Available online at www.sciencedirect.com
ScienceDirect
Data & Knowledge Engineering 00 (2011) 1–25
Data & Knowledge Engineering

A Safe-Exit Approach for Efficient Network-Based Moving Range Queries

Duncan Yung, Man Lung Yiu, Eric Lo*

Department of Computing, Hong Kong Polytechnic University

Abstract

Query processing on road networks has been extensively studied in recent years. However, the processing of moving queries on road networks has received little attention. This paper studies the efficient processing of moving range queries on road networks. We formulate a network-based concept called *safe exits* that guarantee the query result of the client remains unchanged before the client reaches any exit. This significantly reduces the communication overhead between moving clients and the server. We then develop an efficient algorithm for computing safe exits for a client on-demand. We evaluate the proposed techniques using real road network data. Experimental results show that our algorithm constructs safe exits efficiently and they effectively reduce the communication cost.

Keywords: Spatial/Temporal databases

1. Introduction

Online mapping services (e.g., Google Maps, Yahoo Maps) are becoming increasingly popular and easily accessible by users via mobile devices. In these maps, points-of-interests (POIs) such as tourist attractions and gas stations lie on a road network and their proximity is defined based on their shortest path distances but not Euclidean distances. Query processing on road networks has been extensively studied in recent years [8, 12, 19, 21, 22]. However, the processing of *moving queries* over POIs on *road networks* has received, surprisingly, little attention. On the other hand, while a plethora of work has been devoted to moving query processing [1, 2, 9, 18, 23, 25, 30, 31], they all focus on the Euclidean space but not on road networks.

In this paper, we study the efficient processing of moving range queries on road networks. Similar to traditional moving query processing (in the Euclidean space), the key issue is how to maintain the freshness of the query result when the user *moves*. One straightforward method is to let the user (i.e., client) issue the same query to the remote mapping services (i.e., server) periodically (e.g., every second). However, this periodic processing approach cannot solve the problem because the query results may still become out-dated in between each call to the server, not to mention the excessive computation burden imposed on the server side and the high communication frequency burden imposed on the communication channel. Another tempting approach is to predict the movement of the client so as to pre-compute the query results in advance [1, 9, 13, 25, 28]. However, in this paper, we prefer to consider a more practical situation where the moving speed and direction of the client are arbitrary.

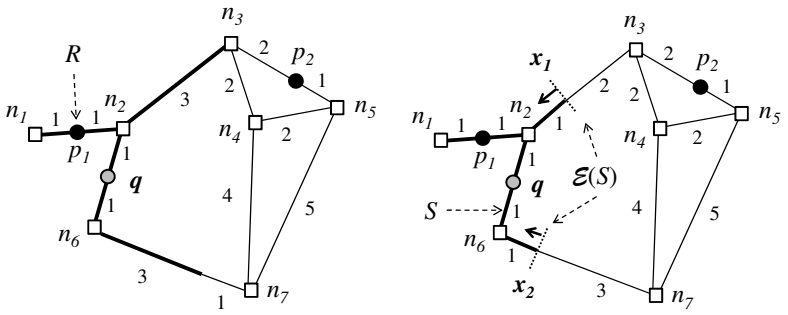
A more attractive direction is the use of *safe region* [2, 18, 30, 31]. When applying this approach to moving querying processing in the Euclidean space, the server returns to the mobile client a safe region S in addition to the

*Email addresses: {cskwyung,csmlyiu,ericlo}@comp.polyu.edu.hk

query result R. The safe region $\mathcal{S}$ of a query result set R is defined as the region where the query result R remains unchanged as long as the location of the client q is within $\mathcal{S}$. When leaving the safe region, the client needs to issue a new query to the server. The safe region approach allows the client to get the fresh query results without excessive overheads on the server side and communication channel.

In this paper, we study the "safe region" of a moving query on a road network. Consider an example that a mobile client q wants to be continuously aware of nearby POIs (e.g., gas stations) that are within $r = 4$ km. Figure 1a shows the locations of client q and a POI dataset $D = \{p_1, p_2\}$. The road segments in bold indicate the range within 4 km from the current location of the client q. Thus, the current result set R contains p_1. The safe region S (for the result set R) contains a set of road segments where the result set R remains unchanged as long as q moves within them. Figure 1b illustrates the safe region S as four bold road segments.

We then propose the notion of *safe exits* $\mathcal{E}(S)$, which capture the border of a network-based safe region $\mathcal{S}$ in a concise manner. In the example of Figure 1b, $\mathcal{E}(S)$ consists of only two exits x_1 and x_2, whereas the safe region S contains four segments. Observe that the representation of safe exits is much smaller than that of a safe region (which may comprise of many road segments). As safe exits are concise in size, they thus incur a low communication cost between the server and the client. In addition, safe exits also enable the client to check the validity of the result effectively. Before reaching any exit, the client q is guaranteed to stay in the safe region and thus its result set remains unchanged. Only when traveling beyond an exit does the client need to contact the server again to recompute the query result set R and the safe exits $\mathcal{E}(S)$.



(a) range query result $R = p_1$ (b) safe region $\mathcal{S}$, and exits $\mathcal{E}(\mathcal{S})$

Figure 1: Safe Exits on a Road Network, at $r = 4$

Our goal is to develop efficient methods to support moving queries over POIs on road networks. As in most moving query processing environments, our goal is to minimize: (i) the computation effort of evaluating the queries in the online mapping services (i.e., server time), (ii) the computation overhead of the mobile devices (i.e., client time), and (iii) the communication overhead between the server and the client. One of the biggest challenges is that the computation of safe exits requires finding not only the result set R but also the *non-result-set* $D - R$. In the example of Figure 1b, exit x_1 satisfies two conditions: (i) it is located within 4km from q and (ii) going beyond x_1 would immediately include $p_2 \in D - R$ in the result set. In other words, constructing the safe exits of a moving query result needs not only the points in the query results but also the points *not* in the query results, i.e., $D - R$. The number of points in the $D - R$ (e.g., p_2 in the example) could be very large, rendering brute-force methods which exhaustively check all points in $D - R$ impractical. Another challenge is that the distance between two points in a road network is modeled by their shortest path distance (*network distance*) but not their Euclidean distances. Existing safe region construction techniques heavily rely on the properties of the Euclidean space (e.g., [2, 18, 30, 31]) and cannot be applied here.

In this paper we present techniques that can compute safe exits efficiently. The contributions of this paper include:

1. Formulation of safe exits as a concise representation of the validity of results for moving range queries on a road network.
2. A set of effective techniques that minimize the computational overhead of safe exits.
3. A thorough experimental study that evaluates the proposed techniques on both real data and synthetic data.

The rest of the paper is summarized as follows. We review the related work in Section 2. We present the architecture and problem setting in Section 3. We then propose safe exit computation techniques in Section 4 and study optimizations in Section 5. We present the experimental study in Section 6, followed by a conclusion in Section 7.

2. Related Work

We compare our problem scenario with related work in Table 1 in terms of the query type, the space domain, the mobilities of query points and objects-of-interest, as well as their motion models. All of them adopt the client-server model where the server evaluates the queries issued by clients on-demand. In the following discussion, we let D be the set of objects-of-interest, q be a query point, and Q be the set of all queries.

Table 1: Classification of Related Work

References	Query Type	Space Domain	Query Point q	Object Dataset D	Motion Model
[8, 15, 19, 21]	range	network	static	static	N/A
[8, 12, 15, 19, 21, 22]	kNN	network	static	static	N/A
[26, 27]	range	network	static	moving	periodic update
[20]	range	network	static	moving	precinct-based update
[24]	range	network	moving	moving	point-based update
[17]	range	network	moving	moving	segment-based update
[14]	range	network	moving	moving	free
[7]	range	Euclidean	moving	moving	known travel path
[25]	kNN	Euclidean	moving	static	known travel path
[13]	kNN	network	moving	static	known travel path
[28]	range	network	moving	static	known travel path
[1, 9]	kNN	Euclidean	moving	static	vector-based update
[18, 23, 30, 31]	kNN	Euclidean	moving	static	free: safe region
[2]	range	Euclidean	moving	static	free: safe region
[3]	range	network	moving	static	free: safe region
This paper	range	network	moving	static	**free: safe exits**

2.1. Static Queries on Road Networks

Papadias et al. [19] study the processing of *static* range queries and nearest neighbor queries on road networks, where the distance between two points is defined by their shortest path distance. They propose two query processing approaches. The *network expansion* approach utilizes Dijkstra's algorithm [5] to compute the distance from the query point q to (multiple) objects, until the termination condition is satisfied. On the other hand, the *Euclidean restriction* approach exploits Euclidean lower-bound distances for restricting the search space; however, it is only applicable to specific road networks whose shortest path distances can be bounded by Euclidean distances. Since [19], specialized indexes like the graph embedding [15, 22], the network-based Voronoi diagram [12], distance index [8], and shortest path quadtrees [21] have been developed to process nearest neighbor queries on road networks efficiently. However, they still focus on static queries but not moving queries.

Recent work [20, 26] examine the monitoring of *static* range queries over moving clients on a road network. [26] requires moving clients to send their latest locations to the server periodically. It focuses on efficient maintenance of query results at the server side. [27] extends the work of [26] on monitoring the results of range queries continuously along time intervals. [20] aims at reducing the communication cost of moving clients. It imposes a partitioning of the road network into precincts and identifies the relevant range queries for each precinct. A moving client reports its latest location to the server only when it enters/leaves a precinct or the range of some query. Observe that both [20, 26] consider static range queries and take the object dataset D to be the moving clients. In contrast, our problem scenario takes a mobile client as a moving query, and the object dataset D as static POIs (e.g., restaurants, gas stations).

2.2. Moving Queries on Moving Objects

In this category of related work, both the query points and objects-of-interests are mobile clients on a road network. Each range query has a radius r and a moving center as a mobile client. [7] assumes that all mobile clients move in the Euclidean space but the query set and the object set are disjoint (i.e., $D \neq Q$). [14, 17, 24] consider the road network domain space and allow each mobile client to be both a query object and a data object (i.e., $D = Q$).

[7] aims at minimizing the communication cost between the server and mobile clients. Each query-object client needs to register its travel path at the server. On the other hand, every data-object client can have an arbitrary movement pattern. The server derives an adaptive safe region for each data object such that, while the object is within the region, it does not affect the result of any query. The computation of an adaptive safe region takes into account the movements of nearby queries. This helps reduce unnecessary location updates from mobile clients.

[24] focuses on optimizing the server processing time and adopts a point-based update policy for mobile clients. When a mobile client deviates from its latest reported location by an error threshold δ, it issues a location update to the server. Another work [17] attempts to reduce the communication cost between the server and mobile clients. It applies the segment-based update policy; a mobile client reports its latest location to the server when it enters/leaves a road segment. The solutions in [17, 24] can be applied to process moving range queries on static objects. However, they cause the server to receive frequent location updates from the moving query client (when it deviates from its last location by δ or enters/leaves a segment). In our proposal, the safe exits could be located on different segments than the client (see Figure 1b), and thus they lead to lower communication frequency.

[14] studies how to detect whether some moving clients are sufficiently close together (e.g., within the range r). In order to reduce the communication cost of mobile clients, the server assigns a proximity region to each mobile client such that the minimum distance between any two different proximity regions must be greater than the range r. While each mobile client is moving within its proximity region, it is guaranteed that its query range contains no objects (without the server having to know their exact locations). However, the semantics of proximity region are different from our safe exits. A mobile client cannot have a proximity region when it is close to some object (even though it can be far from other objects).

2.3. Moving Queries on Static Objects

Our work belongs to this category where the moving queries are issued by mobile clients and the object dataset D refers to static POIs.

[13, 25, 28] assume that the query client's future travel path is given apriori. They attempt to precompute future query results for the client in advance. Tao et al. [25] retrieves the nearest neighbors of any point on a query line segment. [13] and [28] study the problem in the road network domain; they compute the future result for kNN queries and range queries on a road network respectively, for a client with given travel path.

[1, 9] adopt a vector-based policy for mobile clients. The server assumes that each client travels with its last reported velocity and estimates their future locations. A client issues an update to the server only when its estimated location deviates from its current location by an error threshold δ (e.g., when the client's travel direction or speed changes). This triggers the server to recompute the client's future nearest neighbors.

In practice, the server may not have prior knowledge of the client's future movement pattern. In [23], the author discusses a pre-fetching approach which minimizes the communication frequency by replacing the client's k-NN queries by $(k + \Delta k)$-NN queries for the sake of collecting extra points to act as a buffer. However, the tuning of the value of Δk has notoriously been a difficult question in practice.

