $q$ into (or deletes $q$ from) QT, updates the SSP-tree (and the additional data structures that will be described in Section 3), and broadcasts messages (e.g., InsertQS, SplitRegion, DeleteQS, and MergeRegion) to all the moving objects to notify them of these changes. Please note that notifying such common information through broadcasting is desirable because the communication overhead is irrelevant to the number of moving objects in a sense that a single message transmission from the server can be received by all the moving objects.
  - **Region assignment**: When a RequestVR message is arrived from the moving object $o$ that leaves its current vicinity region, the server searches a new vicinity region by traversing the SSP-tree, after which it sends an AssignVR message to $o$ for the purpose of assigning this new vicinity region (together with new query segments) to $o$.

– **Query result update**: When an UpdateResult message is arrived from a moving object $o$ that may affect the result of a query $q$, the server checks whether the current result of $q$ is affected by $o$. If so, the server updates the result of $q$. For example, in response to the UpdateResult message sent by the moving object $o_2$ in Figure 3, the server updates the result of $q_2$ (i.e., the server removes $o_2$ from the result of $q_2$).

## 3. The Proposed Method

As mentioned in the previous section, we focus on the issue of how to efficiently search the appropriate *vicinity regions* for moving objects. In this paper, similarly to the existing distributed methods in Euclidean space, we choose to partition the workspace into many disjoint subspaces for the search process of the vicinity regions. Please note that the space partitioning approaches used in the existing distributed methods cannot be applied in the road network because they assume that the query range (of a query) is even a rectangle instead of a circle [2,3].

As shown in Figure 4a, if the query ranges $QR(q_1)$ and $QR(q_2)$ of the queries $q_1$ and $q_2$, respectively, in the road network are approximated by the rectangles, *false positives* may be generated in the current results of $q_1$ and $q_2$. In the LBS applications, false positives are more harmful than false negatives because the former can lead to the wrong query results. For example, let us assume that the moving object $o_1$ in Figure 4a is assigned (i) the entire workspace as its vicinity region $VR(o_1)$ and (ii) two approximated rectangles of query ranges $QR(q_1)$ and $QR(q_2)$. Then, when $o_1$ moves as shown in the figure, $o_1$ does not send the UpdateResult message to the server because it does not leave the approximated rectangle of $QR(q_2)$. As a result, the server does not know the point of $o_1$'s current location $o_1.p$, and thus $o_1$ is wrongly included in the current result of $q_2$ although $o_1.p \notin QR(q_2)$ (i.e., $dist_N(q_2.p, o_1.p) > q_2.d$).

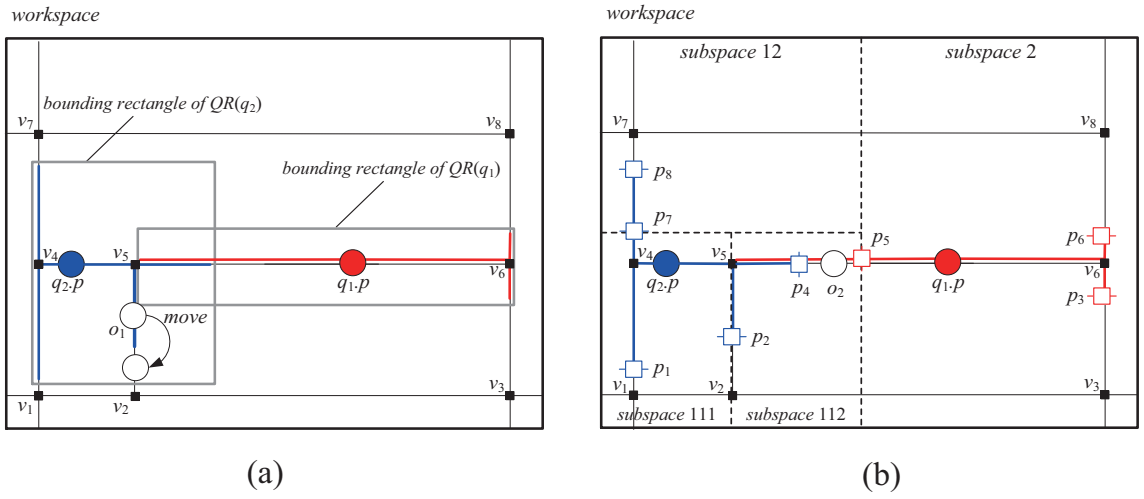


**Figure 4.** Examples of approximation of query ranges and the space partitioning approach used in this paper. (**a**) Approximation of query ranges; (**b**) The space partitioning approach used in this paper.

Figure 4b shows an example of the space partitioning approach used in this paper, which recursively partitions the workspace into two equal subspaces until the number of query segments *inside* each subspace is no more than the split threshold $\theta$. (Note: In the figure, $\theta$ is assumed to be 4.) In this paper, we assume that a subspace is horizontally or vertically partitioned along its longer dimension. Specifically, given a subspace, if its width is longer than its height, it is partitioned vertically, otherwise, it is partitioned horizontally. With such a space partitioning approach, the server can use a subspace as a vicinity region $VR(o)$ of each moving object $o$. The size of $VR(o)$ (i.e., the size of the subspace used as $VR(o)$) is determined by $o$'s capability $o.cap$; thus if $o.cap = n$, $VR(o)$ must contain the point of $o$'s current location $o.p$ and no more than $n$ query segments.

For example, assuming $o_2.cap$ of the moving object $o_2$ in Figure 4b is 3, $o_2$ is assigned (i) *subspace 112* as its vicinity region $VR(o_2)$ and (ii) three query segments $qs[p_2, v_5]$, $qs[v_5, p_4]$, and $qs[v_5, p_5]$.

To efficiently support partitioning the workspace, the *SSP-tree* is used, which will be presented in detail throughout this section.

### 3.1. Query Segment Computation

In this subsection, we describe an algorithm COMPUTESEG for the query segment computation. When a new query $q = (q.p, q.d)$ is issued by a client, COMPUTESEG, which takes a query point $q.p$ and a query distance $q.d$ as inputs, computes the query segments of $q$ based on Dijkstra's algorithm. Before we describe the details of COMPUTESEG, we introduce a distance metric, $mindist_N(q.p, v_iv_j)$, which is defined between a query point $q.p$ and an edge $v_iv_j$ in the road network $G$, and serves as the lower bound for filtering the unnecessary edges in the query segment computation.

**Definition 7.** *Given a query point $q.p$ and an edge $v_iv_j$ in the road network $G$, $\boldsymbol{mindist_N(q.p, v_iv_j)}$ is defined as:*

$$mindist_N(q.p, v_iv_j) = \begin{cases} 0 & \text{if } q.p \in v_iv_j; \\ \min\left(dist_N(q.p, v_i), dist_N(q.p, v_j)\right) & \text{otherwise.} \end{cases} \quad (4)$$

**Lemma 3.** *Given a query point $q.p$ and an edge $v_iv_j$ in the road network $G$, $\forall p \in v_iv_j$, $mindist_N(q.p, v_iv_j) \leq dist_N(q.p, p)$, where $p$ denotes a point.*

**Proof.** We prove this lemma by contradiction. Let us assume that there exists a point $\acute{p} \in v_iv_j$ such that $dist_N(q.p, \acute{p}) < mindist_N(q.p, v_iv_j)$. We distinguish two cases:

1. If $q.p \in v_iv_j$, $mindist_N(q.p, v_iv_j) = 0$. This immediately contradicts the assumption because $dist_N(q.p, \acute{p})$ cannot be less than 0.
2. If $q.p \notin v_iv_j$, $mindist_N(q.p, v_iv_j) = \min\left(dist_N(q.p, v_i), dist_N(q.p, v_j)\right)$. Let us consider the subcase, where $mindist_N(q.p, v_iv_j) = dist_N(q.p, v_i)$. Then, when we simplify the Equation (3) to obtain: $dist_N(q.p, \acute{p}) = \min\left(\, dist_N(q.p, v_i) + dist_E(v_i, \acute{p}), dist_N(q.p, v_j) + dist_E(v_j, \acute{p})\right)$, $dist_N(q.p, \acute{p}) = dist_N(q.p, v_i) + dist_E(v_i, \acute{p})$. This leads to a contradiction to the assumption because $dist_N(q.p, v_i) \leq dist_N(q.p, v_i) + dist_E(v_i, \acute{p})$. The subcase, where $mindist_N(q.p, v_iv_j) = dist_N(q.p, v_j)$, leads to the same contradiction as the former subcase.

Therefore, $\acute{p}$ cannot exist. □

Algorithm 1 is the pseudocode of COMPUTESEG, assuming $e$ is the edge that contains $q.p$ ($e$ is identified by using EI). First, COMPUTESEG initializes an empty min-heap $H$ to traverse the vertices in the road network $G$ in the ascending order of their network distance from $q.p$ (line 1). Next, COMPUTESEG enheaps the endpoints (i.e., vertices) of $e$ into $H$ with keys equal to their network distance from $q.p$ (lines 2–3). Then, COMPUTESEG iteratively deheaps a vertex from $H$.

**Algorithm 1** COMPUTESEG($q.p$, $q.d$)

**Input** $q.p$: query point of $q$, $q.d$: query distance of $q$
**Output** $qs\_set$: a set of query segments $q$
1: initialize an empty min-heap $H$;
2: let $e$ be the edge containing $q.p$;
3: enheap the endpoints (vertices) of $e$ into $H$ with keys equal to their network distance from $q.p$;
4: **repeat**
5: deheap the top entry $[v_i, dist_N(q.p, v_i)]$ from $H$;
6: mark $v_i$ as visited;
7: **for** each adjacent vertex $v_j$ of $v_i$ **do**
8: **if** $v_j$ has not been visited and $dist_N(q.p, v_i) \leq q.d$ **then**
9: set $dist_N(q.p, v_j)$ to $dist_N(q.p, v_i) + dist_N(v_i, v_j)$; // $dist_N(v_i, v_j) = dist_E(v_i, v_j)$
10: enheap $v_j$ into $H$ with a key $dist_N(q.p, v_j)$;
11: **else if** $v_j$ has been visited **then**
12: compute $mindist_N(q.p, v_i v_j)$;
13: **if** $mindist_N(q.p, v_i v_j) \leq q.d$ **then**
14: compute the query segment $qs$ of $q$;
15: insert $qs$ into $qs\_set$;
16: **until** ($H$ is empty)
17: return $qs\_set$;

For each deheaped vertex $v_i$, COMPUTESEG marks $v_i$ as *visited* and checks each of its adjacent vertices $v_j$ whether or not $v_j$ has been visited ($v_j$ is identified by using ET). If $v_j$ has not been visited, COMPUTESEG further checks if $dist_N(q.p, v_i) \leq q.d$. Only if this is the case, is $v_j$ enheaped into $H$ with a key $dist_N(q.p, v_i) + dist_N(v_i, v_j)$ (lines 8–10). The rationale is that if (i) $v_j$ has not been visited and (ii) $dist_N(q.p, v_i) > q.d$, it is guaranteed that $dist_N(q.p, v_j) > q.d$. This implies that no portion of each edge $v_j v_k$ formed by $v_j$ and each of its adjacent vertices $v_k$ (including $v_i$) can be the query segment of $q$ because $mindist_N(q.p, v_j v_k) > q.d$. From Lemma 3, we know that the network distance between $q.p$ and every point $p \in v_j v_k$ is equal to or greater than $mindist_N(q.p, v_j v_k)$, and thus that $\forall p \in v_j v_k$, $dist_N(q.p, p) > q.d$. Therefore, $v_j$ need not be enheaped and expanded. On the other hand, if $v_j$ has been visited (i.e., $v_j$ has been deheaped before), COMPUTESEG further checks if $mindist_N(q.p, v_i v_j) \leq q.d$. If so, COMPUTESEG computes the query segment of $q$ (lines 11–15). This process continues until $H$ becomes empty, and finally COMPUTESEG returns the query segments of $q$.

