Singapore Management University

Institutional Knowledge at Singapore Management University

Research Collection School Of Information Systems

School of Information Systems

10-2011

Direction-Based Surrounder Queries for Mobile Recommendations

Xi GUO
Nagoya University

Baihua ZHENG
Singapore Management University, bhzheng@smu.edu.sg

Yoshiharu ISHIKAWA
Nagoya University

Yunjun GAO
Zhejiang University

Follow this and additional works at: https://ink.library.smu.edu.sg/sis_research

Part of the Databases and Information Systems Commons, and the Numerical Analysis and Scientific Computing Commons

Citation

GUO, Xi; ZHENG, Baihua; ISHIKAWA, Yoshiharu; and GAO, Yunjun. Direction-Based Surrounder Queries for Mobile Recommendations. (2011). *VLDB Journal*. 20, (5), 743-766. Research Collection School Of Information Systems.

Available at: https://ink.library.smu.edu.sg/sis_research/1409

This Journal Article is brought to you for free and open access by the School of Information Systems at Institutional Knowledge at Singapore Management University. It has been accepted for inclusion in Research Collection School Of Information Systems by an authorized administrator of Institutional Knowledge at Singapore Management University. For more information, please email library@smu.edu.sg.

Noname manuscript No.
(will be inserted by the editor)

Direction-Based Surrounder Queries for Mobile Recommendations

Xi Guo · Baihua Zheng · Yoshiharu Ishikawa · Yunjun Gao

Received: date / Accepted: date

Abstract Location-based recommendation services recommend objects to the user based on the user's preferences. In general, the nearest objects are good choices considering their spatial proximity to the user. However, not only the distance of an object to the user but also their *directional relationship* are important. Motivated by these, we propose a new spatial query, namely a *direction-based surrounder* (*DBS*) query, which retrieves the nearest objects around the user from different directions. We define the DBS query not only in a two-dimensional Euclidean space $\mathbb{E}$ but also in a road network $\mathbb{R}$. In the Euclidean space $\mathbb{E}$, we consider two objects a and b are *directional close* w.r.t. a query point q iff the included angle $\angle aqb$ is bounded by a threshold specified by the user at the query time. In a road network $\mathbb{R}$, we consider two objects a and b are *directional close* iff their shortest paths to q overlap. We say object a *dominates* object b iff they are directional close and meanwhile a is closer to q than b. All the objects that are not dominated by others based on the above dominance relationship constitute *direction-based surrounders* (*DBSs*). In this paper, we formalize the DBS query, study it in both the snapshot and continuous settings, and conduct extensive experiments with both real and synthetic datasets to evaluate our proposed algorithms. The experimental results demonstrate that the proposed algorithms can answer DBS queries efficiently.

Keywords Spatial database · Surrounder query · Location-based recommendation · Direction

Xi Guo, Yoshiharu Ishikawa
Graduate School of Information Science, Nagoya University, Furo-cho, Chikusa-ku, Nagoya, Japan 464-8601.
Tel.: +81-052-789-3306, Fax: +81-052-789-3306
E-mail: guoxi@db.itc.nagoya-u.ac.jp, y-ishikawa@nagoya-u.jp

Yunjun Gao (corresponding author)
College of Computer Science, Zhejiang University, Hangzhou 310027, P. R. China.
Tel.: +86-571-87651613, Fax: +86-571-87951250
E-mail: gaoyj@zju.edu.cn

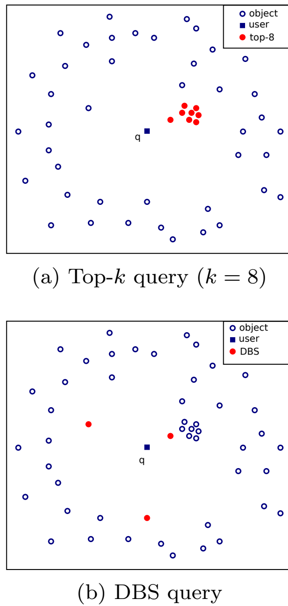
Baihua Zheng
School of Information Systems, Singapore Management University, Stamford Road 80, 178902, Singapore.
Tel.: +65-68280915, Fax: +65-68280919
E-mail: bhzheng@smu.edu.sg

1 Introduction

In location-based services such as mobile recommendations and car navigation, a mobile user often receives the recommendations of *POI* (point of interest) objects based on spatial closeness and the user's preference [14] using some popular mobile queries (e.g., nearest neighbour queries and range queries). For example, "show me the eight nearest convenience stores" is a top-8 nearest neighbour query and "show me the convenience stores within 400 meters" is a circular range query. However, such conventional spatial queries may not be helpful when we want to recommend neighbourhood information around the user.

An example is depicted in Fig. 1. Fig. 1(a) shows the result of a top-8 query. It is observed that all the returned POI objects are located in the north east of q. If the user intends to move to reverse direction (e.g., south), the answer objects are not useful at all. In other words, the usefulness of returned POI objects not only depends on their distances to the user but also their directions to the user. To support object evaluation based on both proximity and direction of POI objects w.r.t. a specified query point, we propose *direction-based surrounder* (*DBS*) queries in this paper. An example DBS

query is depicted in Fig. 1(b) where the three nearest objects surrounding q are returned. Compared with top-k search, DBS retrieves objects that are located in different directions of q and hence it provides a better overview of the surrounding area.



(a) Top-k query ($k = 8$)

(b) DBS query

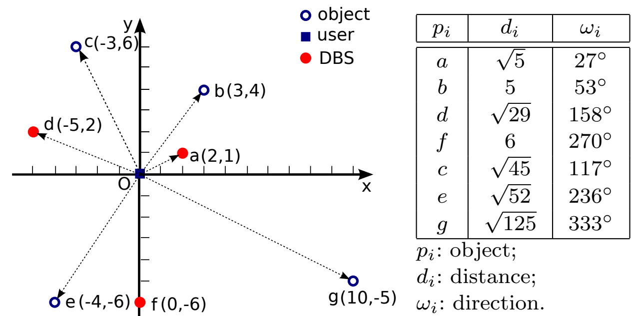
Fig. 1: Motivating example

As shown in Fig. 1(b), a DBS query evaluates objects in a space $\mathbb{X}$ based on not only their distances to the query point but also their direction relationships with the query point. The basic idea is that, for a given query point q, an object p_i is a better candidate than another object p_j if p_i is closer to q and they are *directional close* w.r.t. q in the space $\mathbb{X}$.

In this paper, we consider DBS queries in both a vector space $\mathbb{E}$ and a metric space $\mathbb{R}$ ($\mathbb{X} \in \{\mathbb{E}, \mathbb{R}\}$). For the vector space $\mathbb{E}$, we assume that objects of interest are in the two dimensional Euclidean space and the corresponding queries are called $\mathbb{E}$-*DBS queries*. For the metric space $\mathbb{R}$, we assume that the objects are in a road network and the corresponding queries are called $\mathbb{R}$-*DBS queries*.

For $\mathbb{E}$-DBS queries, we measure the distance and the direction of an object p_i w.r.t. q using the vector which originates from q and ends in p_i. For $\mathbb{R}$-DBS queries, we measure the distance and the direction of an object p_i w.r.t. q based on the shortest path from q to p_i. Typically, when an exact road network is available, an $\mathbb{R}$-DBS query is an appropriate choice for the user. When road network information is not available or not useful (e.g., shopping in a small city area), an $\mathbb{E}$-DBS query would be a good choice.

Before presenting the formal definition of a DBS query, we use Example 1 and Example 2 to illustrate DBS queries in the Euclidean space $\mathbb{E}$ and in a road network $\mathbb{R}$, respectively. They also serve as the running examples in this paper.

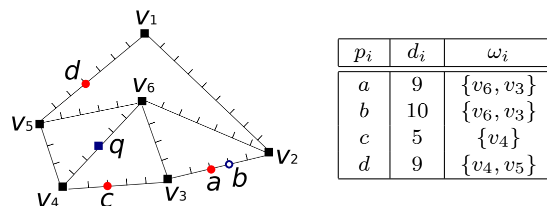


p_i	d_i	ω_i
a	$\sqrt{5}$	27°
b	5	53°
d	$\sqrt{29}$	158°
f	6	270°
c	$\sqrt{45}$	117°
e	$\sqrt{52}$	236°
g	$\sqrt{125}$	333°

p_i: object;
d_i: distance;
ω_i: direction.

Fig. 2: Example of an $\mathbb{E}$-DBS ($\theta = \pi/3$)

Example 1 In Fig. 2, there are seven POIs (a to g) around the user O. We use the vectors $\overrightarrow{a}, \ldots, \overrightarrow{g}$ originating from O to denote the distance d_i and the direction ω_i of a POI object i w.r.t. O. We assume two POI objects i and j are *directional close* if the included angle $\angle iOj$ is bounded by θ ($= \pi/3$) specified by the user. For example, a and b are directional close as $|\omega_a - \omega_b| = 26° < \pi/3$, but objects a and d are not. Given two objects i and j, i *dominates* j iff they are directional close and i has a shorter distance to the user than j does, i.e., $d_i < d_j$. The $\mathbb{E}$-DBS query retrieves all the POI objects that are not dominated by others. Notice that the number of objects returned is affected by the value of θ. In our example, objects a, d, and f are the result. ■



p_i	d_i	ω_i
a	9	$\{v_6, v_3\}$
b	10	$\{v_6, v_3\}$
c	5	$\{v_4\}$
d	9	$\{v_4, v_5\}$

Fig. 3: Example of an $\mathbb{R}$-DBS

Example 2 In Fig. 3, a road network is represented by a graph with six vertices $V = \{v_1, \cdots, v_6\}$ and nine edges. The shortest path from q to a passes vertex v_6 first and then vertex v_3 to reach a, i.e., $q \to v_6 \to v_3 \to a$, denoted as $SP(q, a) = \{v_6, v_3\}$. Here, the distance

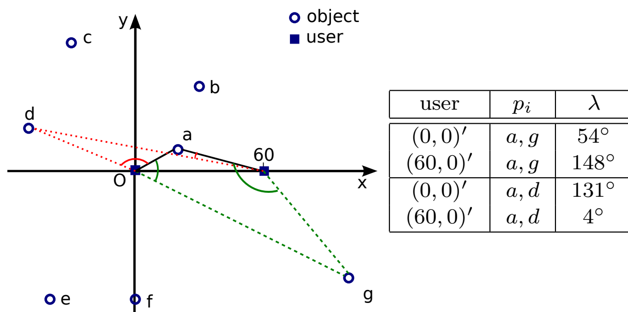
of a to q is set to the length of the shortest path, and the direction of a to q is set to the shortest path itself. Following previous notations, d_i and ω_i of object p_i are illustrated in Fig. 3.