Recent work focuses on the safe region approach [2, 18, 30, 31], in which the server reports a safe region to the client in addition to the result set. While the client moves within the safe region, the result set remains unchanged. The representation of the safe region varies for different types of queries. For instance, Voronoi cell [30, 31] can be used as the safe region for nearest neighbor queries, V*-diagram [18] can be used as the safe region for k nearest neighbor queries, and influence objects [2] can be used for range queries. However, the aforementioned work focuses on moving queries whose clients are able to move freely in the Euclidean space. Their techniques cannot be applied to our problem scenario where the locations and movements of clients are restricted by the underlying road network.

Cheema et al. [3] is an extended paper of [2] for the road network space. It represents a safe region $\mathcal{S}$ on a road network by a collection of segments, e.g., the four bold segments in Figure 1b. In contrast, we employ a concise representation called safe exits $\mathcal{E}(\mathcal{S})$, e.g., the exits x_1 and x_2 in Figure 1b. Observe that safe exits incur much lower

communication cost and client overhead when compared to safe region (segments). The techniques in [3] do not compute safe exits. In Section 3.3, we will provide the definition of safe exits and also illustrate them in detail.

3. Architecture and Problem Setting

We first introduce the architecture in our problem setting, then elaborate preliminary concepts for road networks, and finally present the definition of safe exits on road networks.

3.1. Architecture

We adopt the typical client-server architecture in our problem scenario, as shown in Figure 2. A user carries a mobile device (e.g., iPhone, PDA) as a client, whereas the server stores the dataset of POIs D (e.g., tourist attractions, gas stations) and serves requests from clients. Initially, the client measures its current location q (Step 1a) and issues a range query to the server (Step 2). As discussed in our introduction, we employ *safe exits* for solving the moving query problem. In addition to computing the query result set R (Step 3), the server also derives a set of safe exits $\mathcal{E}(S)$ that captures the validity scope of R (Step 4). Upon receiving the result and the safe exits (Step 5), the client checks its location against safe exits. Before reaching any safe exit, the query result for the client is guaranteed to remain unchanged. Upon traveling beyond a safe exit, the user issues a query to the server again (Step 2) in order to refresh the result and its safe exits.

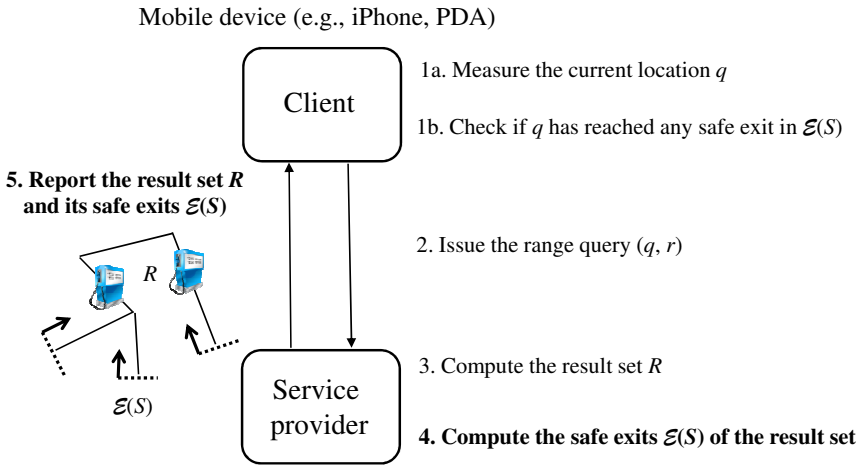


Figure 2: Architecture

We aim at developing efficient techniques to compute safe exits on road networks, with the following objectives:

- Minimize the computation time (i.e., CPU time) of the server;
- Minimize the communication cost (i.e., number of points sent) between the server and the client;
- Minimize the computation overhead of the client.

We assume that the server's main memory is large enough to accommodate the entire dataset D and the graph G, given the typical memory size (4 GB), graph size and data size (10 MB) nowadays. The server employs an in-memory adjacency list structure for storing the graph G. In the adjacency list, each edge is augmented with a list for storing the data points on that edge. We do not exploit any pre-computed distance bounds or road network indices.

For easy referencing, we summarize the notations used in the paper in Table 2.

Table 2: Summary of Notation

Symbol	Description
$G(N, E, W)$	the graph model of road network
n_i	a node on the road network
(n_i, n_j)	an edge in the edge set E
$W(n_i, n_j)$	the weight of the edge (a, b)
(n_i, n_j, λ)	a data point / a location
$(n_i, n_j, \lambda, \lambda')$	a (partial) segment
$\mathcal{U}$	the set of all possible locations on any edge of G
D	the dataset of POIs (on the graph G)
p_i	a POI of the dataset D
$dist(s, t)$	the shortest path distance from point s to point t
$dist_{min}(p, e)$	the exact min. distance between point p and edge e
$dist_{max}(p, e)$	the exact max. distance between point p and edge e
$\Upsilon(p, r)$	the r-range of a point p (on road network)
p^+ / p^-	a result POI / a non-result POI
q	a query point
$\mathcal{S}(q, r)$	the safe region of the range query (q, r)
$\mathcal{E}(S)$	the set of safe exits
x_i	an exit

3.2. Preliminary Background on Road Network

A *road network* is represented by a graph $G = (N, E, W)$ where N is the set of nodes, E is the set of edges, and W is a function that returns the weight of each edge. We assume that edges are undirected. We use the pair (n_i, n_j) to denote the edge that links nodes n_i and n_j, and represent its edge weight by $W(n_i, n_j)$. For the sake of unique representation, we require that $n_i < n_j$.

A *point* is represented by a triple (n_i, n_j, λ), where $\lambda \in [0, W(n_i, n_j)]$. This means that the point falls on the edge (n_i, n_j) and it has distance λ from n_i along the edge. A (partial) *segment* is denoted by a quadruple $(n_i, n_j, \lambda, \lambda')$, where $\lambda < \lambda'$. We use $\mathcal{U}$ to denote the set of all possible locations on any edge of G.

Given two nodes n_s and n_t, the *shortest path distance* $dist(n_s, n_t)$ is the minimum distance of any path between n_s and n_t. Given two points $p = (n_{a_1}, n_{a_2}, \lambda)$ and $p' = (n_{b_1}, n_{b_2}, \lambda')$, their *shortest path distance* is defined as follows:

$$dist(p, p') = \min\{ \; dist_E(p, p'), \min_{a \in \{a_1, a_2\}, b \in \{b_1, b_2\}} dist_E(p, a) + dist(a, b) + dist_E(b, p') \; \}, \quad (1)$$

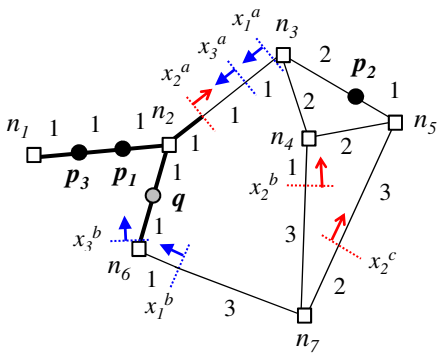
where $dist_E(p, p')$ is the *edge distance* between two points that lie on the same edge (road segment) and is defined as:

$$dist_E(p, p') = \begin{cases} |\lambda - \lambda'|: & \text{if } n_{a_1} = n_{b_1} \text{ and } n_{a_2} = n_{b_2} \\ \infty \; : & \text{otherwise} \end{cases}.$$

Figure 3 illustrates an example road network where each node n_i is a square. An edge (n_i, n_j) connects two nodes and its weight is shown as a number. For instance, edge (n_3, n_5) has its weight as 3 and point p_2 is modeled as the triple $(n_3, n_5, 2)$. The segment between n_3 and p_2 is denoted by the quadruple $(n_3, n_5, 0, 2)$; similarly, the segment between p_2 and n_5 is denoted by the quadruple $(n_3, n_5, 2, 3)$. We compute the shortest path distance between points q and p_2 as: $dist(q, p_2) = dist(q, n_2) + dist(n_2, n_3) + dist(n_3, p_2) = 1 + 3 + 2 = 6$.

Definition 1. Range Query
Given a query point q and threshold r, the range query *on a dataset D retrieves the result set $R(q, r) = \{p \in D \mid dist(q, p) \leq r\}$.*

As an example, we consider the range query with user location q and the threshold $r = 4$ in Figure 3. The result set is $R(q, r) = \{p_1, p_3\}$ because $dist(q, p_1) \leq 4$ and $dist(q, p_3) \leq 4$. Whenever the context is clear, we use R instead of $R(q, r)$.



(a) exits

Point	Location
q	$(n_2, n_6, 1)$
p_1	$(n_1, n_2, 2)$
p_2	$(n_3, n_5, 2)$
p_3	$(n_1, n_2, 1)$
x_1^a	$(n_2, n_3, 3)$
x_1^b	$(n_6, n_7, 1)$
x_2^a	$(n_2, n_3, 1)$
x_2^b	$(n_4, n_7, 1)$
x_2^c	$(n_5, n_7, 3)$
x_3^a	$(n_2, n_3, 2)$
x_3^b	$(n_2, n_6, 2)$

(b) locations of points

Edge	$\Upsilon(p_1, r)$	$\Upsilon(p_3, r)$	$\Upsilon(p_2, r)$	$\Upsilon(p_1, r) \cap \Upsilon(p_3, r)$	$\Upsilon(p_1, r) \cap \Upsilon(p_3, r) - \Upsilon(p_2, r)$
$e_1 = (n_1, n_2)$	$(e_1, 0, 3)$	$(e_1, 0, 3)$	—	$(e_1, 0, 3)$	$(e_1, 0, 3)$
$e_2 = (n_2, n_3)$	$(e_2, 0, 3)$	$(e_2, 0, 2)$	$(e_2, 1, 3)$	$(e_2, 0, 2)$	$(e_2, 0, 1)$
$e_3 = (n_2, n_6)$	$(e_3, 0, 2)$	$(e_3, 0, 2)$	—	$(e_3, 0, 2)$	$(e_3, 0, 2)$
$e_4 = (n_3, n_4)$	—	—	$(e_4, 0, 2)$	—	—
$e_5 = (n_3, n_5)$	—	—	$(e_5, 0, 3)$	—	—
$e_6 = (n_4, n_5)$	—	—	$(e_6, 0, 2)$	—	—
$e_7 = (n_4, n_7)$	—	—	$(e_7, 0, 1)$	—	—
$e_8 = (n_5, n_7)$	—	—	$(e_8, 0, 3)$	—	—
$e_9 = (n_6, n_7)$	$(e_9, 0, 1)$	—	—	—	—

(c) segments

Figure 3: Combined Example, at $r = 4$

3.3. Safe Exits on Road Network

The concept of safe region has been studied in the context of the Euclidean space [2, 18, 30], where the query point is allowed to move freely. In this section, we formulate the safe region of a range query on a road network, where the distance between points is modeled by their shortest path distance. We will then define the notion of safe exits as a concise representation of the network-based safe region.

First, we give the definition of *range*, which is a building block for the subsequent definitions.

Definition 2. Range, and Range Exits
Given a point p and a (road-network) distance threshold r, we define the range $\Upsilon(p, r) = \{z \in \mathcal{U} \mid dist(p, z) \leq r\}$, meaning the set of all locations (on roads) reachable from p within distance r. We then define the set of range exits *as $\mathcal{E}(\Upsilon(p, r)) = \{x \in \mathcal{U} \mid dist(p, x) = r\}$. In other words, the distance of any exit x from p is exactly r.*

In the context of a road network, the range is represented by a set of segments. Figure 3a shows an example road network with point p_1. Suppose that the distance threshold $r = 4$. The range $\Upsilon(p_1, 4)$ covers four segments: $(n_1, n_2, 0, 3)$, $(n_2, n_3, 0, 3)$, $(n_2, n_6, 0, 2)$, $(n_6, n_7, 0, 1)$. The border of a range can be described by range exits. For example, the range $\Upsilon(p_1, 4)$ has two exits $x_1^a = (n_2, n_3, 3)$ and $x_1^b = (n_6, n_7, 1)$.

We now define the *safe region* of the result of a range query q as all the places in the road network where the result of q remains unchanged:

Definition 3. Safe Region
The safe region $\mathcal{S}(q, r)$ of a range query result set $R(q, r)$ is defined as:

$$\mathcal{S}(q, r) = \{q' \in \mathcal{U} \mid R(q', r) = R(q, r)\}. \tag{2}$$

Since the safe region of a range query depends on the result set $R(q, r)$, which are some data points (POIs) in the dataset D, we can also express the safe region in terms of the data points. Figure 4 illustrates how a safe region can be defined based on the data points. In the figure, the dataset D contains three data points, p_1, p_2, and p_3. The result R of query (q, r) is $\{p_1, p_3\}$. First, we see that any location in the range of the $p^+ \in R$ would still treat p^+ as a result point. In other words, the intersection of the ranges of each point $p^+ \in R$ would be the location(s) that always regard $p^+ \in R$ as the results. Second, we see that any location in the range of any non-result point p^- would treat p^- as a result point. In other words, the safe region of the result set $R(q, r)$ *should exclude* the entire range of all non-result points. Third, if the dataset D is empty, all query results are the same (i.e., empty result) regardless of the location of q. In this case, the safe region would be the domain space $\mathcal{U}$.