### *3.2. The Segment-Based Space Partitioning Tree (SSP-Tree)*

#### 3.2.1. Description

The SSP-tree is a hierarchical data structure that recursively splits the workspace into two subspaces. It is a binary tree, where each node $N$ represents a subspace of the workspace, and $N$'s two children represent equal halves of this subspace. Hereafter, we say that a tree node $N$ corresponds to a subspace or vice versa if $N$ represents this subspace, and without ambiguity, we use the symbol '$N$' to denote both a tree node and its corresponding subspace.

Given a set of query segments on the workspace that corresponds to the root, if the number of these query segments is greater than the split threshold $\theta$ (i.e., the minimum number of query segments each moving object should process), the workspace is split into two subspaces, each of which corresponds to a child node $N$ of the root. When a query segment $qs$ *partially intersects* $N$, it is also split into two query segments $qs \cap N$ and $qs - (qs \cap N)$ so that $qs \cap N$ is *inside* $N$. When necessary, we refer to each query segment computed by COMPUTESEG (e.g., $qs$ that partially intersects $N$) as the *original query segment* for distinguishing it from the *newly generated query segments* (e.g., $qs \cap N$ and $qs - (qs \cap N)$). The process recursively continues until each node has no more than $\theta$ query segments inside it. We define the simple intersection relationships between a query segment $qs$ and a node $N$ of the SSP-tree.

**Definition 8.** *Given a query segment $qs[p_a, p_b]$ and a node $N$ of the SSP-tree, let $qs[p_a, p_b]^\circ$ be the interior of $qs[p_a, p_b]$ (i.e., $qs[p_a, p_b]^\circ = qs[p_a, p_b] - \{p_a, p_b\}$). Then, there can be three relationships as follows:*

- ***Inside or contain relationship*** *(see Figure 5a): We say that $qs[p_a, p_b]$ is **inside** $N$ or $N$ **contains** $qs[p_a, p_b]$ if $qs[p_a, p_b] \cap N = qs[p_a, p_b]$.*
- ***Partially intersect relationship*** *(see Figure 5b): We say that $qs[p_a, p_b]$ **partially intersects** $N$ or vice versa if $(qs[p_a, p_b] \cap N \neq qs[p_a, p_b]) \wedge (qs[p_a, p_b]^\circ \cap N \neq \emptyset) \wedge (\{p_a, p_b\} \cap N \neq \emptyset)$.*
- ***Fully intersect relationship*** *(see Figure 5c): We say that $qs$ **fully intersects** $N$ or vice versa if $(qs[p_a, p_b] \cap N \neq qs[p_a, p_b]) \wedge (qs[p_a, p_b]^\circ \cap N \neq \emptyset) \wedge (\{p_a, p_b\} \cap N = \emptyset)$.*

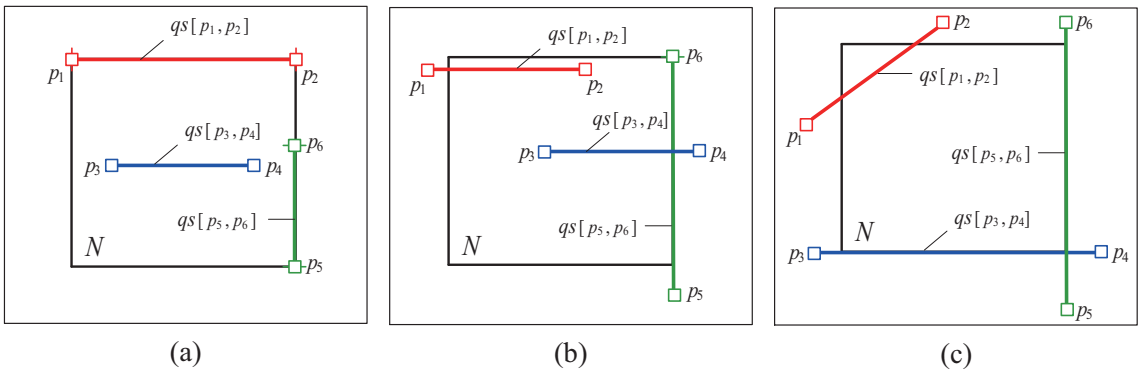


**Figure 5.** Simple intersection relationships between $qs$ and $N$. (**a**) $qs$ is inside $N$ or $N$ contains $qs$; (**b**) $qs$ partially intersects $N$ or vice versa; (**c**) $qs$ fully intersects $N$ or vice versa.

Now, we describe the structure and properties of the SSP-tree. A leaf node of the SSP-tree stores at most $\theta$ tuples of the form $(qs, flag)$, where $qs$ is a query segment inside $N$ and $flag$ is a single bit that is set (i.e., logical 1) if $qs$ is an original query segment. A non-leaf node stores two entries of the form $(ptr, N)$, where $ptr$ is a pointer to a child node and $N$ is a subspace that corresponds to the child node pointed to by $ptr$. The SSP-tree satisfies the following properties:

- A tuple $(qs, flag)$ is stored in a leaf node $N$ if $qs$ is inside $N$.
- A tuple $(qs, flag)$ can be redundantly stored in two leaf nodes if $qs$ lies along the splitting line $SL$ that separates these two leaf nodes (i.e., if $qs \cap SL = qs$).
- For each entry $(ptr, \acute{N})$ stored in a non-leaf node $N$, $\acute{N}$ represents one of the equal halves of $N$'s subspace.
- Each (leaf and non-leaf) node $N$ maintains (i) a variable, called the *count variable* (denoted by $N.C$), which records the total number of query segments inside $N$, and (ii) a list, called the *full list* (denoted by $N.FL$), which stores the original query segments that fully intersects $N$.

Assuming $\theta = 3$, Figure 6 shows an example of the SSP-tree built on two queries $q_1$ and $q_2$ in the road network $G$ in Figure 2a, where the original query segments of $q_1$ and $q_2$ are $\{qs[v_5, v_6], qs[v_6, p_1], qs[v_6, p_8]\}$ $(= QR(q_1))$ and $\{qs[v_5, p_2], qs[v_5, p_3], qs[v_5, p_5]\}$ $(= QR(q_2))$, respectively.

To track each query $q$ and its query segments (including the original query segments and the newly generated query segments), the server maintains the *Query Relevance Table* (*QRT*). We say that a query $q$ is *relevant* to a query segment $qs$ or vice versa if $qs$ belongs to $QR(q)$ (i.e., $qs \in QR(q)$). For example, in Figure 6, the query $q_1$ is relevant to the query segments $qs[v_5, p_4]$, $qs[v_6, p_1]$, $qs[v_6, p_6]$, $qs[v_6, p_7]$, and $qs[p_7, p_8]$. On the other hand, the query $q_2$ is relevant to the query segments $qs[v_5, p_2]$, $qs[v_5, p_3]$, $qs[v_5, p_4]$, and $qs[p_4, p_5]$. A query segment $qs$ can have more than one relevant queries. For example, $qs[v_5, p_4]$ is relevant to both $q_1$ and $q_2$. Each row of QRT is a tuple of the form $(qs, qid\_list)$

indexed by $qs$, where $qs$ is a *distinct* query segment and $qid\_list$ is a list that stores the identifiers of the queries that are relevant to $qs$.

The server also maintains the *Segment Relevance Table* (*SRT*) to identify the relevance between each original query segment $qs$ and the newly generated query segments. Similarly to the above, we say that an original query segment $qs$ is relevant to a newly generated query segment $\acute{qs}$ or vice versa if $\acute{qs}$ is split from $qs$. For example, in Figure 6, the original query segment $qs[v_5, v_6]$ is relevant to the newly generated query segments $qs[v_5, p_4]$ and $qs[v_6, p_6]$, while the original query segment $qs[v_5, p_5]$ is relevant to the newly generated query segments $qs[v_5, p_4]$ and $qs[p_4, p_5]$. It is important to note that in contrast to the original query segments that partially intersect a node $N$ of the SSP-tree, the original query segments that fully intersect $N$ are not split but stored in $N.FL$. For example, the query segment $qs[p_4, p_6]$ is not generated because $qs[v_5, v_6]$ fully intersects the tree node $N_{221}$; instead, $qs[v_5, v_6]$ is stored in $N_{211}.FL$. A newly generated query segment $\acute{qs}$ can have more than one relevant original query segments. For example, $qs[v_5, p_4]$ is relevant to both $qs[v_5, v_6]$ and $qs[v_5, p_5]$. Each row of SRT is a tuple of the form $(\acute{qs}, qs\_list)$ indexed by $\acute{qs}$, where $\acute{qs}$ is a distinct newly generated query segment and $qs\_list$ is a list that stores the original query segments that are relevant to $\acute{qs}$.