We assume two objects are *directional close* if their shortest paths overlap. Object a dominates object b, because they are directional close and meanwhile $|SP(q,a)|$ $(=9) < |SP(q,b)| (=10)$; but object c does not dominate a because they are not directional close. Objects a, c, and d are the $\mathbb{R}$-DBS objects. ■

A DBS query is a new multi-objective optimization problem focusing on the spatial context. We evaluate the dominance relationship between objects based on both *distances* and *directions*. Its formal definition will be presented in Section 3. In order to support DBS query in both the static scenario and the dynamic mobile scenario, we form *snapshot DBS queries* and *continuous DBS queries*.

A *snapshot DBS query* finds out the DBS objects according to the user's current position. Example 1 and Example 2 present examples of snapshot DBS queries in the Euclidean space $\mathbb{E}$ and in a road network $\mathbb{R}$, respectively. The purpose of snapshot DBS queries is to provide the user with the current "best view" and to enable the user to identify the best POI for each direction. A naïve solution is to check objects one by one to determine whether they are dominated by others. However, this brute force based approach is very inefficient as it needs to consider the entire object set. Alternatively, we propose new approaches which can answer snapshot DBS queries efficiently by utilizing some unique properties of DBS.

On the other hand, a *continuous DBS query* retrieves the DBS objects while the user is moving linearly. It is typically used to predict when and how the best view (i.e., the DBS) changes while the user is moving. Example 3 and Example 4 are extended from original Example 1 and Example 2 to illustrate the idea of continuous $\mathbb{E}$-DBS queries and continuous $\mathbb{R}$-DBS queries, respectively.



user	p_i	λ
$(0,0)'$	a,g	54°
$(60,0)'$	a,g	148°
$(0,0)'$	a,d	131°
$(60,0)'$	a,d	4°

Fig. 4: Example of a continuous $\mathbb{E}$-DBS query ($\theta = \pi/3$)

Example 3 As shown in Fig. 4, we assume a user currently located at the position $(60,0)'$ is moving linearly along the x-axis. We list the included angle (denoted as λ) between object a and object g and that between object a and object d when the user is at $(0,0)'$ and $(60,0)'$, respectively in Fig. 4 to demonstrate the dynamic nature of the included angles when user keeps moving.

Object g, which is dominated by a when the user locates at $(0,0)'$, is not dominated by a when user moves to $(60,0)'$ because they are in the different direction w.r.t. the user. On the other hand, object d, which is an DBS object when the user locates at $(0,0)'$, is dominated by a when user moves to $(60,0)'$. Thus, DBS points (i.e., $\{a,g\}$) corresponding to $(60,0)'$ are different from those (i.e., $\{a,d,f\}$) corresponding to $(0,0)'$. ■

Example 4 As shown in Fig. 5, a user is moving from v_6 to v_4 along the edge $e(v_6,v_4)$. The shortest path from q to a is $SP(q,a) = \{v_6, v_3\}$ when the user starts at v_6. However, it changes to $\{v_4, v_3\}$ when the user locates at v_4. Object a, which is not dominated by c when the user locates at v_6, is dominated by c when the user reaches v_4. Consequently, the DBS objects $\{c,d\}$ w.r.t. v_4 are different from the DBS objects $\{a,c,d\}$ w.r.t. v_6. ■

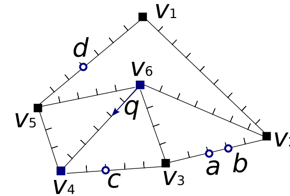


Fig. 5: Example of a continuous $\mathbb{R}$-DBS query

A critical problem in supporting continuous queries is how to update the DBS while the user is moving. A naïve solution is to issue a snapshot query whenever the user moves to a new position. However, it is impractical and is quite costly. Our alternative approach is to predict DBS changes based on pre-computations when the query is submitted. Thus, we can update the DBS result whenever the user arrives at the change position, which is predicted by the algorithm proposed in this paper.

In the following, we formalize the snapshot DBS query and continuous DBS query in both the Euclidean space $\mathbb{E}$ and the road network $\mathbb{R}$, and present the corresponding query processing algorithms. A preliminary report of this work appeared in [16]. In this paper, we extend the original work by (i) augmenting the DBS

queries considering road networks, (ii) presenting more illustrative examples, more detailed theoretical analysis, and more formal proofs; (iii) conducting a more comprehensive experimental evaluation; and (iv) including a complete review of the related work to make this paper self-contained.

The rest of this paper is organized as follows. Section 2 overviews the related work. Section 3 presents the formal definition of a DBS query. Section 4 and Section 5 elaborate the query processing algorithms for snapshot DBS query and continuous DBS query, respectively. Then, Section 6 reports the experimental results and our findings. Finally, Section 7 concludes the paper with some directions for future work.

2 Related Work

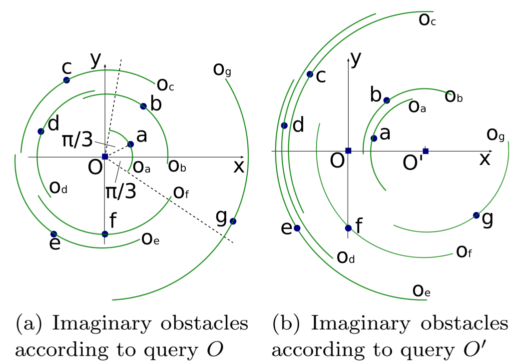
2.1 Direction-Based Spatial Queries in Euclidean Space

There are several studies that consider the direction properties in spatial databases. Among them, *visible nearest neighbor* (*VNN*) queries [5,6] and *nearest surrounder* (*NS*) queries [9] are most relevant.

Visible Nearest Neighbor Queries. Visible nearest neighbor queries are to find nearest objects that are visible (i.e., not blocked by any obstacle) to the query point [5,6]. The concept of the *invisible area* shares some similarity with the dominance region. To be more specific, an invisible area corresponding to an obstacle o is a region within which any object is not visible to the user due to the existence of o. Similarly, the dominance region of an object i is a region where all the objects are dominated by i. In other words, a VNN query does not consider objects falling inside the invisible area of any obstacle.

We can regard an $\mathbb{E}$-DBS query, proposed in this paper, as a special type of VNN queries. We consider each data point is an obstacle and derive its imaginary dominance region as a sector shape defined by the angular parameter θ. An $\mathbb{E}$-DBS query does not consider objects falling inside of the dominance region of any object. Consider, for example, Fig. 6(a). For point a, its dominance region is defined by the angle $\theta = \pi/3$. For the presentation purpose, assume that there is an imaginary arc-shaped obstacle o_a for point a. Objects b and g are invisible from the query point O because they are within the invisible region of o_a. For each POI object i considered by the $\mathbb{E}$-DBS query, we can form its imaginary obstacle o_i in a similar manner. Thus, we can transform an $\mathbb{E}$-DBS query to a VNN query.

However, we cannot directly apply the algorithms proposed for VNN queries [5,6] to DBS queries. The main reason is that the existing VNN query methods consider rectangular (or polygonal) obstacles. In Section 4, we will explain how to exploit the properties of arcs to handle our problem efficiently. Additionally, in the continuous case, the arc obstacles are dynamic due to the movement of the user. As shown in Fig. 6(b), the positions of imaginary arc obstacles change when the user moves from O to O'. Even if we can use VNN search algorithms directly to tackle our problem, we have to issue new VNN queries periodically according to the user's movement which obviously is not practical. Hence, we propose algorithms to update the DBS continuously in Section 5.



(a) Imaginary obstacles according to query O (b) Imaginary obstacles according to query O'

Fig. 6: Imaginary obstacles of $\mathbb{E}$-DBS queries

Nearest Surrounder Queries. Lee et al. [9,26,27] studied the *nearest surrounder* (*NS*) *query* for retrieving objects, each of which is a nearest neighbor of a query point according to an associated angular range. Fig. 7 shows an example of an NS query with a query point (O) and several data objects (a to i). The result set is $\{\langle a, [\alpha_1, \alpha_2)\rangle, \langle b, [\alpha_2, \alpha_3)\rangle, \langle c, [\alpha_3, \alpha_4)\rangle, \langle d, [\alpha_4, \alpha_5)\rangle, \langle e, [\alpha_5, \alpha_6)\rangle, \langle f, [\alpha_6, \alpha_1)\rangle\}$. It means that a to f are nearest neighbors of O within their associated angles. The motivation of our $\mathbb{E}$-DBS query is related to this idea. Both of them focus on providing a whole picture of nearest objects around the user. However, an $\mathbb{E}$-DBS query treats point objects and receives the angle threshold θ, while an NS query assumes rectangular data objects. [26,27] proposed efficient algorithms to answer NS queries for a moving query point and moving data objects, but they do not consider the linear movement of the user.

Other Direction-Based Queries. Patroumpas et al. proposed the notion of an *orientation-based query* which finds objects moving towards the query point [11]. Example queries include "finding the trucks moving towards the port from the west at a distance less than 2km". The basic idea is to use a *polar tree* to index the moving objects by their directions and retrieve the ob-

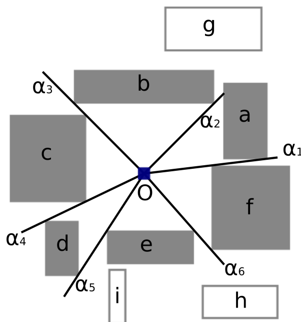


Fig. 7: Nearest surrounder query

jects within the required direction and distance ranges. Chen et al. identified the *path nearest neighbor query* which retrieves the nearest neighbor along the user's moving path [3]. To the best of our knowledge, our work is the first study of direction-based surrounder queries considering distance and direction attributes.