Summarizing the above observations, the safe region of a query result $R(q, r)$ can be expressed as:

$$\mathcal{S}_D(q, r) = \begin{cases} \mathcal{U} & \text{if } D = \emptyset \\ \mathcal{S}_{D-\{p^+\}}(q, r) \cap \Upsilon(p^+, r) & \text{if } p^+ \in D, dist(q, p^+) \leq r \\ \mathcal{S}_{D-\{p^-\}}(q, r) - \Upsilon(p^-, r) & \text{if } p^- \in D, dist(q, p^-) > r \end{cases}.$$

By expanding this recursive equation for each point $p \in D$ iteratively, the safe region $\mathcal{S}(q, r)$ is equivalent to:

$$\mathcal{S}(q, r) = \mathcal{U} \cap \bigcap_{p^+ \in R(q,r)} \Upsilon(p^+, r) - \bigcup_{p^- \in D-R(q,r)} \Upsilon(p^-, r). \tag{3}$$

In the conceptual example of Figure 4, the safe region $\mathcal{S}(q, r)$ is shaded in gray, which is essentially obtained by $\Upsilon(p_1, r) \cap \Upsilon(p_3, r) - \Upsilon(p_2, r)$.

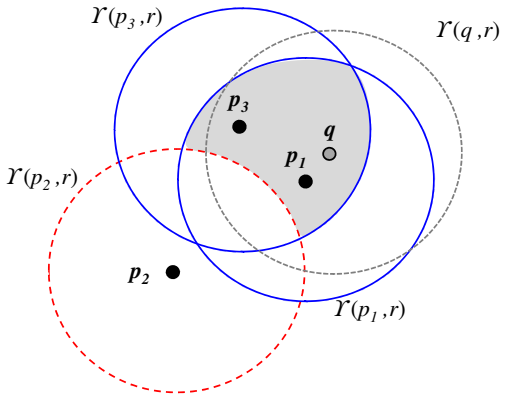


Figure 4: Conceptual Example

In a road network, the safe region is expressed by a set of segments. Figure 3a shows a road network with three points-of-interests $D = \{p_1, p_2, p_3\}$ and a range query $(q, 4)$. The result set of q is $R = \{p_1, p_3\}$ and point p_2 does not belong to the result. By Equation 3, the safe region for the result R can be expressed as $\mathcal{S}(q, r) = \Upsilon(p_1, r) \cap \Upsilon(p_3, r) - \Upsilon(p_2, r)$.

Recall that the range $\Upsilon(p_i, r)$ of each data point p_i is described by a set of segments, as illustrated in Figure 3c. In order to obtain the safe region, we have to perform an intersection and set difference operations on road segments. In the example, p_1 and p_3 are in the result set and intersecting their range $\Upsilon(p_1, 4) \cap \Upsilon(p_3, 4)$ requires the intersection of road segments such as $(e_2, 0, 3)$ and $(e_2, 0, 2)$, where the former segment belongs to $\Upsilon(p_1, r)$ and the latter segment belongs to $\Upsilon(p_3, r)$. The intersection of these two segments is $(e_2, 0, 2)$. As p_2 is not part of the result set, the safe region has to exclude the range $\Upsilon(p_2, r)$ of p_2. The range of p_2 contains the segment $(e_2, 1, 3)$. Thus, the current safe region (on the edge $e_2 = (n_2, n_3)$) has to exclude $(e_2, 1, 3)$, resulting in the segment $(e_2, 0, 1)$. Figure 3a shows the safe region for our example, where the bold segments $(n_1, n_2, 0, 3)$, $(n_2, n_3, 0, 1)$, $(n_2, n_6, 0, 2)$ denote the final safe region.

As mentioned in the introduction, we have to represent a safe region in a concise manner so as to minimize the overall communication cost. As such, we propose to physically represent a safe region by its safe exits.

Definition 4. Safe Exits
Let R be the result set of the range query, point $p^+ \in R$ be any result*, and point $p^- \in D - R$ be any* non-result*. We denote the set $X^\cup = \bigcup_{p \in D} \mathcal{E}(\Upsilon(p, r))$ as the union set of range exits for each $p \in D$. We define the set of* safe exits *as*

$$\mathcal{E}(\mathcal{S}(q, r)) \quad = \quad \{x \in X^\cup \mid \forall p^+ \in R, dist(p^+, x) \leq r \quad and \quad \forall p^- \in D - R, dist(p^-, x) \geq r\}.$$

Furthermore, an exit $x \in \mathcal{E}(\mathcal{S}(q, r))$ is said to be an inclusive exit *if $dist(p^+, x) = r$ for some $p^+ \in R$; it is said to be an* exclusive exit *if $dist(p^-, x) = r$ for some $p^- \in R$.*

Continuing with the example in Figure 3, the range query $(q, 4)$ has the result set $R = \{p_1, p_3\}$. Their range exits x_1^a, x_1^b (of $p_1 \in R$) and x_3^a, x_3^b (of $p_3 \in R$) are illustrated by solid arrows. On the other hand, the range exits x_2^a, x_2^b and x_2^c (of non-result p_2) are illustrated by line arrows. The union set of these exits $X^\cup$ is shown in Table 3. A range exit is not a safe exit if its distance from a result point is beyond r (e.g., x_1^a). Also, a range exit is not a safe exit if its distance from a non-result point is smaller than r (e.g., x_3^a). Only x_2^a and x_3^b are reported as safe exits; however, their semantics are slightly different. x_3^b is called an inclusive exit as it is inside the safe region, i.e., its range does not contain any

non-result (e.g., p_2). In contrast, x_2^a is an exclusive exit because it is barely outside the safe region $\mathcal{S}(q, r)$, i.e., its range barely contains a non-result (e.g., p_2).

These safe exits provide an effective tool for the client to check whether it is within the safe region or not. As long as the client has not reached any safe exit, it is determined to be located within the safe region. Observe that the client leaves the safe region when it passes through a safe exit.

Table 3: Checking Exits of Safe region, $r = 4$

Range exit $x \in X^{\cup}$	Is safe region exit?	Reason
x_1^a	No	$dist(p_3, x) > r$, $dist(p_2, x) < r$
x_1^b	No	$dist(p_3, x) > r$
x_2^a	Yes	—
x_2^b	No	$dist(p_1, x) > r$, $dist(p_2, x) \leq r$
x_2^c	No	$dist(p_1, x) > r$, $dist(p_2, x) \leq r$
x_3^a	No	$dist(p_2, x) < r$
x_3^b	Yes	—

4. Safe Exit Computation

In this section, we develop techniques to compute the safe exits for a range query on a road network. We first present the skeleton of the algorithm in Section 4.1. Afterwards, we discuss the implementation of each detailed operation in Section 4.2 and then present further optimizations in Section 5. For the sake of generality, we do not assume the availability of any pre-computed distance bounds (e.g., landmarks [6, 15], LLR [16, 22]) or road network indices [8, 10, 11, 21].

4.1. The Proposed Algorithmic Skeleton

Algorithm 1 depicts the skeleton of a server-side algorithm for constructing safe exits. It consists of two phases: (1) retrieving relevant points that could contribute to the safe region (see Lines 1–3), and (2) computing the safe region and the safe exits from the retrieved points (see Lines 4–16).

Phase 1: Retrieving useful points.
This phase aims at retrieving potential points that could contribute to the formation of the safe region. We intend to retrieve a small set of data points in order to reduce the computation overhead at the server side. The following lemma states that any point p^- whose distance from q is beyond $3r$ cannot influence the safe region $\mathcal{S}(q, r)$.

Lemma 1. Filtering irrelevant points
If a point p^- satisfies $dist(q, p^-) > 3r$, then we have: $\mathcal{S}(q, r) \cap \Upsilon(p^-, r) = \emptyset$.

Proof. Let p^+ be any point of the result set R. By Equation 3, we obtain: $\mathcal{S}(q, r) \subseteq \Upsilon(p^+, r)$. We attempt to prove by contradiction. For this sake, we assume that $\mathcal{S}(q, r) \cap \Upsilon(p^-, r) \neq \emptyset$. Thus, we have: $\Upsilon(p^+, r) \cap \Upsilon(p^-, r) \neq \emptyset$. Therefore, there exists a location z such that $z \in \Upsilon(p^+, r)$ and $z \in \Upsilon(p^-, r)$. By the triangle inequality of shortest path distance in a road network (Lemma 2), we derive: $dist(q, p^-) \leq dist(q, p^+) + dist(p^+, z) + dist(z, p^-) \leq r + r + r = 3r$. This contradicts with the given condition that $dist(q, p^-) > 3r$. □

Lemma 2. Triangle inequality (from Ref. [12, 19])
The shortest path distance on a road network satisfies the triangle inequality, i.e., $dist(a, c) \leq dist(a, b) + dist(b, c)$ for any locations a, b, c on the road network.

Thus, by Lemma 1, the algorithm needs to retrieve a point p when its distance from q is less than or equal to $3r$. In order to implement this, we can apply a network-based range search algorithm (e.g., Dijkstra's algorithm) to perform a range search from q with a distance threshold $3r$ on the dataset D (see Line 1). The set of retrieved points D' are

Algorithm 1 Computing Range Query and Safe Region (Skeleton)

Algorithm Computing-Range-Query-and-Safe-Region (Dataset D, Query point q, Distance Threshold r)

*/*Phase 1: Retrieving relevant points*/*

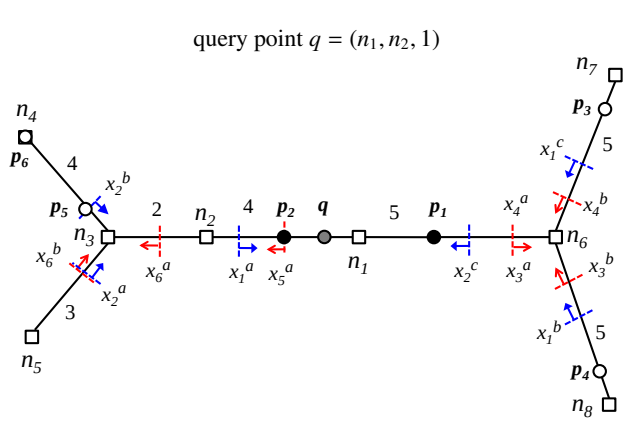
1: Point set D':= `RangeSearch` $(D, q, 3r)$; ▹ by Dijkstra's Algorithm
2: Point set R:={ $p^+ \in D' \mid dist(q, p^+) \leq r$ }; ▹ results
3: Point set NR:={ $p^- \in D' \mid dist(q, p^-) > r$ }; ▹ non-results

*/*Phase 2: Computing the safe region*/*

4: Segment set $\mathcal{S}_{cur}$:=$\mathcal{U}$; ▹ current safe region
5: **while** R or NR is non-empty **do**
6: Point p:= `PickPoint`(R, NR); ▹ Algorithm 4 (Section 5.3)
7: **if** `PrunePoint`(p, R, NR) **then** ▹ pruning (Section 5.2)
8: continue with the next iteration (at Line 5);
9: **else** ▹ refine safe region (Section 4.2)
10: $\Upsilon(p, r)$:= `ComputeSegments`(p, r); ▹ Algorithm 2
11: **if** $p \in R$ **then** ▹ compute the intersection
12: $\mathcal{S}_{cur}$:=$\mathcal{S}_{cur} \cap \Upsilon(p, r)$;
13: **else** ▹ compute the set difference
14: $\mathcal{S}_{cur}$:=$\mathcal{S}_{cur} - \Upsilon(p, r)$;
15: compute the set $\mathcal{E}(\mathcal{S}_{cur})$ of safe exits;
16: **return** $\mathcal{E}(\mathcal{S}_{cur})$ and the result set to the client;

then classified into the actual result set R and the non-result set NR, depending on whether their distances from q are below r or not (see Lines 2-3).

We now illustrate the execution of Phase 1 on the example of Figure 5. Suppose that the user at point q issues a range query with radius $r = 5$. Then, the server executes a range search on the dataset D at point q with the threshold $3r = 15$. The result set R contains points p_1 and p_2, whereas the non-result set NR contains points p_3, p_4, p_5, p_6. (Other non-result points that are farther than $3r$ are not shown).