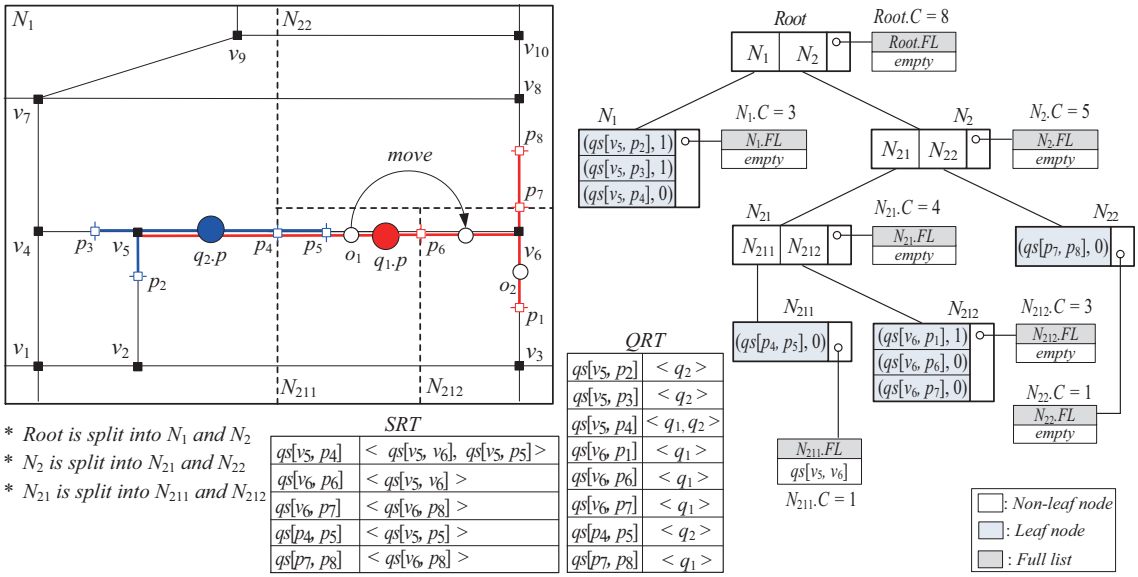


**Figure 6.** An example of the SSP-tree.

Given a (leaf or non-leaf) node $N$ of the SSP-tree and a moving object $o$, $N$ can become a candidate for $o$'s vicinity region if $N$ contains (i) $o.p$ and (ii) $N.C \leq o.cap$. In this connection, the first advantage of maintaining $N.FL$ at each node $N$ is that the server can assign a larger (but still appropriate) vicinity region to each moving object $o$ than not maintaining it. This is because each query segment $qs$ that fully intersects $N$ does not lead to the generation of a new query segment $qs \cap N$, and thus the value of $N.C$ is not increased. Assuming $o$ is assigned $N$ as its vicinity region, the second advantage is that $o$ can reduce the number of sending unnecessary UpdateResult messages to the server because $qs \cap N$ is not generated and assigned to $o$. Here, it should be validated that $qs \cap N$ need not be assigned to $o$. This is accomplished by the following lemma.

**Lemma 4.** *Given a node $N$ of the SSP-tree, a query $q = (q.p, q.d)$, an object $o$ moving only inside $N$, and one of the $q$'s relevant query segments $qs[p_a, p_b]$ that fully intersects $N$, suppose that $o$ enters or leaves $qs[p_a, p_b]$. In such a case, $o$ cannot affect the current result of $q$.*

**Proof.** From Definition 8, we know that both endpoints $p_a$ and $p_b$ of $qs[p_a, p_b]$ are outside $N$. In addition, from Section 2.1, we know that $\forall p \in qs[p_a, p_b]$, $dist_N(q.p, p) \leq q.d$. If $o$ enters

$qs[p_a, p_b]$, it comes from one of the query segments that is connected with $qs[p_a, p_b]$ at the point $p_c$, where $p_c \in qs[p_a, p_b]^\circ$ and $p_c$ is inside $N$, because $o$ is moving only inside $N$. Let this query segment be $qs[p_c, p_d]$. Then, $qs[p_c, p_d]$ must also be the relevant query segment of $q$, i.e., $qs[p_c, p_d] \in QR(q)$, because $dist_N(q.p, p_c) < q.d$. Similarly, if $o$ leaves $qs[p_a, p_b]$, it enters one of the query segments that must also be the relevant query segment of $q$. Therefore, $o$ cannot affect the current result of $q$. □

For example, assuming the capability $o_1.Cap$ of the moving object $o_1$ in Figure 6 is 3, $o_1$ is assigned the tree node $N_{211}$ as its vicinity region together with only the query segment $qs[p_4, p_5]$. When $o_1$ moves as shown in the figure, it sends only the RequestVR message to the server for receiving a new vicinity region. Then, before assigning $o_1$ a new vicinity region (together with new query segments), the server should additionally check whether $o_1$ leaves the original query segment $qs[v_5, v_6]$ by accessing $N_{211}.FL$ because, if so, $o_1$ may affect the current result of the query $q_1$. In this example, $o_1$ does not leave $qs[v_5, v_6]$. Please note that if the query segment $qs[p_4, p_6]$ is generated and assigned to $o_1$, $o_1$ should send two notification messages to the server, one that it has leaved $N_{211}$ and the other that it has leaved $qs[p_4, p_6]$.

#### 3.2.2. Vicinity Region Search

When a new moving object $o$ is registered at the server, an algorithm SEARCH is performed on the SSP-tree to find out $o$'s vicinity region $VR(o)$. Algorithm 2 is the pseudocode of SEARCH. Given a SSP-tree node $N$ (initially the root) and a moving object $o$ with the point of its current location $o.p$ and its capability $o.cap$, SEARCH recursively accesses only the nodes that contain $o.p$ until reaching the node $N$ such that $N.C \leq o.cap$ (lines 1–3). Now, $N$ becomes $o$'s vicinity region. Then, SEARCH invokes SEARCHSEG, which is a recursive depth-first search function that takes $N$ as an input (line 6). SEARCHSEG retrieves the tuples stored in each $N$'s descendent leaf node $\acute{N}$ and returns all the distinct query segments.

**Algorithm 2 SEARCH**$(N, o)$

**Input** $N$: a SSP-tree node initially set to the root, $o$: a moving object
**Output** $VR(o)$: $o$'s vicinity region, $qs\_set$: a set of distinct query segments
1: **if** $N.C > o.cap$ **then**
2: find the entry $(ptr, \acute{N})$ stored in $N$ such that $\acute{N}$ contains $o.p$;
3: SEARCH$(\acute{N}, o)$; // *Recursion*
4: **else** // if $N.C \leq o.cap$
5: set $VR(o)$ to $N$;
6: set $qs\_set$ to $qs\_set \cup$ SEARCHSEG$(N)$;
7: return $VR(o)$ and $qs\_set$;
**Function** SEARCHSEG$(N)$
**Input** $N$: a SSP-tree node that becomes $VR(o)$
**Output** *Result*: a set of distinct query segments
1: **if** $N$ is a non-leaf node **then**
2: **for** each entry $(ptr, \acute{N})$ stored in $N$ **do**
3: SEARCHSEG$(\acute{N})$; // *Recursion*
4: **else** // if $N$ is a leaf node
5: retrieve tuples stored in $N$ and insert the query segments into *Result*;
6: return *Result*;

After Algorithm 2 terminates, the server sends an AssignVR message to $o$ for assigning $N$ as $VR(o)$ together with the retrieved query segments. For example, let us assume that the capability $o_2.cap$ of the moving object $o_2$ in Figure 6 is 4. When $o_2$ is registered at the server, starting from the root, SEARCH recursively traverses the SSP-tree until it reaches the node $N_{21}$. Then, SEARCH invokes SEARCHSEG to find out all the distinct query segments stored in $N_{21}$'s descendent leaf nodes (i.e., $N_{211}$ and $N_{212}$). After SEARCH terminates, the server sends an AssignVR message to $o_2$ for assigning $N_{21}$ as $VR(o_2)$ together with the query segments $qs[v_6, p_1]$, $qs[v_6, p_6]$, $qs[v_6, p_7]$, and $qs[p_4, p_5]$.

### 3.2.3. SSP-Tree Manipulations

The SSP-tree can be manipulated with a set of algorithms, which specify how a query segment is inserted into and deleted from the SSP-tree, and how overflow or underflow of a SSP-tree node is managed.

**Algorithm 3 INSERT**($N$, $qs$)

**Input** $N$: a SSP-tree node initially set to the root, $qs$: a query segment of a query $q$

```
1:  update QT and QRT;
2:  if N is a non-leaf node then
3:    if qs fully intersects N then
4:      insert qs into N.FL;
5:    for each entry (ptr, Ń) stored in N do
6:      if qs intersects Ń then  // if qs is inside, partially intersects, or fully intersects Ń
7:        INSERT(Ń, qs);
8:  else  // if N is a leaf node
9:    if qs fully intersects N then
10:     insert qs into N.FL;
11:   else  // if qs is inside or partially intersects N
12:     if the tuple (qs ∩ N, flag) is not already stored in N then
13:       insert a tuple (qs ∩ N, flag) into N and increases N.C by 1;
14:       if qs partially intersects N then
15:         update QT, QRT, and SRT;
16:       repeat
17:         set N̈ to N's parent;
18:         increase N̈.C by 1;
19:       until (N̈ is the root)
20:       if N.C > θ then
21:         SPLITNODE(N);
```

Algorithm 3 is the pseudocode of the insert algorithm INSERT. Given an original query segment $qs$ of a query $q$, INSERT first updates QT and QRT (line 1). Then, INSERT recursively follows the paths of the SSP-tree, each of which consists of non-leaf and leaf nodes with which $qs$ intersects. At a non-leaf node $N$ in each path, INSERT checks if $qs$ fully intersects $N$. If so, INSERT inserts $qs$ into $N.FL$ (lines 3–4). When reaching a leaf node $N$ in the path, INSERT checks if $qs$ fully intersects $N$. If this is the case, INSERT inserts $qs$ into $N.FL$ (lines 9–10). On the other hand, if $qs$ is inside or partially intersects $N$, INSERT inserts a new tuple $(qs \cap N, flag)$ into $N$ and increases $N.C$ by 1 only when it is not already stored in $N$ (lines 12–13). Here, if $qs$ is inside $N$, i.e., $qs \cap N = qs$, $flag$ is set. In case that $qs$ partially intersects $N$, INSERT additionally updates QT, QRT, and SRT (lines 14–15). Finally, INSERT increases $\ddot{N}.C$ of each node $\ddot{N}$ in the path from $N$'s parent to the root by 1 (lines 16–19). When $N$ overflows (i.e., $N.C > \theta$), INSERT invokes the split algorithm SPLITNODE (lines 20–21).