2.2 Nearest Neighbour Queries in Road Networks

Our $\mathbb{R}$-DBS query presents all nearest objects around a query position q considering both network distances and network directions. Nearest neighbour queries in road networks are well studied in the database area, however, these studies recommend POI objects considering the network distances only.

The snapshot $\mathbb{R}$-DBS query is related to kNN queries in road networks. Papadias et al. [40] proposed a flexible architecture for spatial network databases in order to answer spatial queries including kNN queries. Their IER/INE algorithms find out kNN objects by performing network expansions which are inspired with the Dijkstra's algorithm [28]. Kolahdouzan et al. [41] proposed VN^3 algorithms to answer kNN queries by partitioning a spatial network into smaller Voronoi polygons over objects and pre-computing some network distances. Hu et al. [33,42] build indexes to facilitate kNN search in road networks. In [33], they perform kNN searches by retrieving a set of interconnected trees which are generated from the road network. In [42], they use distance signatures to maintain approximate network distances and build an index based on the distance signatures in order to speed up kNN searches. Lee et al. [35] prune the kNN search space by skipping *Rnet* which are subspaces containing no objects.

The continuous $\mathbb{R}$-DBS query is related to continuous kNN queries for a moving query position on a query path [37,44]. Kolahdouzan et al. [44] proposed IE/UBA algorithms to find out kNN candidates first and then split the query path into sub-paths where the kNN objects are the same. Cho et al. [37] proposed UNICONS algorithms which divide the query path into valid intervals considering the network distance functions of objects w.r.t. a moving query position. In the valid intervals, the kNNs are the same no matter where the query position is. The continuous $\mathbb{R}$-DBS query is different from the continuous kNN queries for a query path. An extreme example is that the kNN queries return k objects but the $\mathbb{R}$-DBS queries find the nearest object to recommend if the shortest paths of the $k-1$ nearest objects passing by the nearest one.

There are many works studying dynamic kNN queries which are different from the continuous kNN queries. The dynamic kNN queries have dynamic objects of interest or even dynamic road networks as well as a dynamic query position. Shahabi et al. [34] focused on kNN queries for moving objects. Their RNE algorithms convert a road network to a higher-dimensional space and retrieve approximate answers in the new space with an acceptable precision. Jensen et al. [36] proposed a framework and also implemented a prototype to answer kNN queries for moving objects and a moving position. Mouratidis et al. [43] proposed IMA/GMA algorithms which update results when the changes on objects, query positions, and edges may influence the current results. Demiryurek et al. [39] proposed more efficient algorithms to solve the same problems in [43]. Their ER-CkNN algorithms avoid blind network expansions in [39] by finding candidates which are selected based on their Euclidean distances to the query position. Samet et al. [38] proposed efficient algorithms to answer kNN queries when many different queries are issued or different sets of objects are used for a static road networks. They proposed the shortest path quad tree to avoid repeatedly calculating shortest paths between two vertices for different query positions.

2.3 Multi-Objective Queries in Spatial Databases

Generally speaking, our DBS query belongs to multi-objective queries which retrieve objects considering multiple attributes. In database area, multi-objective queries are also called *skyline queries* [23]. It has received considerable attentions since it was considered in relational databases for the first time in 2001 [2]. Thereafter, many subsequent algorithms were proposed to improve the performance from different aspects. The well-known centralized algorithms include branch-and-bound skyline algorithm (BBS) [10], sort-filter-skyline (SFS) [4], and linear elimination sort for skyline (LESS) [13], etc. Recently, distributed and parallel skyline processing algorithms (e.g., [24], [25]) have received growing interest with improvements of mobile processing capabilities and developments of wireless networks. Many vari-

ations and extensions are derived from the classical skyline [2] with respect to different research focuses, such as location-based skyline queries in spatial databases.

In spatial databases, the notion of a skyline query provides a new perspective for realizing a location-based service which considers multiple factors including spatial and/or non-spatial attributes. Spatial preferences or attributes (e.g., distance) are different from other static attributes (e.g., price) because they depend on the query point (e.g., location of the mobile user) which moves continuously in most location-based applications.

Approaches [8,18] focus on a dynamic spatial attribute (Euclidean distance) and static non-spatial attributes. For example, [19,21] focus on both the distance information and some static non-spatial attributes, and [22] focuses on two static spatial attributes, i.e., spatial network distance and detour. Huang et al. [8] present skyline points for a continuously moving user (query point) considering distance and static non-spatial attributes. Among the objects given, some are permanent skyline points no matter where the user is as they are dominating static non-spatial values; while some are volatile skyline points because their dominance properties depend on the distances to the moving query point. However, the observation is that their dominance properties do not change abruptly while a user moves with a constant speed. A change of the dominance property happens when the distances of two data points share the same distance to the query point. In this work, the authors proposed a method to predict when two objects actually share the same distance to the query point and perform updates at these moments only.

Zheng et al. [18] proposed approaches to present skyline points for a dynamic query point without assuming that the query point moves with a constant speed and the spatio-temporal coherence exists as in [8]. Observing distributions of data points, they derive a valid scope wherein all query points will receive an identical skyline. The skyline is updated if a new query point falls outside of the valid scope of the original query point. However, those algorithms [8,18] that consider distances only cannot be directly applied to answer DBS queries because we consider not only distances but also directions.

The skyline problem becomes more complicated when we take multiple dynamic attributes into account [19, 21]. Chen et al. [19] predict a new skyline at a moment after the start moment for a moving query point considering a dynamic distance, non-spatial dynamic attributes (time-parameterized), and static attributes. Data points are indexed by an extended TPS-tree which integrates non-spatial dynamic attributes and static attributes as well as dynamic distances indexed by traditional TPS-tree [20]. Skyline points are found out by using a time-parameterized BBS algorithm on the extended TPS-tree. We can issue a new query at each moment by using the predictive skyline query processing algorithms in [19] to update skylines while a query point moves.

Lee et al. [21] proposed alternative algorithms to update skylines continuously for a moving query point with dynamic distances and dynamic non-spatial attributes. They assume the query point moves with a constant speed and dynamic non-spatial attributes values also change linearly. By following the filter-and-refinement principle, they first select candidates which could possibly qualify as skyline points and then trace changes of skylines by only evaluating those candidates. During the candidates selection phase, they derive the candidate region while filtering out the permanently dominated regions and further reduce the candidate set according to some pruning rules. The method efficiently answers continuous skyline queries for moving objects with dynamic attributes. However, it is impossible to directly use that algorithm to solve $\mathbb{E}$-DBS queries as the dynamic direction attribute is different from other dynamic attributes which usually can be described as time-parameterized linear or quadratic functions.

Huang et al. [22] proposed another interesting spatial skyline query problem in road network scenarios. In their work, data points represent intermediate locations (e.g., gas station) that a user wants to visit temporarily on his way to a given destination. Skyline points are found to balance distances of intermediate locations and detours arose by visiting intermediate locations.

2.4 Borrowed Ideas for Implementing Our DBS Algorithms

In order to facilitate continuous $\mathbb{E}$-DBS query algorithms, we borrow some ideas from continuous nearest neighbor (CNN) queries proposed by Tao et al. in [15] and we also employ the basic idea of the work presented by Raptopoulou et al. in [12] to make use of the intersections of distance functions to find changes of nearest neighbors.

In order to facilitate snapshot $\mathbb{R}$-DBS query algorithms, we use the basic idea of the well-known Dijkstra algorithms to find out shortest paths [28–30].

3 Preliminaries

In this section, we formalize DBS queries. There is a set of target objects $P = \{p_1, \ldots, p_n\}$ and a query object q in a space $\mathbb{X}$. DBS queries recommend nearest objects

around q considering their distances d_i and directions ω_i w.r.t. q. Comparing two objects p_i and p_j, p_i represents a better candidate than p_j if $d_i < d_j$ and they are directional close. In the paper, we consider DBS queries in both a vector space $\mathbb{E}$ and a metric space $\mathbb{R}$ ($\mathbb{X} \in \{\mathbb{E}, \mathbb{R}\}$). In the vector space $\mathbb{E}$, we compare objects using Euclidean distances and directions. While in the metric space $\mathbb{R}$, we compare objects using network distances and directions.

Table 1: Symbols and descriptions

Symbol	Description
$\overrightarrow{p_i}$	Vector from q to p_i in $\mathbb{E}$
$SP(q, p_i)$	Shortest path from q to p_i in $\mathbb{R}$
d_i (d_{p_i})	Distance of p_i: it is set to the length of $\overrightarrow{p_i}$ in $\mathbb{E}$ ($d_{p_i} = \lvert\overrightarrow{p_i}\rvert$) or the length of $SP(q, p_i)$ in $\mathbb{R}$ ($d_{p_i} = \lvert SP(q, p_i)\rvert$)
ω_i (ω_{p_i})	Direction of p_i: it is set to the angle between $\overrightarrow{p_i}$ and (1, 0) in $\mathbb{E}$ or the shortest path $SP(q, p_i)$ in $\mathbb{R}$
θ	Threshold for an acceptable angle
λ_{ij} ($\lambda_{p_ip_j}$)	Included angle between $\overrightarrow{p_i}$ and $\overrightarrow{p_j}$
φ_{ij} ($\varphi_{p_ip_j}$)	Partition angle between p_i and p_j

Before defining the *DBS* problem formally, we would like to define the *dominance relationship* first.

Definition 1 (Dominance Relationship) In a metric space $\mathbb{X} \in \{\mathbb{E}, \mathbb{R}\}$, if two objects p_i and p_j are directional close and p_i is closer to q than p_j (i.e., $d_i < d_j$), we say that p_i *dominates* p_j, denoted as $p_i \prec p_j$. □

Note that objects that are not directional close are not comparable. We will define the *direction closeness* in the vector space $\mathbb{E}$ (Section 3.1) and in the metric space $\mathbb{R}$ (Section 3.2) later. Accordingly, *DBS queries* are defined in Definition 2.