(a) road network

Point	Segments	Exits
$p_1 = (n_1, n_6, 2)$	$(n_1, n_6, 0, 5)$ $(n_1, n_2, 0, 3)$ $(n_6, n_7, 0, 2)$ $(n_6, n_8, 0, 2)$	$(n_6, n_7, 2)$ $(n_6, n_8, 2)$ $(n_1, n_2, 3)$
$p_2 = (n_1, n_2, 2)$	$(n_1, n_2, 0, 4)$ $(n_1, n_6, 0, 3)$ $(n_3, n_4, 0, 1)$ $(n_3, n_5, 0, 1)$ $(n_2, n_3, 0, 2)$	$(n_1, n_6, 3)$ $(n_3, n_4, 1)$ $(n_3, n_5, 1)$
$p_3 = (n_6, n_7, 4)$	$(n_6, n_7, 0, 5)$ $(n_6, n_8, 0, 1)$ $(n_1, n_6, 4, 5)$	$(n_1, n_6, 1)$ $(n_6, n_8, 1)$
$p_4 = (n_6, n_8, 4)$	$(n_6, n_7, 0, 1)$ $(n_6, n_8, 0, 5)$ $(n_1, n_6, 4, 5)$	$(n_1, n_6, 1)$ $(n_6, n_7, 1)$
$p_5 = (n_3, n_4, 1)$	$(n_3, n_4, 0, 4)$ $(n_3, n_5, 0, 3)$ $(n_1, n_2, 2, 4)$ $(n_2, n_3, 0, 2)$	$(n_1, n_2, 2)$
$p_6 = (n_3, n_4, 4)$	$(n_3, n_4, 0, 4)$ $(n_3, n_5, 0, 1)$ $(n_2, n_3, 1, 2)$	$(n_2, n_3, 1)$

(b) range segments and exits of data points

Figure 5: Running Example

Phase 2: Computing the safe region.

After we retrieve a sufficient set of points that could contribute to the safe region, the next step is to compute the safe region by utilizing the points in the sets R and NR. We denote the current safe region by $\mathcal{S}_{cur}$, which is represented by a set of segments, and initialized to the graph domain space $\mathcal{U}$. The loop (Lines 5–14) iteratively use each retrieved point to refine $\mathcal{S}_{cur}$.

At Line 6, we employ an ordering for picking the next point p to be processed. By Equation 3, the ordering of processing the points does not influence the correctness of the safe region. However, by carefully choosing the order of processing the retrieved points, we are able to refine the current safe region $\mathcal{S}_{cur}$ to the final safe region at a faster rate. We will discuss this issue in detail in Section 5.3.

After the safe region $\mathcal{S}_{cur}$ has been refined by some points, it eventually shrinks to a small set of segments around q. In this case, the current point p may become no longer useful in refining the safe region further. Intuitively, if p is a result point ($p^+ \in R$) that is located very close to q, then its range $\Upsilon(p^+, r)$ could contain the entire $\mathcal{S}_{cur}$. Such point p^+ cannot refine $\mathcal{S}_{cur}$ further. Similarly, if p is a non-result point ($p^- \in R$) that is located very far from q, then its range $\Upsilon(p^-, r)$ does not intersect $\mathcal{S}_{cur}$. Such point p^- also cannot refine $\mathcal{S}_{cur}$ further. Therefore, we devise a set of pruning rules to check whether a point p can further refine the safe region. The test is done at Line 7. If p can be pruned early, then we can avoid subsequent processing at Lines 10–14, which involves network expansion operations. We will discuss the derivation of the aforementioned pruning rules in detail in Section 5.2.

When a point p is possible to further refine the current safe region $\mathcal{S}_{cur}$ (Lines 10–14), we first compute the set of segments $\Upsilon(p, r)$ that are within distance r from point p. If p is a result point, we shall update $\mathcal{S}_{cur}$ by intersecting it with $\Upsilon(p, r)$; otherwise, we shall exclude $\Upsilon(p, r)$ from $\mathcal{S}_{cur}$. We will study the efficient implementation of these steps in Section 4.2. The computation of $\Upsilon(p, r)$ (at Line 10) actually requires network expansion operations. We thus discuss in Section 5.1 how to optimize this operation.

After processing point p, the algorithm continues with another point p as long as R or NR is non-empty. In the end (Line 15), we compute the set of safe exits $\mathcal{E}(\mathcal{S}_{cur})$ from $\mathcal{S}_{cur}$, and then report the result R and the set of safe exits $\mathcal{E}(\mathcal{S}_{cur})$ to the client.

4.2. Computations of Range Segment Set, Segment Intersection and Difference

We proceed to discuss the detailed implementation of the above operations. First, we present an algorithm for computing $\Upsilon(p, r)$ (for Line 10), which contains the set of segments that are within the range from point p. Then, we investigate how to compute the intersection and the set difference between two sets of segments (for Lines 12 and 14).

4.2.1. Computing the Range Segment Set $\Upsilon(p, r)$

In the road network context, the range $\Upsilon(p, r)$ of a point p is expressed as a set of segments whose distances from p are within r.

Let $dist_{min}(p, e)$ and $dist_{max}(p, e)$ represent the minimum distance and the maximum distance from point p to edge e, respectively. Their derivation will be elaborated later on. With the values of $dist_{min}(p, e)$ and $dist_{max}(p, e)$, we decide (i) whether the range set $\Upsilon(p, r)$ contains a segment of edge e, and (ii) whether the range exit set $\mathcal{E}(\Upsilon(p, r))$ contains an exit on e. These conditions are summarized in Table 4.

Table 4: Conditions for Range Segments and Exits

Condition	$\Upsilon(p, r)$ **segment**	$\mathcal{E}(\Upsilon(p, r))$ **segment**
$dist_{min}(p, e) \leq dist_{max}(p, e) < r$	entire e	no
$dist_{min}(p, e) \leq r \leq dist_{max}(p, e)$	partial e	yes
$r < dist_{min}(p, e) \leq dist_{max}(p, e)$	no	no

We apply Dijkstra's algorithm [5] in order to collect the distances of edges from p, in a single traversal of the graph. We will also present the necessary adaptations for deriving the set of range segments $\Upsilon(p, r)$ and range exits $\mathcal{E}(\Upsilon(p, r))$. The traversal will be terminated once we detect that no more range segment nor range exit can be found in future.

Algorithm 2 is the pseudo-code of this method, which takes a point p and a threshold r as input. It utilizes a set Υ for storing range segments and a set $\mathcal{E}$ for keeping range exits. A min-heap H is used to facilitate the traversal of graph nodes in the ascending order of their distances from p. Initially, we enheap the nodes that are on the edge of p.

In each iteration of the loop, the node (say, n_i) with the minimum distance (from p) is deheaped. If it has not been visited before, we set n_i as visited and enheap each of its unvisited adjacent node n_j into H. Here is a slight difference from Dijkstra's algorithm. We limit the search by using the condition $d' \leq r$ (at Line 11). The reason is that, when the current node n_i satisfies $dist(p, n_i) > r$ and an adjacent node n_j has not been visited, we guarantee that

$dist_{min}(p, n_j) > r$. By the corresponding condition in Table 4, the edge does not contain any range segment nor range exit. Thus, we avoid expanding such adjacent node n_j.

Lines 13–23 are devoted to compute range segment and exit on the edge e. When both nodes of e have been visited (see Line 13), we have the distances $dist(p, n_i)$ and $dist(p, n_j)$, and then we compute $dist_{min}(p, e)$ and $dist_{max}(p, e)$. Based on these values, we compute range segment and exit on the edge e according to the rules in Table 4, and insert them into the sets Υ and $\mathcal{E}$, respectively.

When H becomes empty, the loop (of Lines 6–23) terminates, and the algorithm reports the range segment set Υ and its corresponding exits $\mathcal{E}$.

Algorithm 2 `ComputeSegments` Algorithm

```
    Algorithm ComputeSegments( POI p, Threshold r )
 1: Υ:=∅;                                          ▷ a set of range segments
 2: E:=∅;                                          ▷ a set of range exits
 3: H:=create a min-heap;                          ▷ for traversing the network
 4: let the edge (n_a, n_b) be the edge that contains p;
 5: insert ⟨n_a, dist_E(p, n_a)⟩ and ⟨n_b, dist_E(p, n_b)⟩ into H;
 6: while H is not empty do
 7:     ⟨n_i, d'⟩:=deheap the top entry from H;
 8:     if n_i is unvisited then
 9:         mark n_i as visited, and set dist(p, n_i) to d';
10:         for each adjacent node n_j of n_i do
11:             if n_j is unvisited and d' ≤ r then
12:                 insert ⟨n_j, d' + W(n_i, n_j)⟩ into H;
13:             else if n_j is visited then
14:                 n'_i:=min{n_i, n_j}; n'_j:=max{n_i, n_j};
15:                 let e be the edge (n'_i, n'_j);
16:                 compute dist_min(p, e) and dist_max(p, e);
17:                 if dist_max(p, e) ≤ r then
18:                     insert (e, 0, W(e)) into Υ;
19:                 else if dist_min(p, e) ≤ r then
20:                     compute the range exit of p on edge e;
21:                     insert the range exit into E;
22:                     compute the segment (e, λ, λ') such that its distance from p is within r;
23:                     insert the segment into Υ;
24: return Υ and E;
```

It remains to clarify a few subtle issues: (i) how do we derive $dist_{min}(p, e)$ and $dist_{max}(p, e)$, and (ii) how do we compute the exits on an edge.

Derivation of $dist_{min}(p, e)$ and $dist_{max}(p, e)$.
Let $e = (n_i, n_j)$ be an edge on the road network. Formally, we define $dist_{min}(p, (n_i, n_j))$ and $dist_{max}(p, (n_i, n_j))$ as follows:

$$dist_{min}(p, (n_i, n_j)) = \min_{\lambda \in [0, W(n_i, n_j)]} dist(p, (n_i, n_j, \lambda)),$$
$$dist_{max}(p, (n_i, n_j)) = \max_{\lambda \in [0, W(n_i, n_j)]} dist(p, (n_i, n_j, \lambda)),$$

where (n_i, n_j, λ) (for any $\lambda \in [0, W(n_i, n_j)]$) denotes any possible location on the edge (n_i, n_j).

We will convert these fundamental equations into closed-form equations that involve a finite number of terms, such as the edge weight $W(n_i, n_j)$ and the distances $dist(p, n_i)$ and $dist(p, n_j)$.

First, we express the distance $dist(p, z)$ in Lemma 3 in terms of $W(n_i, n_j)$, $dist(p, n_i)$, and $dist(p, n_j)$.

Lemma 3. Distances from point p
Let p be a point and (n_i, n_j) be an edge such that p does not fall on (n_i, n_j). For any location $z = (n_i, n_j, \lambda)$ on the edge, we have:

$$dist(p, z) = \min\{dist(p, n_i) + \lambda, dist(p, n_j) + W(n_i, n_j) - \lambda\}.$$

Proof. Since p is not on the edge, we have: $dist_E(p, z) = \infty$. We then simplify Equation 1 to obtain: $dist(p, z) = \min\{dist(p, n_i) + \lambda, dist(p, n_j) + W(n_i, n_j) - \lambda\}$. □

Based on the above expression, we present two lemmas for computing $dist_{min}(p, (n_i, n_j))$ and $dist_{max}(p, (n_i, n_j))$ in closed-form.

Lemma 4. Minimum and maximum distances from point p
The minimum distance from p to any location on the edge, denoted by $dist_{min}(p, (n_i, n_j))$, is computed as:

$$dist_{min}(p, (n_i, n_j)) = \begin{cases} 0 & \text{if } p \text{ is on } (n_i, n_j) \\ \min\{dist(p, n_i), dist(p, n_j)\} & \text{otherwise} \end{cases}. \quad (4)$$

The maximum distance from p to any location on the edge, denoted by $dist_{max}(p, (n_i, n_j))$, is computed as:

$$dist_{max}(p, (n_i, n_j)) = \begin{cases} \max\{dist(p, n_i), dist(p, n_j)\} & \text{if } p \text{ is on } (n_i, n_j) \\ \frac{dist(p,n_i)+dist(p,n_j)+W(n_i,n_j)}{2} & \text{otherwise} \end{cases}. \quad (5)$$

Proof. When p is on the edge, $dist_{min}$ is definitely zero. Otherwise, we apply Lemma 3 and attempt to minimize the value of $dist(p, z)$, where z is any location on the edge. In this case, we obtain: $dist_{min}(p, (n_i, n_j)) = \min\{dist(p, n_i), dist(p, n_j)\}$.
When p is on the edge, the furthest location from p is located at either node (n_i or n_j) of the edge. In case p is not on the edge, we apply Lemma 3 and attempt to maximize the value of $dist(p, z)$, where z is any location on the edge. Observe that its value is maximized when the condition below is satisfied: $dist(p, n_i) + \lambda = dist(p, n_j) + W(n_i, n_j) - \lambda$. We then derive: $\lambda = \frac{W(n_i,n_j)+dist(p,n_j)-dist(p,n_i)}{2}$. By substituting λ into the equation of $dist(p, z)$, we obtain: $dist(p, z) = \frac{dist(p,n_i)+dist(p,n_j)+W(n_i,n_j)}{2}$. □

We now illustrate how to compute the minimum and maximum distances. Consider point p_1 and edge (n_4, n_7) in Figure 3. Suppose that we have computed the distances: $dist(p_1, n_4) = 1 + 3 + 2 = 6$ and $dist(p_1, n_7) = 1 + 2 + 4 = 7$. Observe that p_1 is not on edge (n_4, n_7). We then compute: $dist_{min}(p_1, (n_4, n_7)) = \min\{6, 7\} = 6$ and $dist_{max}(p_1, (n_4, n_7)) = \frac{6+7+4}{2} = 8.5$.