Algorithm 4 is the pseudocode of SPLITNODE. Given an overflowed leaf node $N$, SPLITNODE creates (i) two new empty leaf nodes $N_{left}$ and $N_{right}$, and (ii) a new non-leaf node $N_{new}$ that stores entries $(ptr, N_{left})$ and $(ptr, N_{right})$, where $N_{left}$ or $N_{right}$ represents one of the equal halves of $N$ (lines 1–3). Now, $N_{left}$ and $N_{right}$ become $N_{new}$'s children. Next, SPLITNODE inserts all the query segments stored in $N.FL$ (i.e., all the original query segments that fully intersect $N$) into $N_{left}.FL$, $N_{right}.FL$, and $N_{new}.FL$, after which it finds the entry $(ptr, N)$ stored in $N$'s parent to redirect $ptr$ to point to $N_{new}$ (lines 4–5). Now, $N$'s parent becomes $N_{new}$'s parent. Then, SPLITNODE checks for each tuple $(qs, flag)$ stored in $N$ if $qs$ intersects each of $N_{new}$'s children $\acute{N}_{new}$. (Note: For the ease of description, we use $\acute{N}_{new}$ to denote both $N_{left}$ and $N_{right}$ when necessary.) If so, according to two cases, SPLITNODE proceeds as follows:

1. If $flag$ is set, indicating that $qs$ is the original query segment, SPLITNODE checks if $qs$ fully intersects $\acute{N}_{new}$. If so, SPLITNODE inserts $qs$ into $\acute{N}_{new}$ (lines 10–11). Otherwise, SPLITNODE inserts a tuple $(qs \cap \acute{N}_{new}, flag)$ into $\acute{N}_{new}$ and increases $\acute{N}_{new}.C$ by 1 only when

$(qs \cap \acute{N}_{new}, flag)$ is not already stored in $\acute{N}_{new}$ (lines 12–13). In case that $qs$ partially intersects $\acute{N}_{new}$, SPLITNODE additionally updates QT, QRT, and SRT (lines 14–15).

2. If $flag$ is not set, SPLITNODE checks for each of $qs$'s relevant original query segments $\acute{qs}$ if $\acute{qs}$ fully intersects $\acute{N}_{new}$. If so, SPLITNODE inserts $\acute{qs}$ into $\acute{N}_{new}.FL$ (lines 18–19). Otherwise, similarly to the former case, SPLITNODE inserts a tuple $(\acute{qs} \cap \acute{N}_{new}, flag)$ into $\acute{N}_{new}$ and increases $\acute{N}_{new}.C$ by 1 only when $(\acute{qs} \cap \acute{N}_{new}, flag)$ is not already stored in $\acute{N}_{new}$ (lines 20–21). In case that $\acute{qs}$ partially intersects $\acute{N}_{new}$, SPLITNODE additionally updates QT, QRT, and SRT (lines 22–23).

Then, SPLITNODE sets $N_{new}.C$ to $(N_{left}.C + N_{right}.C)$ and increases $\ddot{N}.C$ of each node $\ddot{N}$ in the path from $N_{new}$'s parent to the root by $(N_{new}.C - N.C)$ (lines 24–28). Finally, SPLITNODE discards $N$ (line 29). This split process propagates downward if necessary (lines 30–31).

**Algorithm 4 SPLITNODE($N$)**

**Input** $N$: an overflowed SSP-tree leaf node
1: create two new empty leaf nodes $N_{left}$ and $N_{right}$;
2: create a new empty non-leaf node $N_{new}$;
3: insert entries $(ptr, N_{left})$ and $(ptr, N_{right})$ into $N_{new}$;
4: insert all the query segments stored in $N.FL$ into $N_{left}.FL$, $N_{right}.FL$, and $N_{new}.FL$;
5: find the entry $(ptr, N)$ stored in $N$'s parent and redirect $ptr$ to point to $N_{new}$;
6: **for** each entry $(ptr, \acute{N}_{new})$ stored in $N_{new}$ **do** // $\acute{N}_{new} : N_{left}$ or $N_{right}$
7: **for** each tuple $(qs, flag)$ stored in $N$ **do**
8: **if** $qs$ intersects $\acute{N}_{new}$ **then** // if $qs$ is inside, partially intersects, or fully intersects $\acute{N}_{new}$
9: **if** $flag$ is set **then**
10: **if** $qs$ fully intersects $\acute{N}_{new}$ **then**
11: insert $qs$ into $\acute{N}_{new}.FL$;
12: **else** // if $qs$ is inside or partially intersects $\acute{N}_{new}$
13: insert a tuple $(qs \cap \acute{N}_{new}, flag)$ into $\acute{N}_{new}$ and increase $\acute{N}_{new}.C$ by 1 if it is not stored in $\acute{N}_{new}$;
14: **if** $qs$ partially intersects $\acute{N}_{new}$ **then**
15: update QT, QRT, and SRT;
16: **else** // $flag$ is not set
17: **for** each $qs$'s relevant original query segment $\acute{qs} \in \acute{qs}\_list$ of the tuple ($qs$, $\acute{qs}\_list$) in SRT **do**
18: **if** $\acute{qs}$ fully intersects $\acute{N}_{new}$ **then**
19: insert $\acute{qs}$ into $\acute{N}_{new}.FL$;
20: **else** // if $\acute{qs}$ is inside or partially intersects $\acute{N}_{new}$
21: insert a tuple $(\acute{qs} \cap \acute{N}_{new}, flag)$ into $\acute{N}_{new}$ and increase $\acute{N}_{new}.C$ by 1 if it is not stored in $\acute{N}_{new}$;
22: **if** $\acute{qs}$ partially intersects $\acute{N}_{new}$ **then**
23: update QT, QRT, and SRT;
24: set $N_{new}.C$ to $(N_{left}.C + N_{right}.C)$;
25: **repeat**
26: set $\ddot{N}$ to $N_{new}$'s parent;
27: increase $\ddot{N}.C$ by $(N_{new}.C - N.C)$;
28: **until** ($\ddot{N}$ is the root)
29: discard $N$;
30: **for** each entry $(ptr, \acute{N}_{new})$ stored in $N_{new}$ **do** // $\acute{N}_{new} : N_{left}$ or $N_{right}$
31: SPLITNODE($\acute{N}_{new}$) if $\acute{N}_{new}.C > \theta$;

Algorithm 5 is the pseudocode of the delete algorithm DELETE. Given an original query segment $qs$ of a query $q$, DELETE first updates QRT (line 1). Then, DELETE follows the paths of the SSP-tree, each of which consists of non-leaf and leaf nodes with which $qs$ intersects. At a non-leaf node $N$ in each path, DELETE checks if $qs$ fully intersects $N$. If so, DELETE deletes $qs$ from $N.FL$ (lines 3–4). When reaching a leaf node $N$ in this path, DELETE checks the intersection relationships between $qs$ and $N$. Then, according to three cases, DELETE proceeds as follows:

1. If $qs$ fully intersects $N$, DELETE $qs$ from $N.FL$ (lines 10–11).
2. If $qs$ is inside $N$ (i.e., $qs \cap N = qs$), DELETE deletes the tuple $(qs \cap N, flag)$ from $N$ and decreases $N.C$ by 1 if $q$ is the only relevant query of $qs \cap N$ (lines 12–14).
3. If $qs$ partially intersects $N$, DELETE updates QRT and SRT (line 16). Then, it deletes the tuple $(qs \cap N, flag)$ from $N$ and decrease $N.C$ by 1 if $qs$ is the only relevant original query segment of $qs \cap N$ (lines 17–18).

If the tuple $(qs \cap N, flag)$ is deleted from $N$, DELETE decreases $\ddot{N}.C$ of each node $\ddot{N}$ in the path from its parent to the root by 1 (lines 20–23). Finally, DELETE invokes the merge algorithm MERGENODE, which takes $N$'s parent as an input, to condense the tree if possible (line 24).

**Algorithm 5 DELETE**($N$, $qs$)

**Input** $N$: a SSP-tree node initially set to the root, $qs$: a query segment of a query $q$
1: update QRT;
2: **if** $N$ is a non-leaf node **then**
3: **if** $qs$ fully intersects $N$ **then**
4: delete $qs$ from $N.FL$;
5: **else** // if $qs$ is inside or partially intersects $N$
6: **for** each entry $(ptr, \acute{N})$ stored in $N$ **do**
7: **if** $qs$ intersects $\acute{N}$ **then** // if $qs$ is inside, partially intersects, or fully intersects $\acute{N}$
8: DELETE($\acute{N}$, $qs$);
9: **else** // if $N$ is a leaf node
10: **if** $qs$ fully intersects $N$ **then**
11: delete $qs$ from $N.FL$;
12: **else if** $qs$ is inside $N$ **then** // if $qs \cap N = qs$
13: **if** $q$ is the only relevant query of $qs \cap N$ (= $qs$) **then**
14: delete the tuple $(qs \cap N, flag)$ from $N$ and decrease $N.C$ by 1;
15: **else** // if $qs$ partially intersects $N$
16: update QRT and SRT;
17: **if** $qs$ is the only relevant original query segment of $qs \cap N$ **then**
18: delete the tuple $(qs \cap N, flag)$ from $N$ and decrease $N.C$ by 1;
19: **if** the tuple $(qs \cap N, flag)$ is deleted from $N$ **then**
20: **repeat**
21: set $\ddot{N}$ to $N$'s parent;
22: decrease $\ddot{N}.C$ by 1;
23: **until** ($\ddot{N}$ is the root)
24: MERGENODE($N$'s parent);

**Algorithm 6 MERGENODE**($N$)

**Input** $N$: a non-leaf node of the SSP-tree
1: **if** $N.C \leq \theta$ and both of $N$'s children are leaf nodes **then**
2: create a new empty leaf node $N_{new}$;
3: insert all the original query segments stored in $N.FL$ into $N_{new}.FL$;
4: set $N_{new}.C$ to $N.C$;
5: find the entry $(ptr, N)$ stored in $N$'s parent and redirect $ptr$ to point to $N_{new}$;
6: **for** each entry $(ptr, \acute{N})$ stored in $N$ **do**
7: **for** each tuple $(qs \cap \acute{N}, flag)$ stored in $\acute{N}$ **do**
8: insert $(qs \cap \acute{N}, flag)$ into $N_{new}$ if it is not stored in $N_{new}$;
9: discard $N$ and $N$'s children;
10: MERGENODE($N_{new}$'s parent);

Algorithm 6 is the pseudocode of MERGENODE. Given a non-leaf node $N$, MERGENODE first checks if (i) $N.C \leq \theta$ and (ii) both of $N$'s children are leaf nodes. If so, MERGENODE creates a new empty leaf node $N_{new}$, inserts all the original query segments stored in $N.FL$ into $N_{new}.FL$, and sets $N_{new}.C$ to $N.C$ (lines 2–4). Next, MERGENODE finds the entry $(ptr, N)$ stored in $N$'s parent to redirect $ptr$ to point to $N_{new}$ (line 5). Now, $N$'s parent becomes $N_{new}$'s parent. Then, MERGENODE inserts all

the distinct tuples stored in each $N$'s child $\acute{N}$ into $N_{new}$ (lines 6–8). Finally, MERGENODE discards $N$ and its children (line 9). This merge process propagates upward until the node that does not satisfy the merge condition is reached (line 10).