Definition 2 (DBS Query) Given a set of POI objects $P = \{p_1, \ldots, p_n\}$ and a query point q in a space $\mathbb{X} \in \{\mathbb{E}, \mathbb{R}\}$, the objects that are not dominated by any other object are *direction-based surrounder points* (*DBS* points). A *direction-based surrounder* (*DBS*) query, denoted as $DBS(q, \theta)$ in $\mathbb{E}$ and $DBS(q, G)$ in $\mathbb{R}$, is to find all the direction-based surrounder points, i.e., $\{p_i \mid p_i \in P, \nexists p_j (\neq p_i) \in P, p_j \prec p_i\}$. □

3.1 Directional Closeness for $\mathbb{E}$-DBS Queries

Assume that a set of target objects $P = \{p_1, \ldots, p_n\}$ and a query object q are in a two-dimensional Euclidean space $\mathbb{E}$. The vector $\overrightarrow{p_i}$ from q to p_i is used to capture both the *distance* and *direction* of p_i w.r.t. q. To be more specific, the distance of p_i to q is represented by d_{p_i} ($= \lvert\overrightarrow{p_i}\rvert$) which is the Euclidean distance between p_i and q and the direction of p_i w.r.t. q is represented by ω_{p_i} ($\in [0, 2\pi)$) that is the angle between vector $\overrightarrow{p_i}$ and the unit vector $(1, 0)'$. We use the abbreviations d_i and ω_i to refer to p_i's *distance* and *direction* if the context is clear. For example, the vector $\overrightarrow{a}$ in Fig. 2 has the distance $d_a = \lvert\overrightarrow{a}\rvert = \sqrt{5}$ and the direction $\omega_a = 27°$.

Now we explain how $\mathbb{E}$-DBS actually evaluates objects by considering both the distance and direction. Intuitively, two objects p_i and p_j are in the *same direction* if their directions w.r.t. q happen to be the same (i.e., $\omega_i = \omega_j$). This definition, however, is too strict in practice. Alternatively, we consider that two objects are *almost* in the same direction if their directions are almost equivalent ($\omega_i \approx \omega_j$). Towards this, we introduce a threshold θ ($\in [0, \frac{\pi}{2})$), namely an *acceptable angle*, which can be specified by the user in the query time to evaluate the direction closeness. Given two objects p_i and p_j, their *included angle* formed by vectors $\overrightarrow{p_i}$ and $\overrightarrow{p_j}$ can capture their angular difference, mathematically defined as follows:

$$\lambda_{ij} = \arccos \frac{\overrightarrow{p_i} \cdot \overrightarrow{p_j}}{\lvert\overrightarrow{p_i}\rvert \cdot \lvert\overrightarrow{p_j}\rvert}. \quad (1)$$

Objects p_i and p_j are *directional close* w.r.t. a query point q and a threshold θ iff their included angle λ_{ij} is bounded by θ. Its formal definition is given in Definition 3.

Definition 3 (Directional Close in $\mathbb{E}$) For the given target objects p_i and p_j, we say that p_i and p_j are *directional close* w.r.t. q and a given threshold θ iff the condition $0 \leq \lambda_{ij} \leq \theta$ holds. □

As we have shown in Example 1 (Fig. 2), given $\theta = \pi/3$ and q, object b is directional close to a since the included angle λ_{ab} between their vectors is smaller than θ. On the other hand, object d is not directional close to a due to the fact that $\lambda_{ad} > \theta$.

3.2 Directional Closeness for $\mathbb{R}$-DBS Queries

Formally, a road network $G = (V, E)$ consists of a set of vertices $v_i \in V$, and a set of edges $e \in E$ with each $e(v_i, v_j)$ connecting nodes v_i and v_j. We assume the query issuing position q and a set of POI objects $P = \{p_1, \ldots, p_n\}$ are all located at some edges, i.e., $\forall p \in q \cup P$, $\exists e \in E \wedge p \in e$. Let $SP(q, p_i) = \{sp_1, \cdots sp_j\}$ be the shortest path from user q to a POI object p_i with $\{sp_1, \cdots sp_j\}$ representing the ordered set of vertices in V that $SP(q, p_i)$ passes sequentially, i.e., $SP(q, p_i)$ starts from q, then visits vertices sp_1, sp_2, $\cdots$, sp_j, and

finally reaches the destination p_i. In Fig. 3, $SP(q, a) = \{v_6, v_3\}$ means that the shortest path from q to a passes vertex v_6 and v_3 to reach a.

As explained previously, an $\mathbb{R}$-DBS query is based on both distance and direction, we strategically reformat the shortest path $SP(q, p_i)$ as a two-tuple vector (d_i, ω_i). Here, d_i[1] refers to the distance from q to p_i (i.e., the length of the shortest path from q to p_i), and ω_i denotes the direction of p_i which is represented by the set of nodes passed by $SP(q, p_i)$ (i.e., sp_1, sp_2, $\cdots$, sp_j). Therefore, $SP(q, a) = (9, v_6v_3)$, and $SP(q, b) = (10, v_6v_3)$ in Fig. 3.

Two objects are *directional close* on the road network $\mathbb{R}$ iff their shortest paths from q overlap. Specifically, one object p_i must be on the shortest path of another object p_j in order to be directional close to p_i, as defined in Definition 4. Based on this definition, in Fig. 3, objects a and b are directional close as $SP(q, a).\omega_a = SP(q, b).\omega_b$, and a is located on the shortest path $SP(q, b)$.

Definition 4 (Directional Close in $\mathbb{R}$) On the road network $G = \{V, E\}$, two target objects p_i and p_j are *directional close* w.r.t. q iff $SP(q, p_i).\omega_i \subseteq SP(q, p_j).\omega_j$ or $SP(q, p_j).\omega_j \subseteq SP(q, p_i).\omega_i$. □

3.3 Two Minor Issues for $\mathbb{E}$-DBS Queries

Before we present the detailed search algorithms for DBS queries, we would like to mention two minor issues and their solutions for $\mathbb{E}$-DBS queries.

1. In the following discussions, we consider the case of $0 < \theta < \pi/2$ but omit the case of $\theta = 0$. This is because as $\theta = 0$, the majority of target objects are not dominated as an object p_i can only be dominated by objects p_j lying along the radial line from q to p_i (i.e., $\omega_i = \omega_j$). In the case that all the objects have different directions w.r.t. q, all the objects are $\mathbb{E}$-DBS objects. This contradicts the main objective of our searches which is to find a small set of dominative objects out of a large object set to ease objects selection/analysis process.
2. We assume a query will not be issued at a target object p_i. In other words, the query can only be issued from a position that is different from any the target objects, i.e., $\forall q, \nexists p_i \in P, d_{p_i} = 0$. The reason behind is that when $d_{p_i} = 0$, p_i actually dominates all the other target objects in all the directions, and becomes the only $\mathbb{E}$-DBS object.

[1] In this paper, both $|SP(q, p_i)|$ and $SP(q, p_i).d_i$ refer to the shortest distance from q to p_i on a road network.

4 Snapshot DBS Queries

In this section, we present snapshot $\mathbb{E}$-DBS queries and $\mathbb{R}$-DBS queries and their corresponding search algorithms. When a direction-based surrounder query is issued at a fixed query point q, it is a snapshot DBS query. As mentioned in Section 1, a snapshot DBS query finds the "best view" objects based on the user's current position.

A naïve solution to support snapshot queries is to compare every object with all the other objects. If the object is not dominated by any other object, it is a DBS point. This approach is developed directly based on the definition of a snapshot DBS query, and has $O(n^2)$ time complexity with $n = |P|$. Although we can improve the performance by saving the comparison of an object p_j against others once it is detected to be dominated by some object p_i, it is still inefficient. In the following, we present some efficient search algorithms to support $\mathbb{E}$-DBS queries and $\mathbb{R}$-DBS queries, respectively.

4.1 Snapshot $\mathbb{E}$-DBS Queries

At first, we introduce two observations based on which an efficient search algorithm is developed to answer snapshot DBS queries. Notice that our search algorithm examines the target objects of P based on ascending distance order w.r.t. q, i.e., nearby objects are evaluated earlier[2].

Observation 1: Search space pruning

Given an object $p_j \in P$, it can only be dominated by another object p_i that is directional close to p_j. Consequently, there is no need to evaluate objects that are not directional close to p_j as they for sure will not dominate p_j. In other words, the search space for a dominative object p_i can be pruned based on directional closeness. In our algorithm, objects are evaluated based on ascending order of their distances to q. p_j can only be dominated by those objects visited earlier and meanwhile are directional close to p_j. In order to facilitate the checking of directional closeness, we introduce the notions of a *direction order list* and *adjacent objects* defined in Definition 5 and Definition 6, respectively.

Definition 5 (Direction Order List) Given a set of objects P' and a query point q, its *direction order list* $L_{P'} = \langle p_1, p_2, \cdots p_k \rangle$ is formed by the objects of P'

[2] To simplify the presentation, we assume all the objects have different distances to query point q. However, our algorithm can be easily adjusted to cater for the cases where multiple objects have the same distances to the query point.

based on ascending order of their directions w.r.t. q, i.e., i) $\forall p_i, p_{i+1} \in L_{P'}$, $\omega_{p_i} \leq \omega_{p_{i+1}}$; ii) $\forall p_i \in L_{P'}$, $p_i \in P'$; and iii) $|L_{P'}| = |P'|$. □

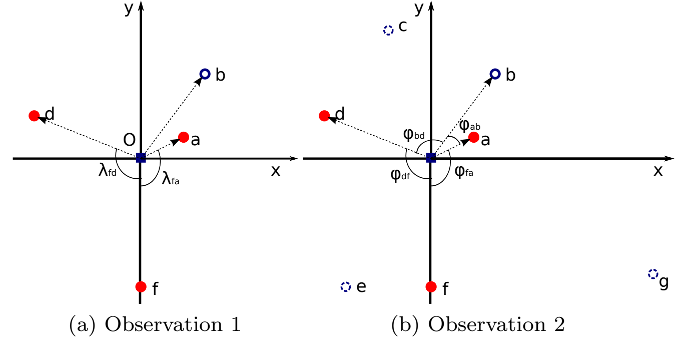
Definition 6 (Adjacent Object) Given a set of objects $P' = \{p_1, p_2, \cdots p_k\}$ and a query point q, objects p_i and p_j are *adjacent* to each other iff they are next to each other in the corresponding direction order list $L_{P'}$, i.e., there is no other object p_m in $L_{P'}$ such that $\omega_{p_i} < \omega_{p_m} < \omega_{p_j}$. Notice that the head entry of $L_{P'}$ is adjacent to tail entry of $L_{P'}$ due to the circular aspect of this notion. □

For each object p_i in P', it has two adjacent objects, i.e., the *predecessor* located ahead of p_i in $L_{P'}$, denoted as p_i^- and the *successor* located right after p_i, denoted as p_i^+. Due to the circular aspect of adjacency, the head entry of $L_{P'}$ has the tail entry of $L_{P'}$ as its predecessor and similarly, p_k, the tail entry of $L_{P'}$ has the head entry of $L_{P'}$ as its successor. Please refer to Example 5 for the detailed explanation.