Finding Range Exits on Edge e.
When an edge e satisfies $dist_{min}(p, e) \le r \le dist_{max}(p, e)$, there exists some range exit on e, according to Table 4. We then apply the following procedure to compute the location(s) of exit(s) on the edge $e = (n_i, n_j)$.

- *Case (1): p is on the edge.*
  We obtain the exit(s) by moving the location of p towards n_i or n_j by distance r.
- *Case (2): p is not on the edge.*
  In the proof of the maximum distance expression in Lemma 4, the location $z^* = (n_i, n_j, \lambda)$ on the edge contributes to the maximum distance $dist_{max}(p, (n_i, n_j))$ from p, when $\lambda = \frac{W(n_i,n_j)+dist(p,n_j)-dist(p,n_i)}{2}$. Thus, we obtain the exit(s) by moving the location of z^* towards n_i or n_j by the distance $dist_{max}(p, (n_i, n_j)) - r$.

As an example, we consider the point p_1 with threshold $r = 4$ in Figure 3. We take a few edges on the graph as examples.

- For edge (n_4, n_7), we have: $dist_{min}(p_1, (n_4, n_7)) = 6$. Since its $dist_{min}$ value is larger than r, there is no exit on (n_4, n_7).
- For edge (n_1, n_2), we have: $dist_{max}(p_1, (n_1, n_2)) = 2$. Since it $dist_{max}$ value is smaller than r, there is also no exit on (n_1, n_2).

- For edge (n_6, n_7), we have: $dist_{min}(p_1, (n_6, n_7)) = 3$ and $dist_{max}(p_1, (n_6, n_7)) = 7$. Thus, we proceed to determine the exit(s) on the edge. First, we compute the the value $\lambda = \frac{W(n_6,n_7)+dist(p_1,n_7)-dist(p_1,n_6)}{2} = \frac{4+7-3}{2} = 4$, such that the location $z^* = (n_6, n_7, 4)$ contributes to the maximum distance $dist_{max}(p_1, (n_6, n_7)) = 7$. Next, we move the location z^* towards n_6 by the distance $7 - r = 3$, in order to obtain the exit $(n_6, n_7, 1)$.

4.2.2. Computing the Intersection of Segment Sets

In Algorithm 1 (Lines 11–12), we refine $\mathcal{S}_{cur}$ to the intersection of $\mathcal{S}_{cur}$ and $\Upsilon(p^+, r)$, when point p^+ is a result (i.e., $p^+ \in R$). We study how to implement this intersection operation efficiently.

Let α be a set of segments. Given an edge e, we define $\alpha[e]$ as the segment of α that falls on e, say, the segment $(e, \lambda_\alpha, \lambda'_\alpha)$. In case α contains no segment on e, then $\alpha[e]$ is empty.

$$\alpha[e] \quad = \quad \begin{cases} (e, \lambda_\alpha, \lambda'_\alpha) \text{ if } \alpha \cap e \neq \emptyset \\ \emptyset \qquad\qquad\quad \text{if } \alpha \cap e = \emptyset \end{cases}. \tag{6}$$

Given two segments $\alpha[e]$ and $\beta[e]$, we compute their intersection as follows:

$$\alpha[e] \cap \beta[e] \quad = \quad \begin{cases} \emptyset & \text{if } \alpha[e] = \emptyset \text{ or } \beta[e] = \emptyset \\ \emptyset & \text{if } \lambda'_\alpha < \lambda_\beta \text{ or } \lambda'_\beta < \lambda_\alpha \\ (e, \lambda_\beta, \lambda'_\alpha) & \text{if } \lambda_\alpha < \lambda_\beta \leq \lambda'_\alpha \leq \lambda'_\beta \\ (e, \lambda_\beta, \lambda'_\beta) & \text{if } \lambda_\alpha < \lambda_\beta \leq \lambda'_\beta < \lambda'_\alpha \\ (e, \lambda_\alpha, \lambda'_\beta) & \text{if } \lambda_\beta \leq \lambda_\alpha < \lambda'_\beta < \lambda'_\alpha \\ (e, \lambda_\alpha, \lambda'_\alpha) & \text{if } \lambda_\beta \leq \lambda_\alpha \leq \lambda'_\alpha \leq \lambda'_\beta \end{cases}.$$

When $\alpha[e] = \emptyset$ or $\beta[e] = \emptyset$, their intersection is definitely empty. This corresponds to the first case. The other cases are derived based on the relative positions of $\alpha[e]$ and $\beta[e]$.

Regarding the implementation issue, the intersection between two sets of segments α and β can be computed efficiently by using a hash table HT. First, we insert each segment of α (with edge e as key) into HT. For each segment of β, we probe HT (with edge e as key) to check if it contains a segment on e. If so, then we compute the intersection $\alpha[e] \cap \beta[e]$ between these two segments. The above procedure takes exactly $O(|\alpha| + |\beta|)$ time.

Continuing with the example of Figure 5. We want to compute the safe region for the query point q with the threshold $r = 5$. Recall that the result set R contains points p_1 and p_2. Suppose that we first process p_1 and then p_2. Initially, we compute $\Upsilon(p_1, r)$ (by Algorithm 2) and then obtain the range segment set $\Upsilon(p_1, r)$ (with four segments) and the range exit set $\mathcal{E}(\Upsilon(p_1, r)) = \{x_1^a, x_1^b, x_1^c\}$. The current safe region $\mathcal{S}_{cur}$ is set to $\Upsilon(p_1, r)$. Next, we compute $\Upsilon(p_2, r)$ (by Algorithm 2) and then obtain the range segment set $\Upsilon(p_2, r)$ (with five segments) and the range exit set $\mathcal{E}(\Upsilon(p_2, r)) = \{x_2^a, x_2^b, x_2^c\}$. Then, we refine $\mathcal{S}_{cur}$ by intersecting itself with $\Upsilon(p_2, r)$. Table 5 depicts the steps for computing the intersection between $\mathcal{S}_{cur}$ and $\Upsilon(p_2, r)$. Their intersection contains the segments $(n_1, n_2, 0, 3), (n_1, n_6, 0, 3)$. Note that $(n_1, n_6, 3)$ indicates a partial edge. The current set of safe exits $\mathcal{E}$ contains x_1^a and x_2^c because they fall into the region $\mathcal{S}_{cur}$.

Table 5: Intersection of range Segments, $r = 5$

$\mathcal{S}_{cur}$	$\Upsilon(p_2, r)$	**Intersection**
$(n_1, n_2, 0, 3)$	$(n_1, n_2, 0, 4)$	$(n_1, n_2, 0, 3)$
—	$(n_2, n_3, 0, 2)$	—
—	$(n_3, n_4, 0, 1)$	—
—	$(n_3, n_5, 0, 1)$	—
$(n_1, n_6, 0, 5)$	$(n_1, n_6, 0, 3)$	$(n_1, n_6, 0, 3)$
$(n_6, n_7, 0, 2)$	—	—
$(n_6, n_8, 0, 2)$	—	—

4.2.3. Computing the Set Difference of Segment Sets

In Algorithm 1 (Lines 13–14), we refine $\mathcal{S}_{cur}$ to the set difference between $\mathcal{S}_{cur}$ and $\Upsilon(p^-, r)$, when point p^- is a non-result (i.e., $p^- \in NR$). We study how to implement this set difference operation efficiently.

Recall that the notation $\alpha[e]$ represents the segment of α that falls on edge e (if any). We then compute the set difference between two segments $\alpha[e]$ and $\beta[e]$ as follows:

$$\alpha[e] - \beta[e] \quad = \quad \begin{cases} \emptyset & \text{if } \alpha[e] = \emptyset \\ (e, \lambda_\alpha, \lambda'_\alpha) & \text{if } \beta[e] = \emptyset \text{ and } \alpha[e] \neq \emptyset \\ (e, \lambda_\alpha, \lambda'_\alpha) & \text{if } \lambda'_\alpha < \lambda_\beta \text{ or } \lambda'_\beta < \lambda_\alpha \\ (e, \lambda_\alpha, \lambda_\beta) & \text{if } \lambda_\alpha < \lambda_\beta \leq \lambda'_\alpha \leq \lambda'_\beta \\ (e, \lambda_\alpha, \lambda_\beta), (e, \lambda'_\beta, \lambda'_\alpha) & \text{if } \lambda_\alpha < \lambda_\beta \leq \lambda'_\beta < \lambda'_\alpha \\ (e, \lambda'_\beta, \lambda'_\alpha) & \text{if } \lambda_\beta \leq \lambda_\alpha < \lambda'_\beta < \lambda'_\alpha \\ \emptyset & \text{if } \lambda_\beta \leq \lambda_\alpha \leq \lambda'_\alpha \leq \lambda'_\beta \end{cases} .$$

Regarding the implementation issue, the set difference between two sets of segments α and β can be computed efficiently by using a hash table HT. This task takes exactly $O(|\alpha| + |\beta|)$ time.

Continuing with the example of Figure 5, the current safe region $\mathcal{S}_{cur}$ now contains two segments $(n_1, n_2, 0, 3), (n_1, n_6, 0, 3)$. Let p_5 be a non-result point chosen to be processed next. We invoke Algorithm 2 to obtain the range segment set $\Upsilon(p_5, r)$ (with four segments) and the range exit set $\mathcal{E}(\Upsilon(p_5, r)) = \{x_5^a\}$. Table 6 depicts the steps for computing the set difference between $\mathcal{S}_{cur}$ and $\Upsilon(p_5, r)$. The set difference contains the segments $(n_1, n_2, 0, 2), (n_1, n_6, 0, 3)$. Then, we update $\mathcal{S}_{cur}$ to the above set difference. The current set of safe exits $\mathcal{E}$ contains x_2^c and x_5^a, where x_2^c is an inclusive exit (from a result) and x_5^a is an exclusive exit (from a non-result).

Table 6: Set Difference of range Segments, $r = 5$

$\mathcal{S}_{cur}$	$\Upsilon(p_5, r)$	**Set Difference**
$(n_1, n_2, 0, 3)$	$(n_1, n_2, 2, 4)$	$(n_1, n_2, 0, 2)$
—	$(n_2, n_3, 0, 2)$	—
—	$(n_3, n_4, 0, 4)$	—
—	$(n_3, n_5, 0, 3)$	—
$(n_1, n_6, 0, 3)$	—	$(n_1, n_6, 0, 3)$

5. Optimizations on Safe Exit Computation

This section presents several optimizations on speeding up the computation of safe exits, in particular the operations `ComputeSegment`, `PrunePoint`, and `PickPoint` (in Algorithm 1). Section 5.1 studies an optimized implementation of computing relevant road segments `ComputeSegment`. Section 5.2 presents two pruning rules for implementing `PrunePoint`. Section 5.3 discusses some heuristics for choosing a point in `PickPoint`.

5.1. Avoiding Expansion of Irrelevant Nodes during Computation of $\Upsilon(p, r)$

As discussed in Section 4.2, we execute Algorithm 2 (`ComputeSegment`) to obtain the range segment set $\Upsilon(p, r)$. In practice, not every segment in $\Upsilon(p, r)$ can refine $\mathcal{S}_{cur}$. Thus, there is no need to obtain the entire $\Upsilon(p, r)$. In order to reduce the running time of Algorithm 2, we develop a condition that can bound the expansion without jeopardizing the correctness.

Figure 6 illustrates an example scenario. The data point p serves as the source of network expansion in Algorithm 2. Let n be the current visited node and $dist(p, n)$ be its distance from p. Let z be any location on any segment in $\mathcal{S}_{cur}$. Observe that the shortest path from p via n to z equals to $dist(p, n) + dist(n, z)$. The idea is that, if we can show that $dist(p, n) + dist(n, z) > r$ for all location $z \in \mathcal{S}_{cur}$, then we can stop the expansion at node n as such a path cannot reach the safe region $\mathcal{S}_{cur}$ within distance r. In other words, such a path cannot lead to the shrinking of $\mathcal{S}_{cur}$.