Now, we analyze the time costs of the SSP-tree manipulations in Lemma 5. For the simplicity of analysis, we assume (i) that each query segment is not redundantly stored in two leaf nodes; and (ii) that the SSP-tree is perfectly balanced, and thus its depth is $\log_2(\frac{|QS\_set|}{\theta})$, where $|QS\_set|$ denotes the total number of query segments stored in the leaf nodes. In addition, the number of original query segments $\acute{qs}\_list$ of each tuple $(qs, \acute{qs}\_list)$ in SRT is assumed to be almost same.

**Lemma 5.** *Let $t_{insert}$, $t_{split}$, $t_{delete}$, and $t_{merge}$ be the time costs of insert, split, delete, and merge operations, respectively. Then, $t_{insert} \approx \alpha_1 \log_2(\frac{|QS\_set|}{\theta}) + \alpha_2 \log_2(\frac{|QS\_set|}{\theta})$, $t_{split} \approx 2 \cdot \alpha_3(\theta+1) \cdot |\acute{qs}\_list| + \alpha_4 \log_2(\frac{|QS\_set|}{\theta})$, $t_{delete} \approx \alpha_5 \log_2(\frac{|QS\_set|}{\theta}) + \alpha_6 \log_2(\frac{|QS\_set|}{\theta})$, and $t_{merge} \approx \alpha_7\theta$, where $\alpha_1, \alpha_2, \ldots, \alpha_7$ are constants.*

**Proof.** Given a new query segment $qs$, INSERT involves finding the leaf node $N$ from the root to store $qs$ and updating the count variables of the non-leaf nodes along the path from $N$ to the root, both of which take time linear to the depth of the SSP-tree, and thus $t_{insert} \approx \alpha_1 \log_2(\frac{|QS\_set|}{\theta}) + \alpha_2 \log_2(\frac{|QS\_set|}{\theta})$. Please note that (i) updating full list $N.FL$ maintained in each node $N$ and (i) updating QT, QRT, and SRT take constant expected time because they are implemented as hash tables. Given an overflowed leaf node $N$, SPLITNODE checks $\theta+1$ tuples against each of two newly generated leaf nodes $\acute{N}_{new}$. (Note: $\acute{N}_{new}$ is a child of the newly generated non-leaf node $N_{new}$.) In addition, for each tuple $(qs, flag)$, if $flag$ is not set, SPLITNODE further checks $\acute{qs}\_list$ of the tuple $(qs, \acute{qs}\_list)$ in SRT. These take time at most $2 \cdot \alpha_3(\theta+1) \cdot |\acute{qs}\_list|$. Because SPLITNODE also involves updating the count variables of the non-leaf nodes along the path from $N_{new}$ to the root, $t_{split} \approx 2 \cdot \alpha_3(\theta+1) \cdot |\acute{qs}\_list| + \log_2(\frac{|QS\_set|}{\theta})$. To delete an existing query segment $qs$, DELETE finds the leaf node $N$ from the root, deletes $qs$ if necessary, and updates the count variables of the non-leaf nodes along the path from $N$ to the root; therefore, $t_{delete} \approx \alpha_5 \log_2(\frac{|QS\_set|}{\theta}) + \alpha_6 \log_2(\frac{|QS\_set|}{\theta})$. Finally, MERGENODE checks each tuple $(qs, flag)$ stored in two mergeable leaf nodes against a new leaf node, and thus $t_{merge} \approx \alpha_7\theta$. □

### 3.3. Vicinity Region-based Communications and Query Processing

In this subsection, we describe how each moving object and the server communicate each other to cooperatively process range monitoring queries. The query processing consists of server-side tasks and object-side tasks.

#### 3.3.1. Server-Side Tasks

The server performs three main tasks: query registration, region assignment, and query result update.

- **Query registration:** When a new query $q$ is issued by a client, the server inserts $q$ into QT and invokes COMPUTESEG (see Algorithm 1) to compute the relevant query segments of $q$. Then, for each query segment $qs$ generated by COMPUTESEG, the server (i) invokes INSERT (see Algorithm 3) to insert $qs$ into the SSP-tree and (ii) broadcasts the $\mathsf{InsertQS}(qs)$ message. In case that a leaf node $\acute{N}$ of the SSP-tree is split into $N_{left}$ and $N_{right}$, the server broadcasts the $\mathsf{SplitRegion}(\acute{N}, N_{left}, N_{right})$ message. On the other hand, when an existing query $q$ is terminated by a client, for each relevant query segment $qs$ of $q$, the server (i) invokes DELETE (see Algorithm 5) to delete $qs$ from the SSP-tree; (ii) broadcasts the $\mathsf{DeleteQS}(qs)$ message; and finally (iii) deletes $q$ from QT. In case that two leaf nodes $N_{left}$ and $N_{right}$ are merged, the server broadcasts the $\mathsf{MergeRegion}(N_{left}, N_{right}, qs\_set_{(left+right)})$ message, where $qs\_set_{(left+right)}$ is the combined set of query segments inside $N_{left}$ and $N_{right}$.

- **Region assignment:** When the server receives the RequestVR$(o.p, o.cap, N_{old})$ message from a moving object $o$, where $o.p$, $o.cap$, and $N_{old}$ denote the point of $o$'s current location, $o$'s capability, and $o$'s previous vicinity region, respectively, it first checks if $o$ leaves each original query segment stored in $N_{old}.FL$. If so, the server updates the result of the corresponding query when necessary. Then, the server invokes SEARCH (see Algorithm 2), after which it sends the AssignVR$(N_{new}, qs_set)$ message to $o$, where $N_{new}$ is $o$'s new vicinity region and $qs\_set$ is a set of query segments inside $N_{new}$.
- **Query Result Update:** When the server receives the UpdateResult$(o.p, qs)$ message from a moving object $o$, it visits QRT to find the relevant queries of $qs$. Then, the server checks each $qs$'s relevant query $q$ if $dist_N(q.p, o.p) \leq q.d$. If so, the server inserts $o$ into the result of $q$ (if it is not already there). On the other hand, if $dist_N(q.p, o.p) > q.d$, the server removes $o$ from the result of $q$ (if it is there).

#### 3.3.2. Object-Side Tasks

Each moving object $o$ maintains a subspace $N$ as its vicinity region and a set $qs\_set$ of query segments inside $N$. Whenever $o$ moves, it checks (i) if it leaves $N$ and (ii) it enters or leaves each query segment $qs \in qs\_set$. If $o$ leaves $N$, it sends the RequestVR$(o.p, o.cap, N_{old})$ message to the server. In response, $o$ receives the AssignVR$(N_{new}, qs_set)$ message from the server. On the other hand, if $o$ enters or leaves $qs$, it sends the UpdateResult$(o.p, qs)$ message to the server. In addition, $o$ expects the following broadcast messages from the server and processes them as follows:

- InsertQS$(qs)$: When $o$ listens to the InsertQS$(qs)$ message, it first checks if $o.p$ is inside $qs$. If so, $o$ sends UpdateResult$(o.p, qs)$ message to the server to let the server insert $o$ into the results of $qs$'s relevant queries. Then, $o$ checks if $qs$ is inside or partially intersects its vicinity region $N$. If so, $o$ inserts $N \cap qs$ into $qs\_set$. In case that the cardinality $|qs\_set|$ of $qs\_set$ is greater than $o.cap$, i.e., $|qs\_set| > o.cap$, $o$ sends the RequestVR$(o.p, o.cap, N_{old})$ to the server.
- SplitRegion$(\acute{N}, N_{left}, N_{right})$: When $o$ listens to the SplitRegion$(\acute{N}, N_{left}, N_{right})$ message, it first checks if its vicinity region $N$ equals $\acute{N}$. If this is the case, if $o.p$ is inside $N_{left}$, $o$ sets $N_{left}$ as its new vicinity region $N_{new}$; otherwise, it sets $N_{right}$ as $N_{new}$. Then, for each query segment $qs$ ($\in qs\_set$) that intersects $N_{new}$, $o$ replaces it with $qs \cap N_{new}$.
- DeleteQS$(qs)$: When $o$ listens to the DeleteQS$(qs)$ message, it just deletes $qs$ from $qs\_set$ if $qs$ is inside its vicinity region $N$.
- MergeRegion$(N_{left}, N_{right}, qs_set)$: When $o$ listens to the MergeRegion$(N_{left}, N_{right}, qs_set_{(left+right)})$ message, it first checks if its vicinity region $N$ equals $N_{left}$ or $N_{right}$. If so, $o$ sets $N_{left} \cup N_{right}$ as its new vicinity region $N_{new}$ and replaces $qs\_set$ with $qs\_set_{(left+right)}$.

## 4. Performance Evaluation

In this section, we evaluate and compare the performance of the proposed distributed method (denoted by DM) with that of the centralized method (denoted by CM) in terms of the communication cost and the server computation cost because existing work for range monitoring queries in the road network mostly adopts CM [1]. CM uses the same basic data structures (i.e., EI, ET, and QT) as DM. The communication cost of CM was measured by the total number of location-update messages transmitted from moving objects, while that of DM was measured by the total number of messages transmitted between the server and moving objects, i.e., the sum of (i) the number of point-to-point messages (RequestVR, UpdateResult, and AssignVR) and (ii) the number of broadcast messages (InsertQS, SplitRegion, DeleteQS, and MergeRegion). On the other hand, the server computation cost of CM and DM was measured by the amount of CPU-time the server takes for query processing. Because we are interested in the advantages of maintaining the full list at each SSP-tree node, we implemented two versions of the SSP-tree; one is the proposed SSP-tree, where each node maintains the full list, and the other is the naïve SSP-tree, where each node does not maintain the full list. The simulations

were coded in Java on Intel Xeon E5-2620 6-core Processor with 8GB RAM running Linux Ubuntu 12.04 (64-bit) operating system.

### 4.1. Simulation Setup

Each set of simulations was conducted on a real road network of the city of Oldenburg in Germany (normalized to $[0, 10{,}000]^2$), which consists of 6105 vertices and 7035 edges, and is obtained from https://iapg.jade-hs.de/personen/brinkhoff/generator/ [15]. The number of queries whose query points are uniformly placed on the road network is varied from 1000 to 10,000, and the query distance of each query is varied from 50 to 500. On the other hand, the number of moving objects is varied from 10,000 to 100,000, and the minimum computational capability of each moving object is varied from 10 to 100. The movement of each moving object follows the *random waypoint model* [16], which is one of the most widely used mobility models: each moving object $o$ chooses a random point of destination on the road network and moves along the shortest path to the destination at a constant speed distributed uniformly from 0 to the maximum speed, which we set to 50 per simulation time step. Upon reaching the destination, $o$ remains stationary for a certain period of time. When this period expires, $o$ chooses a new destination and repeats the same process during the entire simulation time steps. We list the set of used parameters and their default values (stated in boldface) in the simulations in Table 2. In each simulation, we evaluated the effect of one parameter while the others were fixed at their default values. We ran each simulation for 1000 simulation time steps and measured the average of (i) the total number of messages transmitted and (ii) CPU-time (in ms) consumed at each simulation time step. We set each object in CM sends the location-update message to the server at each simulation time step. We set 5% of queries to be updated (i.e., reinserted after they are deleted) at each simulation time step. Please note that this update rate is sufficient to study the performance of the proposed method because we focus on the queries with the static query points.