Example 5 Let us consider the example in Fig. 2 again. Assume objects are evaluated based on ascending order of their distances to q, and those objects that have been evaluated form the set $P' = \{a, b, d\}$ with $L_{P'} = \langle a, b, d \rangle$, as shown in Fig. 8(a). Object a is adjacent to b and d. Object b has object a as its predecessor and has d as its successor, i.e., $b^- = a$ and $b^+ = d$. Suppose we are now evaluating object f. As f for sure will not be dominated by any object that has not been evaluated due to shorter distance to q, we only need to consider objects in P'. As objects in P' are closer to q than f, they for sure satisfy the distance requirement specified in the dominance relationship of DBS queries. Therefore, we only need to consider the directional closeness. Obviously, the object in P' that has the smallest included angle with f must be either d or a, i.e., the predecessor and successor of f if we consider the set of objects $P' \cup \{f\}$. Therefore, by comparing λ_{af} and λ_{df} with θ, we can decide whether f is dominated. ■

Based on this observation, we develop the following property to prune away objects that do not need evaluation when we evaluate an object p_i.

Property 1 Let object p_i be the target object currently evaluated, and suppose set P' maintains all the objects evaluated (i.e., those objects being closer to q compared with p_i). Assume p_j and p_k are the predecessor and successor of p_i. Object p_i is a DBS object iff both included angles λ_{ij} and λ_{ik} are larger than θ. □



(a) Observation 1 (b) Observation 2

Fig. 8: Two observations

Observation 2: Early termination

The second observation is that the directional closeness enables us to terminate the object evaluation process earlier without examining all the objects in P. We use Example 6 to explain the basic idea. Notice that a new concept *partition angle* is used in Example 6 and its formal definition is presented in Definition 7.

Definition 7 (Partition Angle) Given a set of objects P', let $p_j \in P'$ be the successor object of $p_i \in P'$. The *partition angle* φ_{ij} is defined based on p_i and its successor p_j, as expressed in Eq. (2):

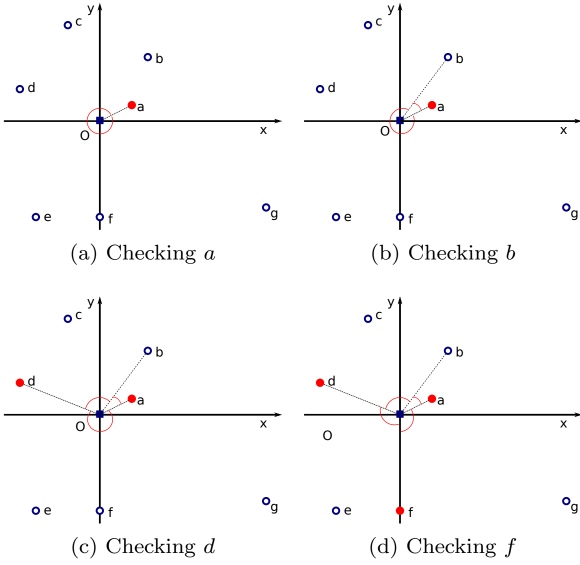
$$\varphi_{ij} = (\omega_j - \omega_i) \mod 360°. \quad (2)$$

□

Example 6 Let us continue our running example. Assume that we have checked the objects a, b, d, and f already, i.e., $P' = \{a, b, d, f\}$. As shown in Fig. 8(b), four partition angles are formed. They are $\varphi_{ab} = 26°$, $\varphi_{bd} = 105°$, $\varphi_{df} = 112°$, and $\varphi_{fa} = 117°$. In other words, the 2π angular range is partitioned by the objects of P' into four sub-angular regions. For a new object, it for sure will be located into one sub-angular region, and have objects $\{a, b\}$ (or $\{b, d\}$, or $\{d, f\}$, or $\{f, a\}$) as its adjacent objects. Given the fact that all the partition angles are smaller than or equal to $2\theta = 120°$, the new object will be definitely dominated by at least one of its adjacent objects, as stated in Property 2. ■

Property 2 If all the partition angles formed by the checked objects are not larger than 2θ, the remaining objects that have not been evaluated are dominated and the evaluation can be terminated safely. □

The algorithm to answer snapshot DBS queries is developed based on the above two observations. Before we explain the details of the search algorithm, we first use Example 7 to illustrate the basic idea.



(a) Checking a (b) Checking b

(c) Checking d (d) Checking f

Fig. 9: Processing snapshot DBS query ($\theta = \pi/3$)

Example 7 Given $P = \{a, b, c, d, e, f, g\}$, a snapshot DBS query is issued at point O with $\theta = \frac{\pi}{3}$. Objects are evaluated based on ascending order of their distances to O. Consequently, a is evaluated first, and it can be output as a DBS object immediately, as depicted in Fig. 9(a). After the evaluation of a, $P' = \{a\}$, and $\varphi_{aa} = 0$[3]. Next, we check the second-nearest object b, as illustrated in Fig. 9(b). As $\lambda_{ab} < \theta$, it is dominated by a. Since there is one partition angle $\varphi_{ba} > 2\theta$, the early termination condition is not satisfied and the procedure continues. Then, we examine object d, the third nearest object to O, as depicted in Fig. 9(c). It can be returned as a DBS object as its two adjacent objects (i.e., a and b) are not directional close to d, i.e., $\lambda_{db} > \theta$ and $\lambda_{da} > \theta$. After the evaluation, $P' = \{a, b, d\}$, and partition angles φ_{ab} ($\leq 2\theta$), φ_{bd} ($\leq 2\theta$), and φ_{da} ($> 2\theta$) are formed. The procedure continues and we examine object f, as shown in Fig. 9(d). It is also a DBS object as it is not dominated by its adjacent objects, and meanwhile the procedure terminates since all the partition angles (i.e., φ_{ab}, φ_{bd}, φ_{df}, and φ_{fa}) are smaller than 2θ. All the DBS objects (i.e., a, d, f) are found. ■

Algorithm 1 lists the pseudo-code of the snapshot DBS query processing algorithm. First, we invoke an existing NN search algorithm to retrieve the nearest neighbor object using a spatial index (lines 3 -4). This object is for sure a DBS object as it is closer to q than any other object. After initializing variables, we check the target objects according to the increasing distance order (lines 8-15). In the algorithm, we maintain all the object that have been checked in P', sorted by their directions w.r.t. q. Whenever a new object p is evaluated, it is first inserted into P', and its predecessor and successor are retrieved, denoted as p^- and p^+ respectively (line 10). Based on Property 1, we compare included angles of p and its adjacent objects to decide whether p is dominated (lines 11-12). Thereafter, we update the partition angle set (line 14). The process repeats until early termination condition is satisfied (based on Property 2) or all the objects are evaluated (line 15).

[3] Notice that we use Property 2 as a termination condition when we have checked more than one object.

Algorithm 1 Snapshot DBS Query

1: **procedure** SnapshotDBSQuery(q, θ)
2: $S \leftarrow \emptyset$;
3: InitNNQuery(q); ▷ Initialize the NN query
4: $p \leftarrow$ GetNext(); ▷ Get the first NN object
5: $S \leftarrow \{p\}$; ▷ result set
6: $P' \leftarrow [p]$; ▷ Initialize the evaluated object set
7: $\Phi \leftarrow \{\varphi_{pp}\}$; ▷ Initialize the partition angle set
8: **repeat**
9: $p \leftarrow$ GetNext(); ▷ Get the next NN object
10: $\langle p^-, p^+ \rangle \leftarrow P'$.insert($p$);
▷ Insert p to P' and get its adjacent objects
11: **if** $\lambda_{pp^-} \geq \theta \wedge \lambda_{pp^+} \geq \theta$ **then**
12: $S \leftarrow S \cup \{p\}$; ▷ p is on the DBS
13: **end if**
14: $\Phi \leftarrow (\Phi - \{\varphi_{p^-p^+}\}) \cup \{\varphi_{p^-p}, \varphi_{pp^+}\}$;
▷ Update the partition angle set
15: **until** $\forall \varphi \in \Phi$, $\varphi \leq 2\theta$ or all the objects are processed
16: **output** S;
17: **end procedure**

Now we analyze the time complexity of Algorithm 1. The cost of the algorithm comes from the loop (lines 8-15), if we ignore the costs of function InitNNQuery and function GetNext. In the best case, the loop will terminate after checking $\lceil 2\pi/2\theta \rceil$ nearest neighbors. In the worst case, the loop will terminate after checking all objects (e.g., all of them are on some ray originating from the user) which is rather uncommon. In general, the time complexity of the loop is $O(1)$. The experimental results of snapshot queries shown in Section 6.1.1 also demonstrates the efficiency of Algorithm 1 in supporting snapshot DBS queries.

4.2 Snapshot $\mathbb{R}$-DBS Queries

As the dominance relationship in $\mathbb{R}$ relies on distance metric and directional closeness that are different from those in $\mathbb{E}$, a new search algorithm is needed. In the following, we first define an important property to facilitate the process, and then explain the search algorithm.