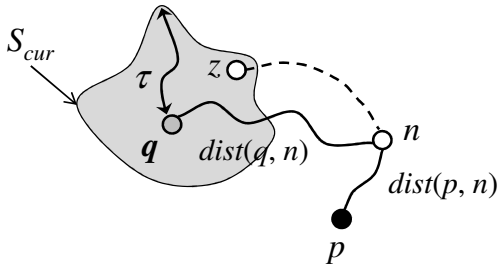


Figure 6: Restricting Traversal

Although powerful, this idea requires knowledge of the distance $dist(n, z)$ for each possible $z \in \mathcal{S}_{cur}$, which is not yet available in Algorithm 2. We cannot afford to invoke expensive network expansion to compute $dist(n, z)$, not to mention the large number of possible locations in $\mathcal{S}_{cur}$.

To solve the problem, we discover that the range search operation (Line 1) of Algorithm 1 provides useful distance information that can be exploited. During the range search, we simply collect the distance $dist(q, n)$ of each visited node n from q. Since the search range is $3r$, the collected set of node-distance pairs is:

$$M_q = \{\langle n, dist(q, n)\rangle \mid n \in N, dist(q, n) \leq 3r\} \tag{7}$$

We then store the pairs of M_q as a hash table in order to support constant-time lookup of the distance $dist(q, n)$ based on a key node n.

We exploit the above 'free' distance information and establish Lemma 5 to determine efficiently whether the expansion at node n can be stopped. When the condition $dist(p, n) + dist(q, n) - \tau > r$ is satisfied, it is guaranteed that $dist(p, n) + dist(n, z) > r$ holds for all $z \in \mathcal{S}_{cur}$.

Lemma 5. Restricting traversal by using τ
Let p be a data point and n be the node being examined. If $dist(p, n) + dist(q, n) - \tau > r$, then the shortest path from p via n to any location z in $\mathcal{S}_{cur}$ must have a length greater than r, i.e., $dist(p, n) + dist(n, z) > r$.

Proof. Let z be any location on any segment in $\mathcal{S}_{cur}$. By triangular inequality (Lemma 2), we have: $dist(q, n) \leq dist(q, z) + dist(z, n)$. Since $\tau \geq dist(z, q)$ (by Definition 5), we then derive: $dist(q, n) \leq \tau + dist(z, n)$. We then rearrange the terms to get: $dist(n, z) \geq dist(q, n) - \tau$. Adding the term $dist(p, n)$ to both sides, we obtain: $dist(p, n) + dist(n, z) \geq dist(p, n) + dist(q, n) - \tau$. By using the given condition $dist(p, n) + dist(q, n) - \tau > r$, we derive: $dist(p, n) + dist(n, z) > r$. □

We then propose to modify Algorithm 2 as follows. Just after Line 9, we check whether the current node n_i satisfies $dist(p, n_i) + dist(q, n_i) - \tau > r$. If so, then we skip further processing on n_i. This saves the computation cost from expanding irrelevant nodes in the algorithm. Observe that the distance $dist(p, n_i)$ is available from the deheaped entry and the distance $dist(q, n_i)$ can be obtained from the hash table M_q for free. This checking takes only $O(1)$ time.

5.2. Pruning Useless Retrieved Data Points

In Algorithm 1 (Line 7), we apply pruning rules to check whether point p contributes to the current safe region $\mathcal{S}_{cur}$. In case p can be pruned, we save the cost of computing the range segment set $\Upsilon(p, r)$ (at Line 10).

These pruning rules are particularly useful when the range r is large or the number of data points is large. To understand this, consider the scenario where the value of r is scaled up by f times (for some constant f). We expect that the sizes of R and NR increase at least by f times. For each point p (in R or NR), we compute its range set $\Upsilon(p, r)$. As r is scaled up by f times, the cost of each range computation also increases at least by f times. Thus, the total computation cost increases by f^2 and it does not scale well with respect to f.

Thus, it is important for us to develop rules for pruning useless retrieved data points. Intuitively, these rules should be designed in such a way that:

- They are effective. A useless retrieved data point can be pruned away with high probability.

- They are efficient. Ideally, the rules should not invoke any (expensive) traversal operations on the road network.

First, we introduce the following notion to capture the proximity between the query point q and the current safe region $\mathcal{S}_{cur}$.

Definition 5. Furthest Safe Region Distance
The furthest distance of the current safe region $\mathcal{S}_{cur}$ to the query point q is defined as follows. Whenever the context is clear, we use τ instead of $\tau(q, \mathcal{S}_{cur})$.

$$\tau(q, \mathcal{S}_{cur}) = \max_{(e,\lambda,\lambda') \in \mathcal{S}_{cur}} dist_{max}(q, (e, \lambda, \lambda')) \tag{8}$$

Recall from Section 5.1 that the distances of some nodes from q can be obtained from the hash table M_q for free. We discover that the value τ can be computed efficiently by using the hash table M_q. According to Definition 5, we examine each segment (e, λ, λ') of $\mathcal{S}_{cur}$ and compute its maximum distance $dist_{max}(q, (e, \lambda, \lambda'))$ to the query point q. By Equation 5, the value $dist_{max}(q, (e, \lambda, \lambda'))$ can be derived from the distances from q to both nodes of e (distances obtained from the hash table M_q). Observe that there is no need to invoke expensive graph traversal operations for computing τ.

We propose the following algorithm as the implementation of `PrunePoint` (used in Algorithm 1). By Lemma 3, the distance $dist(q, p)$ can be computed from the distances from q to both adjacent nodes of p (by using the hash table M_q). We use p^+ to represent the data point p when it is a result (i.e., $p^+ \in R$); otherwise, we use p^- (i.e., $p^- \in NR$). We proceed to elaborate pruning rules for p^+ and p^-, respectively.

Algorithm 3 `PrunePoint`Algorithm

```
Algorithm PrunePoint( Point p, Point set R, Point set NR )
1:  obtain the value of τ;
2:  if p ∈ R then
3:      let p+ be p, and remove it from R;
4:      if dist(q, p+) + τ ≤ r then                          ▷ apply Lemma 6
5:          return true;
6:      else
7:          return false;
8:  if p ∈ NR then
9:      let p− be p, and remove it from NR;
10:     if dist(q, p−) − τ > r then                          ▷ apply Lemma 7
11:         return true;
12:     else
13:         return false;
```

For the case of a result p^+, we apply Lemma 6 to check whether p^+ is sufficiently close to $\mathcal{S}_{cur}$. If so, then the range $\Upsilon(p^+, r)$ cannot help shrink $\mathcal{S}_{cur}$. Figure 7a illustrates this scenario. As discussed before, we can efficiently obtain the values of $dist(q, p^+)$ and τ that are used in the lemma.

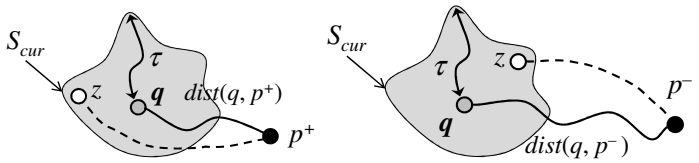
Lemma 6. Pruning a result by using τ
Given a result $p^+ \in R$, if $dist(q, p^+) + \tau \leq r$, then p^+ cannot contribute to the current safe region $\mathcal{S}_{cur}$.

Proof. Let z be any location on any segment in $\mathcal{S}_{cur}$. By triangular inequality (Lemma 2), we have: $dist(z, p^+) \leq dist(z, q) + dist(q, p^+)$. Since $\tau \geq dist(z, q)$ (by Definition 5), we then derive: $dist(z, p^+) \leq \tau + dist(q, p^+)$. By using the given condition $dist(q, p^+) + \tau \leq r$, we obtain: $dist(z, p^+) \leq r$. Thus, $z \in \Upsilon(p^+, r)$. We conclude that $\mathcal{S}_{cur} \subseteq \Upsilon(p^+, r)$, so p^+ cannot contribute to $\mathcal{S}_{cur}$. □

For the case of a non-result p^-, we utilize Lemma 7 to determine whether p^- is sufficiently far from $\mathcal{S}_{cur}$. If so, the range $\Upsilon(p^-, r)$ also cannot help shrink $\mathcal{S}_{cur}$. Figure 7b depicts this situation. Observe that $dist(q, p^-)$ and τ used in the lemma can be retrieved quickly.

Lemma 7. Pruning a non-result by using τ
Given a non-result $p^- \in NR$, if $dist(q, p^-) - \tau > r$, then p^- cannot contribute to the current safe region $\mathcal{S}_{cur}$.

Proof. Let z be any location on any segment in $\mathcal{S}_{cur}$. By triangular inequality (Lemma 2), we have: $dist(p^-, q) \leq dist(p^-, z) + dist(z, q)$. Since $\tau \geq dist(z, q)$ (by Definition 5), we then derive: $dist(p^-, q) \leq dist(p^-, z) + \tau$. We then rearrange the terms to get: $dist(z, p^-) \geq dist(q, p^-) - \tau$. By using the given condition $dist(q, p^-) - \tau > r$, we derive: $dist(z, p^-) > r$. Thus, $z \notin \Upsilon(p^+, r)$. We conclude that $\mathcal{S}_{cur}$ does not intersect $\Upsilon(p^-, r)$. Thus, p^- cannot contribute to $\mathcal{S}_{cur}$. □



(a) a point $p^+ \in R$ (b) a point $p^- \in NR$

Figure 7: Pruning Data Points

5.3. Ordering of the Retrieved Point Sets

According to Equation 3, the correctness of the safe region is preserved regardless of the ordering of processing the data points. However, different orderings lead to different rates of refining the current safe region $\mathcal{S}_{cur}$. Obviously, the smaller the $\mathcal{S}_{cur}$, the higher chance that subsequent encountered points can be discarded (by pruning rules in Section 5.2). If we choose an appropriate ordering that helps refine $\mathcal{S}_{cur}$ to a small region at a fast rate, then the power of pruning rules will be improved significantly, enabling more irrelevant data points to be discarded, thus reducing their processing cost.

We propose the following algorithm as the implementation of `PickPoint` (used in Algorithm 1). Recall that the sets R and NR store result points and non-result points, respectively. Their data points can be used for refining the safe region $\mathcal{S}_{cur}$. First, we employ a set-based ordering for choosing a set (R or NR) to be considered (at Line 1). Second, we pick the next point p from the chosen set (at Line 2). Finally, we return p to the caller for processing.

Algorithm 4 `PickPoint`Algorithm

Algorithm `PickPoint`(Point set R, Point set NR)
1: pick a point set (R or NR) to be processed;
2: pick a point p from the chosen set;
3: **return** p;

Ordering of Sets R and NR.
We first investigate set-based orderings (for Line 1). Observe that the sets R and NR store result points and non-result points, respectively. We propose two intuitive orderings for choosing a set to be considered next.

- **Process all result points before any non-result point**
  This is a simple strategy. However, it suffers from the problem that consecutive result points could be located close together and their ranges cannot help shrink $\mathcal{S}_{cur}$ much.

- **Round robin.**
  This strategy considers the set R and NR in alternate manner. If the next non-result point to be processed is close to a previous result point that was processed, then $\mathcal{S}_{cur}$ can be shrunk to a small region.

We illustrate the above set-based orderings with the example of Figure 8. Let us consider the simple set-based ordering that considers all result points before any non-result point. Suppose that we examine the points in the order $p_1, p_3, p_2, p_4, p_5, p_6$. We first process p_1 and then p_3. The current safe region $\mathcal{S}_{cur}$ is the intersection between $\Upsilon(p_1, r)$

and $\Upsilon(p_3, r)$, which is large. The next result point to be processed is p_2. Subsequent points (e.g., p_2, p_4, p_5) can refine $\mathcal{S}_{cur}$ so they cannot be pruned.

We then examine the round robin ordering. Suppose that we examine the points in the order p_1, p_4, p_3, p_5, p_2, p_6. We first process result point p_1 and then non-result point p_4. The current safe region $\mathcal{S}_{cur}$ is the set difference between $\Upsilon(p_1, r)$ and $\Upsilon(p_4, r)$, which is small. Next, we process result point p_3 and then non-result point p_5. Now, $\mathcal{S}_{cur}$ already becomes the final safe region. As a result, the subsequent point (e.g., p_6) can be pruned easily.

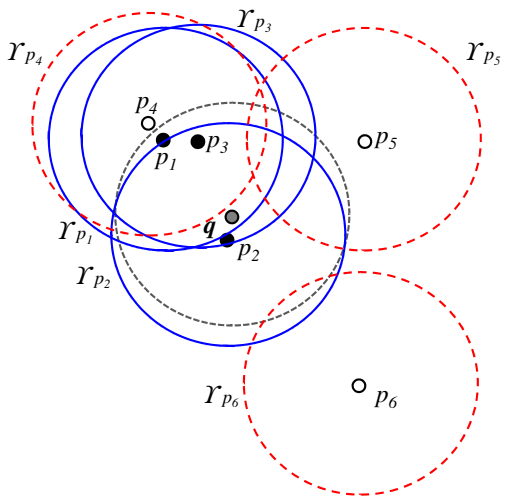


Figure 8: Example of Ordering Data Points

Ordering of Points within R and NR.
We then study point-basted orderings (for Line 2). Ideally, if we know the distances between all pairs of points in R and NR, then we have the luxury of choosing the optimal ordering that leads to rapid shrinking of the current safe region $\mathcal{S}_{cur}$. For instance, one could choose a pair of result points that are far apart (e.g., p_1 and p_2), and compute $\mathcal{S}_{cur}$ as $\Upsilon(p_1, r) \cap \Upsilon(p_2, r)$, which is small. Alternatively, one could choose a pair of result point and non-result point (e.g., p_1 and p_4) that are close together, and compute $\mathcal{S}_{cur}$ as $\Upsilon(p_1, r) - \Upsilon(p_4, r)$, which is even smaller. However, such an ordering scheme may not be feasible because it incurs expensive network expansions to obtain such distances for all pairs of points.