**Table 2.** Simulation parameters and their values.

| Simulation Parameter | Value Used (Default) |
|---|---|
| Number of queries | 1000∼10,000 (**5000**) |
| Query distance | 50∼500 (**250**) |
| Number of moving objects | 10,000∼100,000 (**50,000**) |
| Minimum computational capability | 10∼100 (**50**) |

### 4.2. Simulation Results

#### 4.2.1. Effect of the Number of Queries

In the first simulation, we varied the number of queries from 1000 to 10,000 and studied the effect of the number of queries on the communication cost and the server computation cost. Figure 7 shows the effect of the number of queries on (i) the communication cost (i.e., the total number of messages communicated between the server and moving objects) and (ii) the server computation cost (i.e., the amount of CPU-time the server takes for query processing). As shown in Figure 7a, the number of queries does not affect the communication cost of CM because, in CM, moving object periodically send location-updates to the server. On the other hand, as the number of queries is increased, the communication cost of DM is also increased. However, DM clearly outperforms CM because, in DM, moving objects utilize their computational capabilities for sending messages to the server only when necessary, i.e., when they (i) leave their current vicinity regions or (ii) enter or leave any of the assigned query segments. This figure also shows that DM with the SSP-tree (denoted by $\text{DM}_{(\text{SSP-tree})}$) performs much better than DM with the naïve SSP-tree (denoted by $\text{DM}_{(\text{naïve SSP-tree})}$). This is because by using the SSP-tree, where each node maintains the full list, the server can assign larger vicinity regions to moving objects than that using the naïve SSP-tree, and thus the moving objects can reduce the number of sending RequestVR messages to the server for receiving new vicinity

regions (see the first advantage of the full list describe in Section 3.2). Please note that assigning larger vicinity regions to the moving objects makes them not frequently leave their current vicinity regions. In addition, the moving objects in $DM_{(SSP\text{-}tree)}$ can also reduce the number of sending unnecessary UpdateResult messages to the server (see the second advantage of the full list describe in Section 3.2). As compared to CM and $DM_{(naïve\ SSP\text{-}tree)}$, on average, $DM_{(SSP\text{-}tree)}$ incurs 12.6% and 66.2%, respectively, of the communication cost.

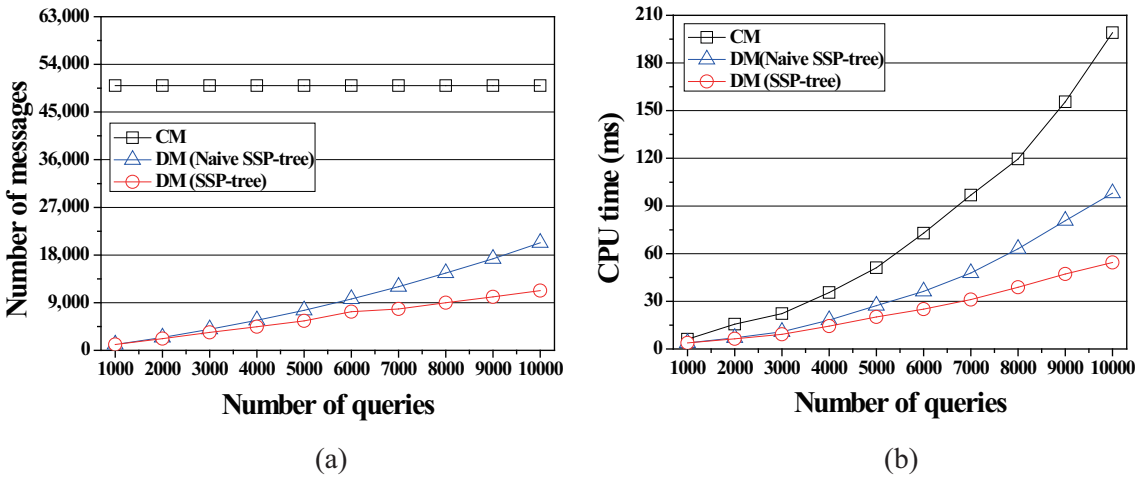


**Figure 7.** Effect of the number of queries on the communication cost and the server computation cost. (**a**) Total number of messages vs. the number of queries; (**b**) The amount of CPU-time vs. the number of queries.

As shown in Figure 7b, the performance of all methods in terms of the server computation cost degrades as the number of queries is increased. However, as expected, CM performs much worse than $DM_{(naïve\ SSP\text{-}tree)}$ and $DM_{(SSP\text{-}tree)}$ because, in CM, the server checks all moving objects if they affect the current results of all queries. It is also observed from this figure that $DM_{(SSP\text{-}tree)}$ outperforms $DM_{(naïve\ SSP\text{-}tree)}$. In both methods, the amount of CPU-time the server takes is mainly affected by the search process for assigning vicinity regions to moving objects. As already mentioned above, because of the first advantage of the full list, the SSP-tree helps the server assign larger vicinity regions to the moving objects than the naïve SSP-tree does. As a result, the server in $DM_{(SSP\text{-}tree)}$ can reduce more CPU-time than that in $DM_{(naïve\ SSP\text{-}tree)}$. As compared to CM and $DM_{(naïve\ SSP\text{-}tree)}$, on average, $DM_{(SSP\text{-}tree)}$ takes 32.3% and 63.7%, respectively, of the amount of CPU-time.

#### 4.2.2. Effect of the Query Distance

In this simulation, we varied the query distance of the range monitoring queries from 50 to 500 to examine how the query distance affects the performance of the proposed method. As shown in Figure 8a, the query distance does not affect the communication cost of CM because of the same reason mentioned in the first simulation. On the other hand, as the query distance is increased, the communication cost of $DM_{(naïve\ SSP\text{-}tree)}$ and $DM_{(SSP\text{-}tree)}$ is increased. As the query distance becomes longer, excessive intersections among query segments occur. This increases the number of node splits of the naïve SSP-tree and the SSP-tree, and thus accelerates the height growth of the naïve SSP-tree and the SSP-tree. As a result, the servers in $DM_{(naïve\ SSP\text{-}tree)}$ and $DM_{(SSP\text{-}tree)}$ are led to assign smaller vicinity regions to moving objects. Because the moving objects are assigned the vicinity regions of small size, they frequently leave these vicinity regions and send RequestVR messages to the server. However, $DM_{(SSP\text{-}tree)}$ performs better and is less sensitive to this parameter than $DM_{(naïve\ SSP\text{-}tree)}$ because, again, of the first advantage of maintaining the full list at each node in the

SSP-tree. On average, $DM_{(SSP\text{-}tree)}$ incurs 12.1% and 51.4% of the communication cost, as compared to CM and $DM_{(naïve\ SSP\text{-}tree)}$, respectively.

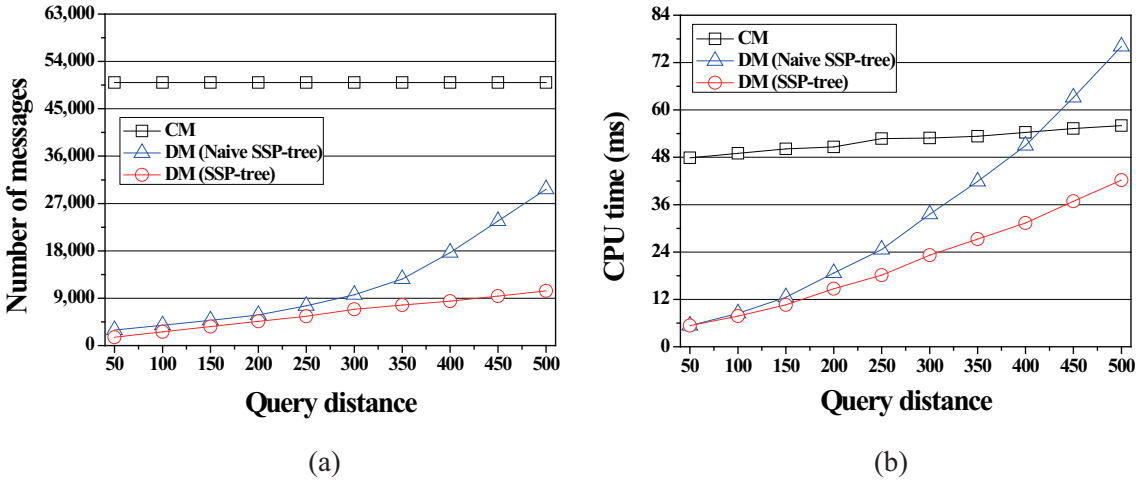


**Figure 8.** Effect of the query distance on the communication cost and the server computation cost. (**a**) Total number of messages vs. the query distance; (**b**) The amount of CPU-time vs. the query distance.

Figure 8b shows the effect of the query distance on the server computation cost. In contrast to $DM_{(naïve\ SSP\text{-}tree)}$ and $DM_{(SSP\text{-}tree)}$, the server computation cost of CM is nearly not affected by the query distance because an increase in the query distance does not increases the amount of CPU-time the server takes for checking moving objects if they affect the current results of queries. However, $DM_{(SSP\text{-}tree)}$ still performs best in all cases. As compared to CM and $DM_{(naïve\ SSP\text{-}tree)}$, on average, $DM_{(SSP\text{-}tree)}$ takes 41.7% and 64.9%, respectively, of the amount of CPU-time.

### 4.2.3. Effect of the Number of Moving Objects

In this simulation, we varied the number of moving objects from 10,000 to 100,000 to study how the number of moving objects affects the performance of the proposed method.