Property 3 Given a DBS query issued at point q on a road network $G(V, E)$, object p_i dominates another object p_j iff p_i is on the shortest path from q to p_j, i.e., if $p_i \in SP(q, p_j)$, $p_i \prec p_j$. □

Property 3 inspires a simple approach to answering a snapshot DBS query on a road network. Assume the shortest paths from the query point to each of the objects $p_i \in P$ is available. Take Fig. 3 as an example. Suppose all the shortest paths from q to $p \in P = \{a, b, c, d\}$ are known, i.e., $SP(q, a) = (9, v_6v_3)$, $SP(q, b) = (10, v_6v_3)$, $SP(q, c) = (5, v_4)$, $SP(q, d) = (9, v_4v_5)$. As a locates on $SP(q, b)$ and is closer to q than b, a is a DBS point. On the other hand, c and d are the only points located on their shortest paths and they are also DBS points. Consequently, $DBS(q, G) = \{a, c, d\}$. Given the fact that shortest path searches have been well studied in the literature [28–30], we assume the shortest paths from q to each object $p \in P$ is identified by some existing search algorithm (e.g., the Dijkastra algorithm used in our experiments), and the DBS objects could be found based on Property 3 to form the answer set.

5 Continuous DBS Queries

In this section, we extend the original DBS queries to a dynamic scenario. In addition to considering snapshot DBS queries issued at fixed query points, we consider the case where users keep moving when issuing DBS queries. Accordingly, we form *continuous DBS queries* to represent the processing of DBS queries when the query point keeps moving.

As pointed out in Section 1, a continuous DBS query presents up-to-date DBS objects while the user keeps moving. Naturally, we can issue a new snapshot DBS query whenever the user changes her position. In other words, a continuous DBS query can be converted to snapshot DBS queries. However, this simple approach is not preferred as a large number of snapshot DBS queries will be generated and many of them share the same results. Alternatively, we propose a *prediction-based* approach, i.e., predicting when and how the DBS objects change in the near future.

In a dynamic moving environment, the user's position keeps changing with different movement patterns. Our goal is to develop flexible algorithms which can support continuous DBS queries issued by mobile users with various movement patterns. As the first step, this work focuses on the prediction of the future locations of mobile users moving in a constant speed. To be more specific, let $t = 0$ be the *current time*, and the location of the user at a future time t (≥ 0), denoted as $\overrightarrow{q} = (x_q, y_q)'$, is mathematically expressed in Eq. (3).

$$\overrightarrow{q} = \begin{pmatrix} x_q \\ y_q \end{pmatrix} = \begin{pmatrix} x_v \\ y_v \end{pmatrix} t + \begin{pmatrix} \bar{x}_q \\ \bar{y}_q \end{pmatrix}, \tag{3}$$

where the user moves from $(\bar{x}_q, \bar{y}_q)'$ with a constant velocity $(x_v, y_v)'$.

Accordingly, a continuous DBS query is formally defined in Definition 8. Note that the following definition can be easily extended to an *interval-based DBS query* that is based on a given time interval $[t_s, t_e]$, instead of the time duration $[0, \tau]$. For $\mathbb{R}$-DBS queries, we assume the user moves on the road network within the time duration $[0, \tau]$. For $\mathbb{E}$-DBS queries, users can move freely.

Definition 8 (Continuous DBS Query) In a space $\mathbb{X} \in \{\mathbb{E}, \mathbb{R}\}$, a *continuous DBS query* with parameter τ ($\tau > 0$), which is issued by a user at position $(\bar{x}_q, \bar{y}_q)'$ moving in constant velocity $(x_v, y_v)'$, locates all the DBS points corresponding to the user locations during the time interval $[0, \tau]$. □

5.1 Processing Continuous $\mathbb{E}$-DBS Queries

In order to illustrate the concept of continuous $\mathbb{E}$-DBS queries, we extend our running example, as shown in Example 8.

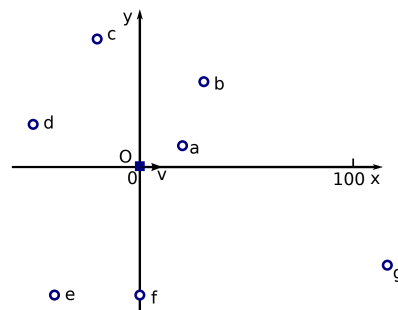


Fig. 10: Example of a continuous $\mathbb{E}$-DBS query

Example 8 Let us extend our example of snapshot $\mathbb{E}$-DBS queries to the continuous case. Fig. 10 illustrates that the user is moving from position $(0, 0)'$ with a constant speed $(1, 0)'$. The user issues a continuous DBS query for the time interval $[0, 100]$ with $\theta = \pi/3$. It means that we need to predict the changes of DBS during $[0, 100]$. The result will be as follows:

$$DBS = \begin{cases} \{a, d, f\} & t \in [0, 4) \\ \{a, d, f, g\} & t \in [4, 23) \\ \{a, f, g\} & t \in [23, 59) \\ \{a, g\} & t \in [59, 100]. \end{cases} \tag{4}$$

The output indicates that initially, objects a, d and f are the DBS points. These three objects remain as the only DBS points until user reaches $(4, 0)'$ at $t = 4$. At that point, object g becomes new DBS object and hence the result set is changed to $\{a, d, f, g\}$. It remains the same until the user moves to $(23, 0)'$ at $t = 23$ at which the DBS object d is dominated by a and hence removed from the result set. Finally, DBS object f is also removed from the result set at $t = 59$ when the user reaches $(59, 0)'$.

Based on this example, we understand that although the user keeps changing her position from $t = 0$ to $t = 100$, the change of the DBS points happens only at $t = 4$, $t = 23$, and $t = 59$. We therefore name those moment as *change moments*, and a continuous DBS query can be easily converted to snapshot DBS queries issued from user's locations at those change moments. For example, if we can detect that $t = 4$, $t = 23$, and $t = 59$ are the only three change moments corresponding to our example continuous query, we can issue 4 snapshot DBS queries w.r.t. the user's positions at time $t = 0$, $t = 4$, $t = 23$, and $t = 59$. Consequently, our algorithm focuses on how to predict the change moments effectively. ■

5.1.1 Basic Idea

The solution to continuous DBS queries shares the same framework as snapshot DBS queries. In processing snapshot queries, we check the target objects one by one based on ascending order of their distances to the fixed query point. For each target object p_i evaluated, we compare its included angles formed with its adjacent objects against the angular threshold θ to evaluate if p_i is dominated, according to Property 1. The evaluation process can be safely terminated if all the partition angles formed by examined objects are bounded by 2θ or all the target objects have been evaluated.

Similarly, a continuous DBS query intends to evaluate those objects nearer to the query point earlier. However, the user, who issued a continuous query, keeps moving and hence the distance from a target object to the user's current location keeps changing. Like in Example 8, object a is nearest to q when $t = 0$ and object g becomes nearest when $t = 75$. How to determine the ordering of objects based on their distance to the query point (i.e., user's current location) in a dynamic scenario is critical. In our work, we propose a *process tree* to facilitate the ordering.

As illustrated in Fig. 11, a process tree is in tree structure, and the height of the tree is bounded by the number of target objects considered. Its root node n_{00} includes the time interval $[0, \tau]$ considered by the given continuous DBS query (e.g., (0, 100) in our example).

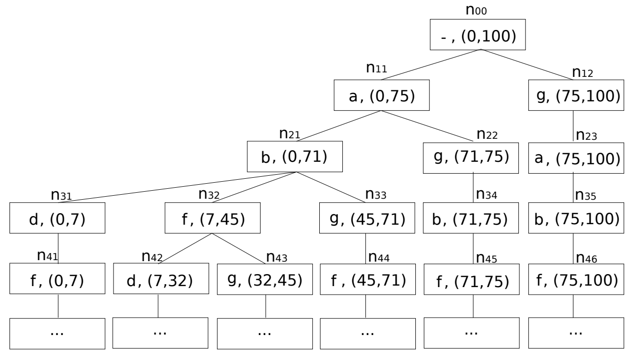


Fig. 11: Process tree for continuous DBS query

Let the root node be at level 0, its immediate child nodes (e.g., n_{11}) be at level 1, its immediate grandchild nodes (e.g., n_{21}) be at level 2, and so on. Each child node n_{ij} at level i actually corresponds to an ith nearest neighbor to q within certain time interval. Consequently, each non-root node n_{ij} is in the format of $\langle p_k, \mathcal{I}_{p_k} \rangle$ with p_k being the ith nearest neighbor to q within the time duration $\mathcal{I}_{p_k} = (t_1, t_2) \subseteq [0, \tau]$. For example, as shown in Fig. 11, node n_{22} at level 2 has its content $\langle g, (71, 75) \rangle$, meaning that object g is the second nearest neighbor to q during the interval $(71, 75)$, and node n_{33} at level 3 has its content $\langle g, (45, 71) \rangle$ which means object g is the third nearest neighbor to q during the interval $(45, 71)$.

Note that given a parent node $n_{ij} = \langle p_k, \mathcal{I}_{p_k} \rangle$ and a child node $n_{(i+1)j'} = \langle p_m, \mathcal{I}_{p_m} \rangle$, the interval associated with the child node $n_{(i+1)j'}$ is always bounded by that associated with the parent node n_{ij}, i.e., $\mathcal{I}_{p_m} \subseteq \mathcal{I}_{p_k}$. In order to fulfill this requirement, for an object p_i that is the jth nearest neighbor to q at duration $\mathcal{I}_i$, multiple nodes $\langle p_i, \mathcal{I}_{ik} \rangle$ might have to be generated at level j with each corresponding to a sub-interval $\mathcal{I}_{ik}$ of $\mathcal{I}_i$ $(\cup_k \mathcal{I}_{ik} = \mathcal{I}_i)$. For example, nodes n_{34} and n_{35} at level 3 both correspond to object b with n_{34} associated with interval $(71, 75)$ and n_{35} associated with interval $(75, 100)$, and nodes n_{44}, n_{45}, n_{46} at level 4 all correspond to object f, with n_{44} associated with time interval $(45, 71)$, n_{45} associated with time interval $(71, 75)$, and node n_{46} associated with time interval $(75, 100)$. We will explain the reason behind this design when we illustrate how to utilize process trees to conduct continuous DBS searches in the following. It is worth noting that the process tree is similar to a partially persistent data structure for an ordered list.