We propose to design practical ordering heuristics that incur a small computation overhead for ordering the points (within R and NR). Specifically, our proposed orderings exploit certain "already-known" distances obtained from the range search operation (Line 1) of Algorithm 1, where we already know the distance from q to each point p (in R or NR). Thus, we propose two orderings for picking the next result $p^+ \in R$:

- **In ascending order of distance** $dist(q, p^+)$
  If we first process a result point that is close to q, then the non-result point to be processed next is likely to be far from the above result point. It is because all non-result points are located away from q by at least distance r.

- **In descending order of distance** $dist(q, p^+)$
  If we first process a result point that is far from q, then the non-result point to be processed next is likely near to the above result point. Thus, the set difference of their ranges lead to a small $\mathcal{S}_{cur}$.

Consider the example of Figure 8. Suppose we first process result point p_2 which is the closest to q. As we adopt round robin, the next non-result point to be processed can be p_4,p_5 or p_6. No matter which non-result point we choose, the resulting current safe region will be big. On the contrary, if we first process result point p_1, which is the furthest from q, we can choose the non-result point p_4 to process. The resulting current safe region is small. A small current safe region implies a higher chance that the next retrieved point can be pruned.

Similarly, we propose two orderings for picking the next non-result $p^- \in NR$:

- **In ascending order of distance** $dist(q, p^-)$
- **In descending order of distance** $dist(q, p^-)$

Unlike result points, we recommend to examine non-result points in the ascending order of their distances from q. This is because non-result points that are near to q are also near to result points. If we process a non-result point that is near to any processed result point, then the current safe region can be shrunk rapidly. Consider the example of Figure 8. Suppose that we have chosen the result point p_2 and compute the current safe region $\mathcal{S}_{cur}$ as $\Upsilon(p_2, r)$. If we choose point p_6 as the next non-result point, then $\mathcal{S}_{cur}$ can only be shrunk slightly. In contrast, if we choose point p_4 as the next non-result point, then $\mathcal{S}_{cur}$ can be shrunk rapidly.

6. Performance Study

In this section we present the result of a comprehensive performance study. In the experiments, we use the USA road network (175K nodes, 179K edges) obtained from [www.maproom.psu.edu/dcw/]. The movement of a client follows paths randomly generated by the random trip road model (RTR). In that model, the speed of the client remains constant on the same edge but it may be different on different edges [20]. Each client trip consists of location records at 100,000 timestamps, where the time duration between adjacent timestamps is 1 second. The default number of POIs is set to the number of gas stations in the U.S.[1] (121,446). We implemented all solutions in C++ and conducted experiments on a 2.5GHz Linux machine with 8GB memory. We simulate both the client and the server on this machine. In each experiment, we only vary the value of one parameter, while fixing the other parameters to their default values as listed in Table 7.

We evaluate the performance of our solution using the following measures:

- (i) the total communication cost, which is the total number of points (both safe exit points and data points) sent by the server;
- (ii) the total communication frequency, which is the number of messages sent from the client to the server;
- (iii) the total server CPU time; and
- (iv) the total client CPU time.

As a remark, the total client CPU time is taken as the total execution time for the client to process the entire trip (with 100,000 timestamps). The server CPU time is measured as the total execution time of the server on the entire trip.

For comparison, we also report the performance of the periodic querying approach (PERIOD) and a basic version (BASIC) of our solution. PERIOD issues a new range query to the server at every timestamp. BASIC uses the same Phase 1 (i.e., retrieving useful points) as our advanced (ADV) solution but it turns off all the pruning rules (therefore in Phase 2 it examines all useful points obtained from Phase 1).

In the following, we first study the effectiveness of our pruning rules (Section 6.1) and ordering heuristics (Section 6.2). Then, we examine the performance of PERIOD, BASIC, and ADV, under different client moving speeds (Section 6.3), different query ranges (Section 6.4), and different data densities (Section 6.5).

6.1. Effectiveness of Pruning Rules

In this experiment, we study the effectiveness of pruning rules (used in ADV) by enabling/disabling them. Table 8 shows the performance for different combinations of pruning rules. The performance measurement includes the total number of pruned result points and non-result points, the total number of nodes visited, and the total server CPU time.

Lemma 7 prunes non-result points that cannot contribute to safe exits. This lemma is powerful as it cuts the server CPU time by 82% when other pruning rules are in place (comparing cases 1 and 3). It can still save 62% of the server CPU time when other pruning rules are not in place (comparing cases 5 and 8).

[1] U.S. Economic Census. http://www.census.gov/econ/census02/data/us/US000_44.htm

Table 7: Default Parameter Value

Parameter	Value
client speed	50 km/hour
query range	10 km
number of POIs (=number of gas stations in the U.S.)	121,446
heuristics for examining result points and non-result points	round-robin
heuristics for ordering result points	points farthest from q first
heuristics for ordering non-result points	points closest to q first

Lemma 6 prunes any result point that cannot alter safe exits. It is also effective and reduces the server CPU time by 64% when other pruning rules are in place (comparing cases 1 and 4). Even when other pruning rules are not in place, it saves 26% of the server CPU time (see cases 6 and 8).

Since the size of non-results (NR) is much larger than results (R), it is more effective to prune non-results than results. Thus, Lemma 7 achieves lower server CPU time than Lemma 6 (comparing cases 5 and 6). Observe that the reduction in results and non-results translate to a saving in the number of visited nodes as well; both cases 5 and 6 have fewer visited nodes than case 8.

The time spent on traversing the road network can be reduced by applying Lemma 5. When the other pruning rules are in place, it can save the number of visited nodes by 6.5% (comparing cases 1 and 2). It achieves an even larger reduction of 18.5% when other pruning rules are not in place (comparing cases 7 and 8). However, the server CPU time reduction of this pruning rule is not as substantial as the pruning of points.

Case	Combination			Performance			
	Pruning Result Points (Lemma 6)	**Pruning Non-Result Points** (Lemma 7)	**Restricting Traversal** (Lemma 5)	**Num. Pruned Result**	**Num. Pruned Non-Result**	**Num. Node Visited**	**Server Time (sec)**
1	yes	yes	yes	15,769	56,024	275,352	12.72
2	yes	yes	no	15,769	56,024	294,723	12.88
3	yes	no	yes	15,769	0	403,413	71.20
4	no	yes	yes	0	56,024	474,366	35.69
5	no	yes	no	0	56,024	665,637	37.12
6	yes	no	no	15,769	0	1,537,928	73.68
7	no	no	yes	0	0	1,555,262	97.92
8	no	no	no	0	0	1,908,842	99.11

Table 8: Effect of Different Pruning Rules

6.2. Effect of Different Ordering Heuristics

We then investigate the effect of different ordering heuristics and show the performance of various combinations in Table 9. Note that it is always better to apply round-robin (examining results and non-results alternatively) than processing all results first. The reason is that the round-robin heuristic exploits the non-result points to restrict the safe region effectively. Also, as expected, it is more beneficial to process result points that are far away from q first and process non-result points that are closer to q first. Overall, the winners above are the chosen heuristics in our proposed algorithm ADV.

6.3. Effect of Speed

We proceed to examine the effect of the speed of the moving client on the performance of PERIOD, BASIC, and ADV. Figure 9 shows the total communication cost, total communication frequency, total server CPU time, and total client CPU time, with respect to different client moving speeds (ranging from 10km/hr to 100km/hr). PERIOD incurs constant communication cost and frequency because it issues queries to the server periodically (regardless of the client moving speed). However, PERIOD cannot guarantee the correctness of the result at anytime and it suffers

Table 9: Effect of Different Ordering Heuristics

Case	Combination			Performance		
	Set Ordering	**Result Ordering**	**Non-result Ordering**	**Num. Pruned Result.**	**Num. Pruned Non-Result.**	**Server Time (sec)**
1	round robin	far to near	near to far	15,769	56,024	12.99
2	all result points first	far to near	near to far	14,441	56,924	13.81
3	round robin	far to near	far to near	14,454	51,767	19.27
4	all result points first	far to near	far to near	14,441	56,024	37.12
5	round robin	near to far	far to near	68	53,430	40.10
6	all result points first	near to far	far to near	0	56,024	40.13
7	round robin	near to far	near to far	7,753	53,058	28.07
8	all result points first	near to far	near to far	0	56,924	36.27

from stale results at high speed. BASIC and ADV outperform PERIOD by an order of magnitude in terms of the communication cost and frequency. When the speed increases, the client reaches a safe exit sooner, and thus both the communication cost and frequency increase. The trend in Figure 9c follows that of Figure 9b because the number of safe exit computations at the server-side equals to the communication frequency. Since ADV has incorporated various optimization techniques (e.g., network traversal restriction, pruning rules, access orderings), its server CPU time is much smaller than BASIC. As shown in Figure 9d, all approaches have negligible client-side overhead (a total of 0.1 second for a trip with 100,000 timestamps). The client time appears bumpy just because the measured time is tiny and it is slightly affected by some environment/platform factors.

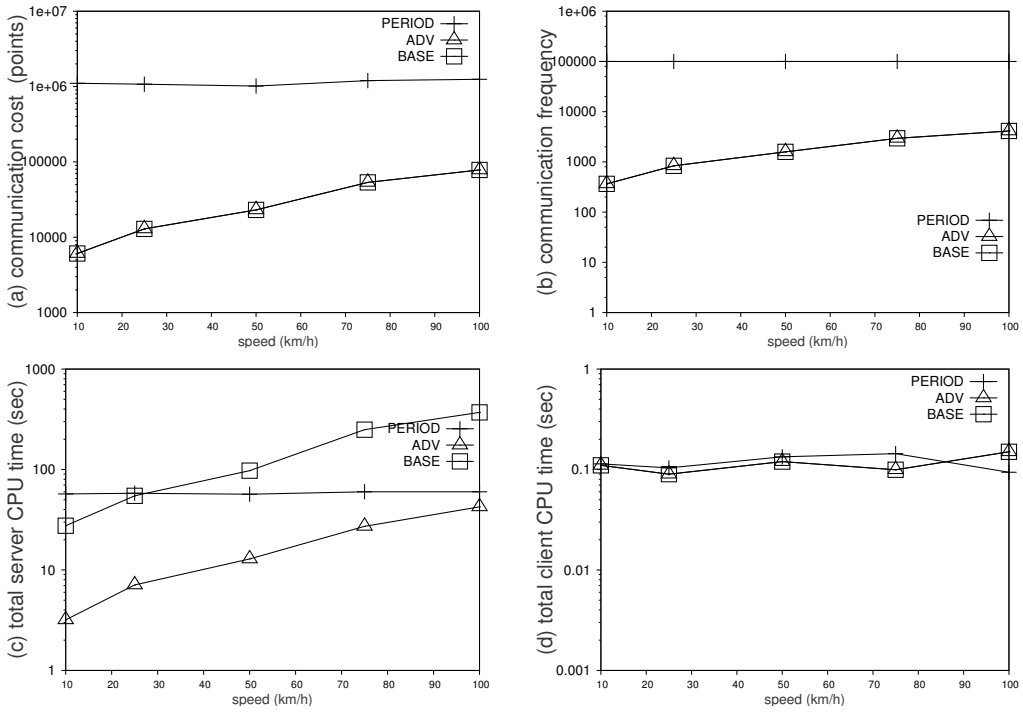


Figure 9: Varying query speed

6.4. Effect of Query Range

We next examine the effect of the query range on the performance of PERIOD, BASIC, and ADV. Figure 10 shows the total communication cost, total communication frequency, total server CPU time, and total client CPU time, as a

function of the query range. The safe exit methods (BASIC and ADV) outperform PERIOD significantly in terms of communication cost and frequency. Even though PERIOD has a constant communication frequency, the result size increases with the query range so the total communication cost also rises. For BASIC and ADV, the number of POIs in the query result increases with the query range. The intersection of ranges of these result points becomes small, rendering a small safe region and high communication frequency. ADV achieves a much smaller server CPU time than BASIC because of the optimizations used in ADV. Again, all approaches have negligible client CPU time.