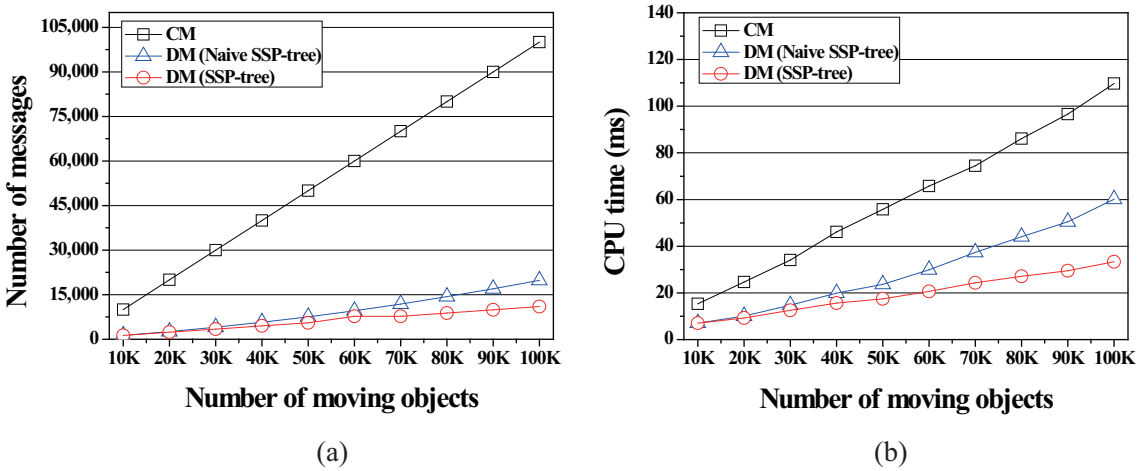


**Figure 9.** Effect of the number of moving objects on the communication cost and the server computation cost. (**a**) Total number of messages vs. the number of moving objects; (**b**) The amount of CPU-time vs. the number of moving objects.

As shown in Figure 9, as the number of moving objects is increased, the overhead of all methods is increased in terms both of the communication cost and the server computation cost. The communication cost of CM is proportional to the number of moving objects because they periodically send location-updates to the server. In contrast, the communication cost of $DM_{(naïve\ SSP\text{-}tree)}$ and $DM_{(SSP\text{-}tree)}$ is slightly increased due to the benefits of utilizing the computational capabilities of moving objects. Similarly, the server computation cost of $DM_{(naïve\ SSP\text{-}tree)}$ and $DM_{(SSP\text{-}tree)}$ is less sensitive to the number of moving objects than CM. We can observe from Figure 9 that $DM_{(SSP\text{-}tree)}$ performs best in terms both of the communication cost and the server computation cost.

#### 4.2.4. Effect of the Minimum Computational Capability

In this simulation, we varied the value of the minimum computational capability of each moving object to study how the value of $\theta$ affects the performance of the proposed method. The value of $\theta$ indicates (i) the minimum number of query segments each moving object should process and (ii) the split threshold of the naïve SSP-tree and the SSP-tree. As shown in Figure 10, both the communication cost and the server computation cost of CM are not affected by this parameter at all because CM does not utilizes the computational capabilities of moving objects. On the other hand, the performance of $DM_{(naïve\ SSP\text{-}tree)}$ and $DM_{(SSP\text{-}tree)}$ in terms both of the communication cost and the server computation cost is improved as the value of the minimum computational capability is increased. This is because a larger value of $\theta$ increases the average number of query segments each moving object should process, and thus the server can assign a larger vicinity region to each moving object. However, $DM_{(SSP\text{-}tree)}$ performs a lot better than $DM_{(naïve\ SSP\text{-}tree)}$. As compared to $DM_{(naïve\ SSP\text{-}tree)}$, on average, $DM_{(SSP\text{-}tree)}$ incurs 81.9% of the communication cost and takes 78.2% of the amount of CPU-time.

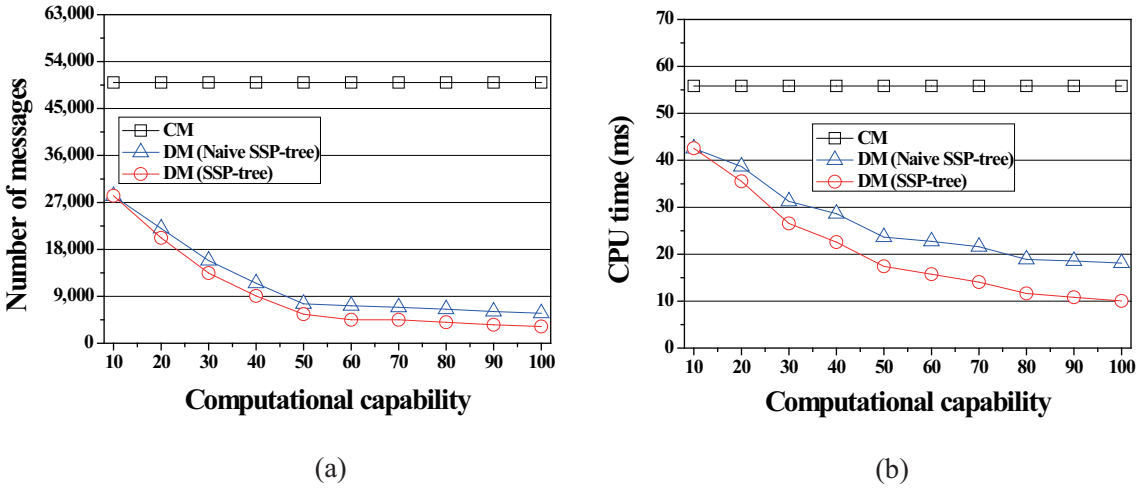


**Figure 10.** Effect of the minimum computational capability on the communication cost and the server computation cost. (**a**) Total number of messages vs. the minimum computational capability; (**b**) The amount of CPU-time vs. the minimum computational capability.

## 5. Discussion

Our work focuses on distributed processing of static range monitoring queries over moving objects. To utilize the computational capability $o.cap$ of each moving object $o$ by using the SSP-tree, we fix the system parameter $\theta$ in advance to the minimum number of query segments $o$ should process by assuming the server lets $o$ to select one of the predefined values as $o.cap$ when $o$ is registered at the server, so that $o.cap \geq \theta$. This is because when a new moving object $\acute{o}$ with its capability $\acute{o}.cap < \theta$ is registered at the server, the SSP-tree needs to be reconstructed from the scratch, similarly to the index structures used in the existing distributed methods for processing static range monitoring queries in Euclidean space [2,3]. Figure 11a shows the effect of the number of queries on the initial

construction time of the SSP-tree. When the number of queries is 5000 (the default value of this parameter), the construction time of the SSP-tree takes 8.7 s, which can be amortized by the long running time of the range monitoring queries if $\theta$ is fixed or is not frequently changed. However, for the case where $\theta$ is dynamically changed, more investigations should be done.

In addition, different from the centralized methods [4–7], in our work, moving objects may suffer from the additional computational burden of participating in query processing tasks. Figure 11b shows the effect of the minimum computational capability of each moving object $o$ on the amount of CPU-time consumed by $o$. It can be easily observed from the figure that as the value of the minimum computational capability is increased (i.e., as the value of $\theta$ is increased), the computational burden on $o$ is also increased. Therefore, there needs to be future consideration of how to enable the moving objects to voluntarily participate in query processing tasks so that the advantages of the proposed method can be fully realized.

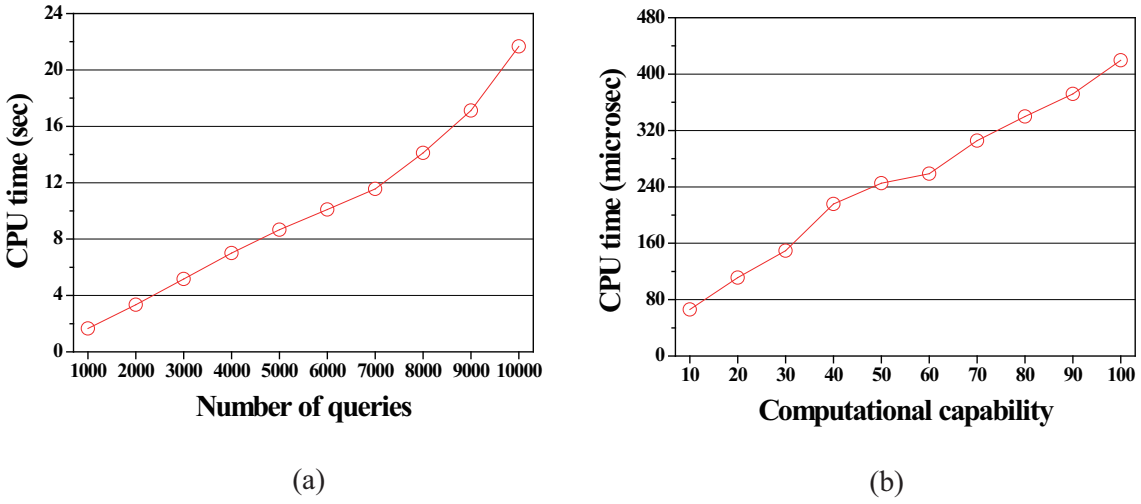


**Figure 11.** Construction time of the SSP-tree and CPU-time consumed by a moving object. (**a**) Construction time of the SSP-tree vs. the number of queries; (**b**) The amount of CPU-time consumed by a moving object in $DM_{(SSP\text{-}tree)}$ vs. the minimum computational capability; all other parameters have their default values (see the values stated in boldface in Table 2).

Finally, another issue that needs to be addressed is the energy efficiency of moving objects. With regard to identifying the location of moving objects, our work makes the same assumption that the centralized methods do, namely, that each moving object $o$ periodically measures its current location through GPS. However, GPS is an energy-intensive module, and thus its periodic usage can be a major energy drain for $o$. To achieve the energy efficiency of each moving object $o$, the strategy that makes $o$ allow the GPS module to sleep should be considered. This would entail developing the method for determining a proper sleep duration $\Delta t$ for $o$, so that $o$ can move freely without sensing and checking its current location against nearby queries during $\Delta t$.

## 6. Related Work

Most of the early researches on spatial databases assumed the static objects in Euclidean space and focused on (i) developing efficient spatial access methods (e.g., the R-tree [17] and its variants [18,19]) and (ii) processing of snapshot queries, which retrieves the results of queries only once at a specific snapshot in time. Papadias et al. incorporated the road network into the existing spatial databases and proposed two basic methods for processing several types of snapshot queries (e.g., range queries, nearest neighbor queries, spatial join queries, and closest pair queries) in the road network [20]. The first method is the *Euclidean Restriction* (*ER*). ER first identify the candidate objects

by their Euclidean distance from the query point, after which it discards the false positives based on their network distances from the query point. The second method is the *Network Expansion* (*NE*), which performs the search directly from the query point by gradually expanding the nearby vertices in the order of their network distances from the query point. Both ER and NE use the R-tree to speed up query processing. Zhong et al. proposed the *G-tree*, a hierarchy structure, for processing nearest neighbor queries in the road network [21]. Assuming only the broadcast communication is available, Sun et al. proposed the *Network Partition Index* (*NPI*) for processing range queries and nearest neighbor queries in the road network [22].