Given the process tree, we can conduct the continuous DBS search by evaluating the target objects one by one based on ascending order of their distances to q. Initially, the nodes at level 1 will be evaluated. As they are the nearest objects to q (corresponding to dif-

ferent time intervals), they are DBS objects. As shown in Fig. 11, object a is nearest to q (and hence a DBS object) during the time interval $(0, 75)$, and object g is nearest to q (and hence a DBS object) during the rest time interval (i.e., $(75, 100)$). Since there are two objects at level 1, the continuous DBS query is split into two sub-queries q_{11} and q_{12}, each of which corresponds to time intervals $(0, 75)$ and $(75, 100)$, respectively. The reason we split the query into sub-queries associated with disjointed time intervals is to facilitate the distance-based ordering of objects.

Consider to process sub-query q_{11}. We need to evaluate objects that are the second nearest to q during interval $(0, 75)$. Based on the process tree, we can understand that objects b and g are the second nearest objects during intervals $(0, 71)$ and $(71, 75)$, respectively. Hence, the sub-query q_{11} is further split into two sub-queries q_{21} and q_{22}, each of which is associated with time intervals $(0, 71)$ and $(71, 75)$, respectively. For q_{21}, it has its own set of examined objects $P'_{21} = \{a, b\}$, and q_{22} also has its own set of examined objects $P'_{22} = \{a, g\}$. Based on Property 1, we can decide whether b (or g) is dominated by comparing its included angle with its adjacent objects[4]. Thereafter, we can form the partition angles, as in a snapshot DBS query processing, and safely terminate the processing of the subquery if the early termination condition is satisfied. Otherwise, we need to find out the next nearest neighbor within the time interval associated with the current sub-query (e.g., $(0, 71)$ of q_{21}) by visiting the child nodes (e.g., n_{31}, n_{32}, n_{33}), and continue the above process.

Based on this example, we understand that actually each node of the process tree corresponds to a sub-query of the initial continuous DBS query. Take node n_{42} as an example. It corresponds to sub-query q_{42} with time interval $(7, 32)$. Within this interval, q has a, b, f, and d as the top-4 nearest objects, which are captured by node n_{42} and its ancestor nodes (i.e., n_{32}, n_{21}, and n_{11}).

Now we understand how the process tree can facilitate the ordering of objects based on their distances to q. The next issue we have to address is how to construct a process tree. The construction of process tree is an incremental process and the tree is generated level by level. We regard the user's moving trajectory as the query line segment, and invoke an existing continuous nearest neighbor (CNN) search algorithm to find all the nearest neighbors (or k nearest neighbors). Naturally, the retrieved nearest objects will form the nodes of level 1. After evaluating all those nearest objects, we can invoke the CNN search algorithm to find the second nearest neighbors to form the nodes of level 2. The process continues until the continuous DBS query processing terminates. As to be presented next, the expansion of the process tree is well integrated with the processing of continuous DBS queries.

Algorithm 2 presents the pseudo-code of the continuous DBS query processing algorithm. It invokes function FindDBS recursively following the expansion of the process tree explained above. Function FindDBS first invokes function CNNQuery to retrieve the k-th nearest neighbor objects $\langle p, \mathcal{I}_p \rangle$ for the moving query point $\overrightarrow{q}$ within the time interval $\mathcal{I}$ and expands the process tree accordingly[5] (line 8). For each $\langle p, \mathcal{I}_p \rangle$, it then invokes function FindAdjacentObj to locate the adjacent objects of p and invokes function DomCheck to decide whether p is dominated based on Property 1 (lines 9-12). Notice that $\langle p, \mathcal{I}_p \rangle$ may correspond to multiple nodes in the process tree, i.e., time interval $\mathcal{I}_p$ is split into several sub-intervals. This is caused by the design of the process tree that the time interval of a child node could not be larger than that of the parent node. Thereafter, the early termination condition is checked via function CannotTerminate. If it is not satisfied, the objects evaluation continues by examining the next nearest objects via function FindDBS (lines 14-15).

Algorithm 2 Continuous $\mathbb{E}$-DBS Query

```
1: procedure ContinuousEDBSQuery(q, θ, I)
                          ▷ I is the target time interval: I = [0, τ].
2:     r ← CreateRootNode();                    ▷ Create a root node.
3:     S ← ∅;
4:     FindDBS(q⃗, I, 1, r, S);
5:     output S;
6: end procedure

7: procedure FindDBS(q, I, k, n, S)
8:     foreach ⟨p, I_p⟩ ∈ CNNQuery(q⃗, I, k) do
                              ▷ Find k-th NN object(s) while I.
9:         A ← FindAdjacentObj(p, I_p);
                              ▷ Find p's adjacent objects while I_p.
10:        foreach ⟨p⁻, p⁺, I'⟩ ∈ A do
11:            DomCheck(p, ⟨p⁻, p⁺, I'⟩, S);
                                             ▷ Check dominance.
12:            S ← UpdateDBS(S);
13:        end for
14:        if CannotTerminate(p, I_p) then
                          ▷ Termination condition is not satisfied.
15:            FindDBS(q⃗, I_p, k + 1, S);
                                        ▷ Expand the child nodes.
16:        end if
17:    end for
18: end procedure
```

[4] How to locate the adjacent objects for a given object when q keeps changing will be explained next.

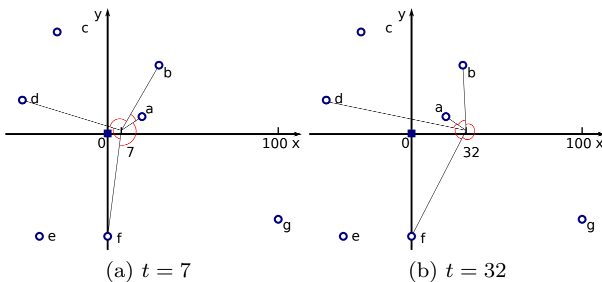
[5] In Appendix A.1, we explain how to implement the function CNNQuery by extending the existing incremental NN algorithm proposed by Tao et al. in [15].

In the following subsections, we will explain three major components of above algorithm. They are i) function FindAdjacentObj to find adjacent objects; ii) function DomCheck to conduct a dominance test; and iii) function CannotTerminate to evaluate the early termination condition in the dynamic scenario.

5.1.2 Finding Adjacent Objects

In processing a snapshot DBS query, we form a direction order list L for all the examined objects w.r.t. q. By simply inserting a new object p_i to L, those objects next to p_i in L are the adjacent objects. However, in a dynamic scenario, the position of q and the direction λ_{p_i} keep changing. We use Example 9 to demonstrate its complexity.

Example 9 Let us consider our example in Fig. 11 again. Assume we are evaluating sub-query q_{42} associated with time interval $(7, 32)$, and the object currently evaluated is d. Based on the process tree, we can know that objects a, b, and f (i.e., those associated with ancestor nodes of n_{42}) are closer to q than d within interval $(7, 32)$. When $t = 7$, the direction order list $L_{t=7} = \langle a, b, f\rangle$ and d's adjacent objects are b and f, as depicted in Fig. 12(a). When $t = 32$, the direction order list $L_{t=32} = \langle b, a, f\rangle$ and d's adjacent objects are a and f, as shown in Fig. 12(b). The change (i.e., a replaces b as the new adjacent object to d) happens at $t = 18$ when a and b are *co-linear* with q (i.e., $\lambda_a = \lambda_b$). In other words, b and f are adjacent to d during interval $(7, 18)$, and a and f are adjacent to d during interval $(18, 32)$. Accordingly, we maintain two direction order lists $\langle a, b, d, f\rangle_{(7,18)}$ and $\langle b, a, d, f\rangle_{(18,32)}$. ■



(a) $t = 7$ (b) $t = 32$

Fig. 12: Change of direction order

Based on the above observation, we develop Property 4 to guide the detection of the moment where objects in the direction order list switch their positions.

Property 4 If two objects are in the same side of the user's moving trajectory, their direction order changes when they are co-linear w.r.t. the query point. □

We can employ a sweeping line algorithm to find the point along the moving trajectory $\overrightarrow{q}$ where two objects change their positions in the direction order list L. To be more specific, let $\langle a, b\rangle$ be a pair of objects that lie in the same side of $\overrightarrow{q}$, and let $line(a, b)$ represent a line passing by both a and b. The intersection between $line(a, b)$ and $\overrightarrow{q}$ is the point when a and b change their positions in L. Thereafter, we can easily derive the time t when the user reaches the detected intersection point.

Example 9 (Continued) Fig. 13 illustrates the process of forming direction order lists for a sub-query q_{42}. At the beginning, we traverse the node $n_{11} = \langle a, (0, 75)\rangle$ of the process tree and initialize the direction order list to $\langle a\rangle$. Then, we reach its child node $n_{21} = \langle b, (0, 71)\rangle$, and update the direction order list to $\langle a, b\rangle$. However, the objects b and a are co-linear when $t = 18$. Thus, we split the interval into two sub-intervals $(0, 18)$ and $(18, 71)$, and maintain two direction order lists accordingly, with $L_{(0,18)} = \langle a, b\rangle$ and $L_{(18,71)} = \langle b, a\rangle$. Next we reach the node $n_{32} = \langle f, (7, 45)\rangle$ and insert f into both list $L_{(7,18)}$ and list $L_{(18,45)}$. Notice that the time interval corresponding to both lists shrink as f is the third nearest neighbor only within interval $(7, 45)$. As object f has no co-linear objects in both lists, no changes are detected. Finally, we reach node n_{42}, and we can update the direction order lists similarly. ■

Tree Node	Time Interval	Direction Order List	Operation
$n_{11}\langle a, (0, 75)\rangle$	$(0, 75)$	$\langle \boldsymbol{a}\rangle$	insert a
$n_{21}\langle b, (0, 71)\rangle$	$(0, 18)$	$\langle a, \boldsymbol{b}\rangle$	insert b
	$(18, 71)$	$\langle \boldsymbol{b}, a\rangle$	swap(a, b)
$n_{32}\langle f, (7, 45)\rangle$	$(7, 18)$	$\langle a, b, \boldsymbol{f}\rangle$	insert f
	$(18, 45)$	$\langle b, a, \boldsymbol{f}\rangle$	insert f
$n_{42}\langle d, (7, 32)\rangle$	$(7, 18)$	$\langle a, b, \boldsymbol{d}, f\rangle$	insert d
	$(18, 32)$	$\langle b, a, \boldsymbol{d}, f\rangle$	insert d

Fig. 13: Incremental maintenance of direction order lists

The detailed algorithm to detect the adjacent objects is shown in Algorithm 3. It takes a process tree node $n = \langle p, \mathcal{I}_p\rangle$ as input and returns the adjacent objects of p within $\mathcal{I}_p$. We assume that the direction order list corresponding to the parent node n is known and maintained by parameter D_{parent}. For each list $\langle l, \mathcal{I}\rangle \in D_{parent}$, we first insert p into the list (line 6), and then invoke function ColinearObjList to find out all the objects that are co-linear with p within the time interval $\mathcal{I}$, maintained in set C in the format of

$\langle p', t' \rangle$ where p' is p's co-linear object at time t' (line 7). To ease the update, we assume that objects in C are sorted based on ascending order of t'.