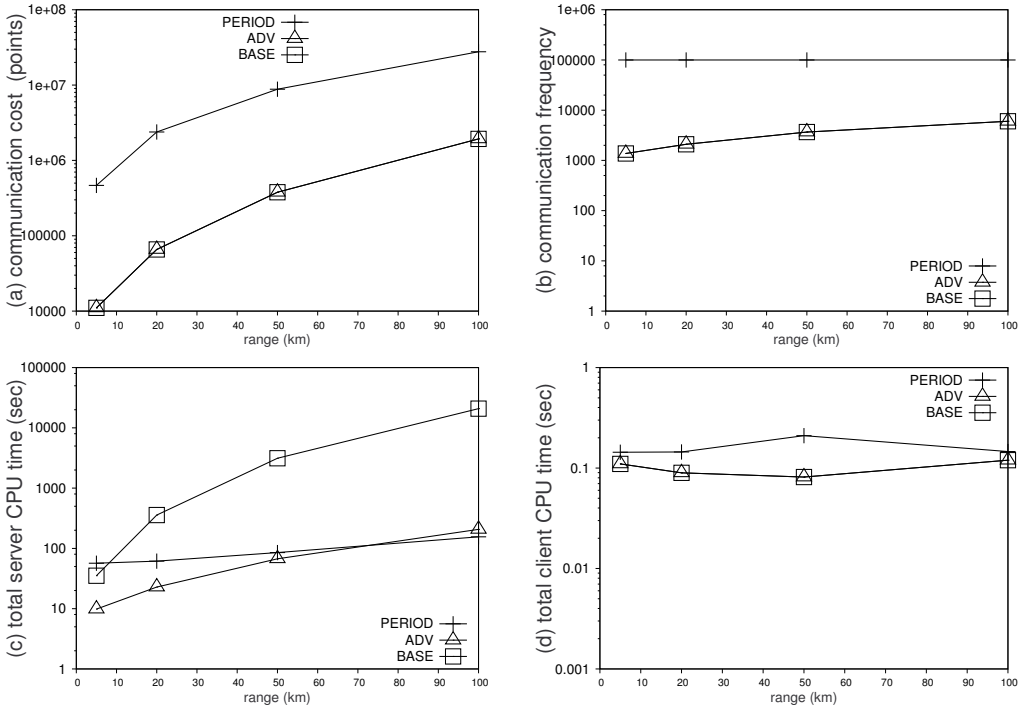


Figure 10: Varying query range

6.5. *Effect of Data Density*

We capture the data density by the number of point-of-interests (POIs) in the synthetic datasets. Figure 11 plots the cost of these methods with respect to the data density. Again, the safe exit methods achieve much lower communication cost and frequency than PERIOD. Note that the result size increases proportionally with the data density. Thus, the total communication cost of PERIOD rises correspondingly. Also, the methods BASIC and ADV obtain smaller safe regions and thus incur higher communication cost and frequency. Regarding the computation cost, ADV has the best server CPU time whereas all methods have tiny client CPU time.

7. Conclusion

This paper studies the processing of moving range queries on road networks. We propose the concept of safe exits as concise representations of the safe regions for such queries. We then present an algorithmic framework to compute safe exits efficiently, and develop several pruning rules and heuristics to avoid processing irrelevant point-of-interest and graph nodes.

The experimental study on real datasets show that our pruning rules significantly reduce the number of data points and network nodes to be processed, and our ordering heuristics help compute safe exits rapidly. Our best solution (ADV) incorporates these pruning rules and ordering heuristics. Experimental results show that the communication cost of ADV is much lower than PERIOD by 1-2 orders of magnitude. Also, ADV outperforms BASIC significantly

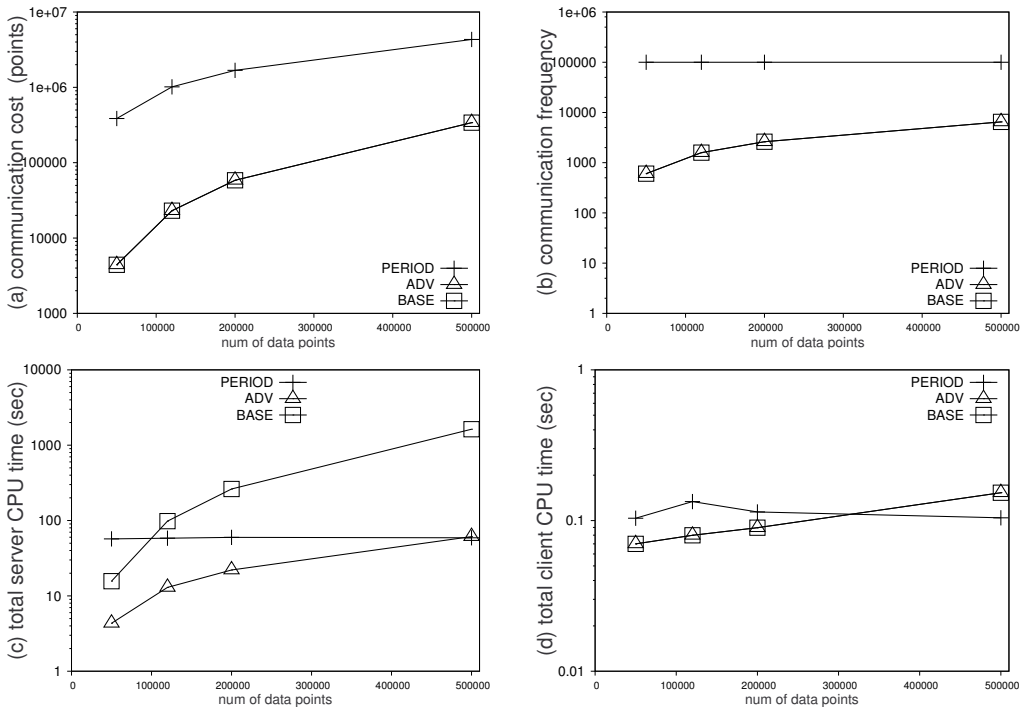


Figure 11: Varying POIs

in terms of server CPU time. We recommend ADV as the best solution because of its low communication cost for the client and low computation time at the server. It is desirable for the real-world scenario where mobile devices have limited communication bandwidth and the server aims at achieving high throughput.

There are several promising directions for future research. First, we plan to study moving k-NN queries on road networks. Although the safe region of a moving k-NN query can also be represented by safe exits in a compact manner, it remains an open question to compute such safe exits efficiently. Unlike range queries, different query locations can have different kNN distances and we will develop alternative pruning rules to tackle this problem. Second, we will study moving queries over incomplete spatial data objects collected from multiple sources [4]. As some data objects may not have exact locations, it is important to deduce their approximate locations and provide accuracy bounds on query results. Third, we will consider influence range queries over a spatial dataset with ratings (e.g., restaurants) [29]. The relevance of an object to query result is determined by both its rating and distance from user. The challenge is that our existing pruning distance bounds no longer hold for influence range queries and we need to design efficient processing techniques for them.

Acknowledgement

This work was supported by ICRG grants A-PJ79 and G-U807 from the Hong Kong Polytechnic University.

References

[1] R. Benetis, C. S. Jensen, G. Karciauskas, and S. Saltenis. Nearest and Reverse Nearest Neighbor Queries for Moving Objects. *VLDB J.*, 15(3):229–249, 2006.

[2] M. A. Cheema, L. Brankovic, X. Lin, W. Zhang, and W. Wang. Multi-Guarded Safe Zone: An Effective Technique to Monitor Moving Circular Range queries. In *ICDE*, pages 189–200, 2010.

[3] M. A. Cheema, L. Brankovic, X. Lin, W. Zhang, and W. Wang. Continuous Monitoring of Distance-Based Range Queries. *IEEE TKDE*, 23(8):1182–1199, 2011.

[4] A. Cuzzocrea and A. Nucita. Enhancing Accuracy and Expressive Power of Range Query Answers over Incomplete Spatial Databases via a Novel Reasoning Approach. *Data Knowl. Eng.*, 70(8):702–716, 2011.

[5] E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959.

[6] A. V. Goldberg and C. Harrelson. Computing the Shortest Path: A* Search Meets Graph Theory. In *SODA*, pages 156–165, 2005.

[7] Y.-L. Hsueh, R. Zimmermann, and W.-S. Ku. Efficient Location Updates for Continuous Queries over Moving Objects. *J. Comput. Sci. Technol.*, 25(3):415–430, 2010.

[8] H. Hu, D. L. Lee, and V. C. S. Lee. Distance Indexing on Road Networks. In *VLDB*, pages 894–905, 2006.

[9] G. S. Iwerks, H. Samet, and K. P. Smith. Maintenance of K-nn and Spatial Join Queries on Continuously Moving Points. *ACM TODS*, 31(2):485–536, 2006.

[10] N. Jing, Y.-W. Huang, and E. A. Rundensteiner. Hierarchical Encoded Path Views for Path Query Processing: An Optimal Model and Its Performance Evaluation. *IEEE TKDE*, 10(3):409–432, 1998.

[11] S. Jung and S. Pramanik. An Efficient Path Computation Model for Hierarchically Structured Topographical Road Maps. *IEEE TKDE*, 14(5):1029–1046, 2002.

[12] M. R. Kolahdouzan and C. Shahabi. Voronoi-Based K Nearest Neighbor Search for Spatial Network Databases. In *VLDB*, pages 840–851, 2004.

[13] M. R. Kolahdouzan and C. Shahabi. Alternative Solutions for Continuous K Nearest Neighbor Queries in Spatial Network Databases. *GeoInformatica*, 9(4):321–341, 2005.

[14] H.-P. Kriegel, P. Kröger, and M. Renz. Continuous Proximity Monitoring in Road Networks. In *GIS*, page 12, 2008.

[15] H.-P. Kriegel, P. Kröger, M. Renz, and T. Schmidt. Hierarchical Graph Embedding for Efficient Query Processing in Very Large Traffic Networks. In *SSDBM*, pages 150–167, 2008.

[16] N. Linial, E. London, and Y. Rabinovich. The Geometry of Graphs and Some of its Algorithmic Applications. *Combinatorica*, 15(2):215–245, 1995.

[17] F. Liu, T. T. Do, and K. A. Hua. Dynamic Range Query in Spatial Network Environments. In *DEXA*, pages 254–265, 2006.

[18] S. Nutanong, R. Zhang, E. Tanin, and L. Kulik. The V*-Diagram: A Query-Dependent Approach to Moving KNN Queries. *PVLDB*, 1(1):1095–1106, 2008.

[19] D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao. Query Processing in Spatial Network Databases. In *VLDB*, pages 802–813, 2003.

[20] P. Pesti, L. Liu, B. Bamba, A. Iyengar, and M. Weber. RoadTrack: Scaling Location Updates for Mobiles on Road Networks with Query Awareness. *PVLDB*, 3(2):1493–1504, 2010.

[21] H. Samet, J. Sankaranarayanan, and H. Alborzi. Scalable Network Distance Browsing in Spatial Databases. In *SIGMOD*, pages 43–54, 2008.

[22] C. Shahabi, M. R. Kolahdouzan, and M. Sharifzadeh. A Road Network Embedding Technique for k-Nearest Neighbor Search in Moving Object Databases. In *ACM-GIS*, pages 94–10, 2002.

[23] Z. Song and N. Roussopoulos. K-Nearest Neighbor Search for Moving Query Point. In *SSTD*, pages 79–96, 2001.

[24] D. Stojanovic, A. N. Papadopoulos, B. Predic, S. Djordjevic-Kajan, and A. Nanopoulos. Continuous Range Monitoring of Mobile Objects in Road Networks. *Data Knowl. Eng.*, 64(1):77–100, 2008.

[25] Y. Tao, D. Papadias, and Q. Shen. Continuous Nearest Neighbor Search. In *VLDB*, pages 287–298, 2002.

[26] H. Wang and R. Zimmermann. Snapshot Location-Based Query Processing on Moving Objects in Road Networks. In *GIS*, page 50, 2008.

[27] H. Wang and R. Zimmermann. Processing of Continuous Location-Based Range Queries on Moving Objects in Road Networks. *IEEE TKDE*, 23(7):1065–1078, 2011.

[28] K. Xuan, G. Zhao, D. Taniar, and B. Srinivasan. Continuous Range Search Query Processing in Mobile Navigation. In *ICPADS*, pages 361–368, 2008.

[29] M. L. Yiu, H. Lu, N. Mamoulis, and M. Vaitis. Ranking Spatial Data by Quality Preferences. *IEEE TKDE*, 23(3):433–446, 2011.

[30] J. Zhang, M. Zhu, D. Papadias, Y. Tao, and D. L. Lee. Location-based Spatial Queries. In *SIGMOD*, pages 443–454, 2003.

[31] B. Zheng and D. L. Lee. Semantic Caching in Location-Dependent Query Processing. In *SSTD*, pages 97–116, 2001.