Later on, the focus was extended to indexing moving objects. Assuming that the trajectories of moving objects are known a priori or predictable, Saltenis et al. proposed the *Time-Parameterized R-tree* (*TPR-tree*) for indexing moving objects, where the location of each moving object is transformed into a linear function of time [23]. Tao et al. proposed the improved version of the TPR-tree, called the *TPR*$^*$*-tree*, which uses the exactly same data structure as the TPR-tree but applies new insert and delete algorithms [24]. However, the known-trajectory assumption does not hold for most real-life application scenarios (e.g., the velocity of a typical customer on the road are frequently changed), which leads those index structures to become prohibitively expensive to update. To deal with a large number of moving objects that move arbitrarily, Lee et al. proposed a generalized bottom-up update strategy for the R-tree [25]. Wang and Zimmermann introduced the dual index design for snapshot range queries over moving objects in the road network, which utilizes the R-tree to index the road network and the in-memory grid structure to index the points of moving objects' location [26].

Motivated by LBSs, another research direction has recently focused on processing monitoring queries. Many methods for monitoring queries have been proposed, which can be broadly classified into two categories according to the mobility of query points and objects. The first category focuses on static queries over moving objects, and the second category deals with moving queries over static/moving objects. Because our work belongs to the first category, we elaborate on the review of the representative methods in the first category and briefly review the methods in the second category. Indexing queries, instead of indexing frequently moving objects, has been considered to be an attractive strategy, which reduces the server computation cost for updating index structures because monitoring queries remain active for a long period of time and are static. Prabhakar et al. suggested to use the R-tree to index queries [5], while Kalashnkov et al. used the in-memory grid structure [6]. Wang and Roger extended [26] for processing monitoring queries in the road network [7]. These methods assume that moving objects periodically send location-updates to the server. The server, meanwhile, continually (i) receives the location-update stream; (ii) determines the queries that are affected by the movements of the objects; and (iii) updates their results if necessary. However, constant location-updates generated by a huge number of moving objects may incur significant communication bottleneck and greatly increase the overhead at the server for determining the affected queries and keeping their results up to date. In addition, because the transmission of a location-update message over a wireless connection takes a substantial amount of energy, the handheld device carried by each moving object exhausts its battery life quickly.

To help each moving object reduce the number of sending location-updates to the server, the *safe region* method was proposed in [4,5]. The safe region, assigned to each moving object $o$, is the area that (i) contains the point of $o$'s current location and (ii) guarantees that the current results of all the queries will remain valid as long as $o$ does not leave it. Therefore, $o$ need not send a location-update to the server as long as it does not leave its safe region. Although the safe region method improves the overall system performance to a certain degree, because the size of a safe region assigned to each object $o$ is typically small, $o$ easily leaves its current safe region and contacts the server in order to receive a new safe region. Recently, the distributed methods, namely the *Monitoring Query Management* (*MQM*) method [2] and the *Query Region-tree* (*QR-tree*) method [3] were proposed. Unfortunately, these distributed methods only deal with the objects moving freely in Euclidean space.

Focusing on processing moving monitoring queries over static objects, the safe region methods were also proposed in [8,9]. Similarly to the safe region assigned to a moving object, the safe region assigned to a query $q$ is the region that (i) contains $q.p$ and (ii) guarantees that while $q.p$ remains inside it, the result of $q$ remain unchanged. More recently, several algorithms were proposed to compute the *safe exits* of a query $q$ in the road network [10–12]. Safe exits are a set of points that guarantee the result of $q$ remains unchanged before $q.p$ reaches any of these points. There were also proposed methods for processing moving monitoring queries over moving objects in the road network [13,14]. Mouratidis et al. proposed the *Incremental Monitoring Algorithm* (*IMA*) and the *Group Monitoring Algorithm* (*GMA*) [13]. IMA is based on NE [20], while GMA extends IMA with the shared execution paradigm. Liu and Hua proposed the distributed method that utilizes the computational capabilities of moving objects [14]. However, this method is not comparable to our proposed method (i.e., $DM_{(SSP\text{-}tree)}$) because it assumes that there is no restriction on the number of queries the server can assign to moving objects. Specifically, in this method, although the number of queries a moving object $o$ can process is at most $n$, the server can assign $o$ more than $n$ queries. On the other hand, in our proposed method, the server must assign $o$ no more than $n$ query segments.

## 7. Conclusions

In this paper, we addressed the problem of the efficient processing of range monitoring queries over the objects moving along the road network. Given a set of geographically distributed moving objects on the road, the primary goal of our study is to keep the results of queries up to date, while incurring the minimum communication cost and server computation cost by letting the moving objects evaluate several queries that are relevant to them. To achieve this, we introduced the concept of vicinity region and proposed a new spatial index structure, namely the SSP-tree. By assigning each moving object (i) a vicinity region and (ii) a set of query segments inside this region, the moving object can locally monitor whether it may affect the results of nearby queries. The SSP-tree is used to efficiently search the appropriate vicinity regions for moving objects based on their heterogeneous computational capabilities. We also described the details of how each moving object and the server communicate each other to cooperatively process range monitoring queries. Through a series of simulations, we showed the effectiveness of proposed method for processing static range monitoring queries in the road network.

**Acknowledgments:** This research project was supported by Ministry of Culture, Sports and Tourism (MCST) and from Korea Copyright Commission in 2017.

**Author Contributions:** HaRim Jung initiated the idea, developed the research concept, and wrote the manuscript. Ung-Mo Kim oversaw all of the work and revised the manuscript.

**Conflicts of Interest:** The authors declare no conflict of interest.

## References

1. Ilarri, S.; Mena, E.; Illarramendi, A. Location-dependent query processing: Where we are and where we are heading. *ACM Comput. Surv.* **2010**, *42*, 1–73.
2. Cai, Y.; Hua, K.A.; Cao, G.; Xu, T. Real-time processing of range-monitoring queries in heterogeneous mobile databases. *IEEE Trans. Mob. Comput.* **2006**, *5*, 931–942.
3. Jung, H.; Kim, Y.S.; Chung, Y.D. QR-tree: An efficient and scalable method for evaluation of continuous range queries. *Inf. Sci.* **2014**, *274*, 156–176.
4. Hu, H.; Xu, J.; Lee, D.L. A generic framework for monitoring continuous spatial queries over moving objects. In Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data, Baltimore, MD, USA, 13–16 June 2005.
5. Prabhakar, S.; Xia, Y.; Aref, W.G.; Hambrusch, S. Query indexing and velocity constrained indexing: Scalable techniques for continuous queries on moving objects. *IEEE Trans. Comput.* **2002**, *51*, 1124–1140.
6. Kalashnkov, D.V.; Prabhakar, S.; Hambrusch, S.E. Main memory evaluation of monitoring queries over moving objects. *Disrtib. Parallel Database* **2004**, *15*, 117–135.

7. Wang, H.; Roger, Z. Processing of continuous location-based range queries on moving objects in road networks. *IEEE Trans. Knowl. Data Eng.* **2011**, *23*, 1065–1078.
8. Cheema, M.A.; Brankovic, L.; Lin, X.; Zhang, W.; Wang, W. Continuous monitoring of distance-based range queries. *IEEE Trans. Knowl. Data Eng.* **2011**, *23*, 1182–1199.
9. Al-Khalidi, H.; Taniar, D.; Betts, J.; Alamri, S. Monitoring moving queries inside a safe region. *Sci. World J.* **2014**, *2014*, doi:10.1155/2014/630396.
10. Yung, D.; Man, L.Y.; Lo, E. A safe-exit approach for efficient network-based moving range queries. *Data Knowl. Eng.* **2012**, *72*, 126–147.
11. Cho, H.J.; Kwon, S.J.; Chung, T.S. A safe exit algorithm for continuous nearest neighbor monitoring in road networks. *Mob. Inf. Syst.* **2013**, *9*, 37–53.
12. Cho, H.J.; Ryu, K.; Chung, T.S. An efficient algorithm for computing safe exit points of moving range queries in directed road networks. *Inf. Syst.* **2014**, *41*, 1–19.
13. Mouratidis, K.; Yiu, M.L.; Papadias, D.; Mamoulis, N. Continuous nearest neighbor monitoring in road networks. In Proceedings of the 32nd international conference on Very large data bases, Seoul, Korea, 12–15 September 2006.
14. Liu, F.; Hua, K.A. Moving query monitoring in spatial network environments. *Mob. Netw. Appl.* **2012**, *17*, 234–254.
15. Brinkhoff, T. A framework for generating network-based moving objects. *GeoInformatica* **2002**, *6*, 153–180.
16. Broch, J.; Maltz, D.A.; Johnson, D.; Hu, Y.-C.; Jetcheva, J. A performance comparison of multi-hop wireless ad hoc network routing protocols. In Proceedings of the 4th Annual ACM/IEEE International Conference on Mobile Computing and Networking, Dallas, TX, USA, 25–30 October 1998.
17. Guttman, A. R-trees: A dynamic index structure for spatial searching. In Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data, Boston, MA, USA, 18–21 June 1984.
18. Beckmann, N.; Kriegel, H.-P.; Schneider, R.; Seeger, B. The R*-tree: An efficient and robust access method for points and rectangles. In Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, USA, 23–25 May 1990.
19. Roussopoulos, N.; Faloutsos, C. The R+-tree: A dynamic index for multi-dimensional objects. In Proceedings of the 13th International Conference on Very Large Data Bases, San Francisco, CA, USA, 1–4 September 1987.
20. Papadias, D.; Zhang, J.; Mamoulis, N.; Tao, Y. Query processing in spatial network databases. In Proceedings of the 29th international conference on Very large data bases, Berlin, Germany, 9–12 September 2003.
21. Zhong, R.; Li, G.; Tan, K.L.; Zhou, L.; Gong, Z. G-tree: An efficient and scalable index for spatial search on road networks. *IEEE Trans. Knowl. Data Eng.* **2015**, *27*, 2175–2189.
22. Sun, W.; Chen, C.; Zheng, B.; Chen, C.; Liu, P. An air index for spatial query processing in road networks. *IEEE Trans. Knowl. Data Eng.* **2015**, *27*, 382–395.
23. Saltenis, S.; Jensen, C.; Leutenegger, S.; Lopez, M.A. Indexing the positions of continuously moving objects. In Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, Dallas, TX, USA, 16–18 May 2000.
24. Tao, Y.; Papadias, D.; Sun, J. The TPR*-tree: An optimized spatio-temporal access method for predictive queries. In Proceedings of the 29th International Conference on Very Large Data, Berlin, Germany, 9–12 September 2003.
25. Lee, M.L.; Hsu, W.; Jensen, C.S.; Cui, B.; Teo, K.L. Supporting frequent updates in R-trees: A bottom-up approach. In Proceedings of the 29th International Conference on Very Large Data Bases, Berlin, Germany, 9–12 September 2003.
26. Wang, H.; Zimmermann, R. A novel dual-index design to efficiently support snapshot location-based query processing in mobile environments. *IEEE Trans. Mob. Comput.* **2010**, *9*, 1280–1292.