In the sequel, we evaluate each co-linear object of C and make necessary update to the direction order list l accordingly (lines 8-13). For a given co-linear object p' within t', we first find out p's predecessor p^- and the successor p^+ within the time interval (t_s, t'), and maintain them in $\mathcal{A}$ (line 9). Then, we switch the order of p and p' in the list l to form a new direction order list l', preserved in $\mathcal{D}$ (line 11). The process repeats until all the co-linear objects are evaluated. We deal with the last subinterval (t', I_e) at line 14 and line 15. We then attach $\mathcal{D}$, the set of direction order list of p within interval $\mathcal{I}_p$, to the tree node $\langle p, \mathcal{I}_p \rangle$ which will be used for function FindAdjacentObj. Finally, the algorithm terminates by returning $\mathcal{A}$.

Algorithm 3 Finding Adjacent Objects

1: **function** FindAdjacentObj(p, $\mathcal{I}_p$)
 ▷ p: object, $\mathcal{I}_p = \langle I_s, I_e \rangle$: time interval.
2: $\mathcal{A} \leftarrow \emptyset$; ▷ Set of p's adjacent objects while $\mathcal{I}_p$.
3: $\mathcal{D} \leftarrow \emptyset$; ▷ Set of p's direction order lists while $\mathcal{I}_p$.
4: **foreach** $\langle l, \mathcal{I} \rangle \in D_{parent}$ **do**
 ▷ D_{parent} is the direction lists of p's parent node.
5: $t_s \leftarrow I_s$;
6: $l \leftarrow l.$Insert(p); ▷ Insert p into l.
7: $C \leftarrow$ ColinearObjList($p, l, \mathcal{I}$);
 ▷ C carries p's co-linear objects and moments.
8: **foreach** $\langle p', t' \rangle \in C$ **do**
 ▷ p' is co-linear with p at t'.
 ▷ Process items in increasing order of t'.
9: $\mathcal{A} \leftarrow \mathcal{A} \cup \{\langle p^-, p^+, (t_s, t') \rangle\}$;
 ▷ Add p's adjacent objects while (t_s, t').
10: $l' \leftarrow l.$Swap(p, p');
 ▷ Create a new list by swapping p and p'.
11: $\mathcal{D} \leftarrow \mathcal{D} \cup \{\langle l', (t_s, t') \rangle\}$;
 ▷ Add the new list to $\mathcal{D}$.
12: $t_s \leftarrow t'$;
13: **end for**
14: $\mathcal{A} \leftarrow \mathcal{A} \cup \{\langle p^-, p^+, (t_s, I_e) \rangle\}$;
15: $\mathcal{D} \leftarrow \mathcal{D} \cup \{\langle l, (t_s, I_e) \rangle\}$;
16: **end for**
17: Attach $\mathcal{D}$ to the tree node $\langle p, \mathcal{I}_p \rangle$;
18: **return** $\mathcal{A}$;
19: **end function**

Let us consider the time complexity of Algorithm 3. Assume before processing the target tree node $\langle p, \mathcal{I}_p \rangle$, the number of checked objects in the branch is m. In the worst case, the outer loop (lines 4 to 16) will be executed $\binom{m}{2} + 1$ times, because m objects may have at most $\binom{m}{2}$ co-linear moments which separate the interval into $\binom{m}{2} + 1$ sub-intervals. Lines 6 and 7 have $O(m)$ cost respectively in the worst case. The inner loop executes m times in the worst case in which the target object becomes co-linear with all the checked objects. Therefore, the time complexity of Algorithm 3 is $O((\binom{m}{2}+1) \times 3m) = O(m^3)$ in the worst case. Although the time complexity in the worst case is high, in general there is no need to expand the processing tree too deep due to the early termination strategy in Section 5.1.4.

5.1.3 Checking Dominance

After confirming the adjacent objects of the target object during a certain time interval, the next step is to determine the dominance relationships between the target object and its adjacent objects, i.e., function DomCheck in Algorithm 2. Unlike in snapshot DBS query, the included angle between two objects changes while the user moves in the dynamic environment. Example 10 provides an example to demonstrate the dynamic nature of the included angle between two objects. Based on the observation made from the example, Property 5 is developed.

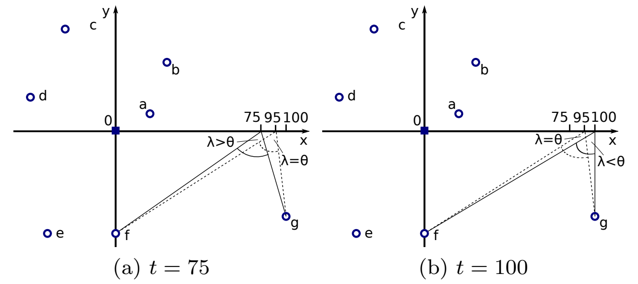


Fig. 14: Change of dominance relationship

Example 10 Take Fig. 11 as an example again. Assume that we are evaluating sub-query q_{46} associated with time interval $(75, 100)$, and the object evaluated currently is f. We have known that the adjacent objects of f is a and g during $(75, 100)$, returned by the function FindAdjacentObj, and we want to evaluate whether f is dominated by a and/or g. Take the evaluation of g as an example. As shown in Fig. 14(a), when $t = 75$, $\lambda_{fg} > \theta = \pi/3$. However, as shown in Fig. 14(b), when $t = 100$, $\lambda_{fg} < \theta$. The change happens at the moment $t = 95$ when $\lambda_{fg} = \theta$. In other words, f is not dominated by g during $(0, 95)$, but it is dominated by g during $(95, 100)$. The dominance relationship of two adjacent objects changes when their included angle equals to θ. ■

Property 5 Object p_i is not dominated by its adjacent object p_j during the time interval when their included angle $\lambda_{ij} \geq \theta$, i.e.,

$$\lambda_{ij} = \arccos \frac{\overrightarrow{p}_i \cdot \overrightarrow{p}_j}{|\overrightarrow{p}_i||\overrightarrow{p}_j|} \geq \theta \quad (0 \leq \lambda_{ij} \leq \pi). \tag{5}$$

□

Note that $\overrightarrow{p}_i$ and $\overrightarrow{p}_j$ are time-parameterized vectors that change with parameter t. To obtain the time intervals for which the formula holds, we need to find out the critical moments when $\lambda_{ij} = \theta$. Since λ_{ij} is a continuous function, these critical moments divide the time interval into two sub-intervals where $\lambda_{ij} > \theta$ in one sub-interval and $\lambda_{ij} < \theta$ in the other. In fact, the equation $\lambda_{ij} = \theta$ is a quartic equation with variable $t \in [t_s, t_e]$ and the solutions $t_i \in [t_s, t_e]$ $(i = 1, \ldots, 4)$ of the equation returned by GNU Scientific Library [7] (see Appendix A.2 for details) form the critical moments. We use the midpoints of every sub-intervals to determine whether $\lambda_{ij} \geq \theta$ or $\lambda_{ij} < \theta$, namely,

$$\lambda_{ab}(t)|_{t=[t_s^j,t_e^j]\in I_j} \begin{cases} \geq \theta, \text{ if } \cos\lambda_{ab}(\frac{t_s^j+t_e^j}{2}) \leq \cos\theta \\ < \theta, \text{ if } \cos\lambda_{ab}(\frac{t_s^j+t_e^j}{2}) > \cos\theta \end{cases}, \quad (6)$$

where $[t_s^j, t_e^j]$ represents a sub-interval $\mathcal{I}_j$.

For object p, we need to consider two adjacent objects p^- and p^+. We calculate the time intervals $\mathcal{I}^-$ and $\mathcal{I}^+$ when $\lambda_{pp^-} \geq \theta$ and $\lambda_{pp^+} \geq \theta$, respectively. Then we take their intersection to obtain the time interval while p is on the DBS.

The pseudo-code of DomCheck is depicted in Algorithm 4. It invokes the function UndomInterval to find out the un-dominated intervals $\mathcal{I}^-$ and $\mathcal{I}^+$ for object p where $\lambda_{pp^-} \geq \theta$ or $\lambda_{pp^+} \geq \theta$. Then, the intersection of two interval sets $\mathcal{I}^-$ and $\mathcal{I}^+$ is derived. The time complexity of Algorithm 4 is $O(1)$.

Algorithm 4 Checking Dominance

```
1: function DomCheck(p, ⟨p⁻, p⁺, I⟩)
2:     I⁻ ← UndomInterval(p, p⁻, I);
                   ▷ Set of intervals where p is not dominated by p⁻
3:     I⁺ ← UndomInterval(p, p⁺, I);
                   ▷ Set of intervals where p is not dominated by p⁺
4:     I_dbs ← I⁻ ∩ I⁺;
                   ▷ Set of intervals where p is on the DBS
5:     S ← S ∪ {⟨p, I_dbs⟩};
                   ▷ Add p and its intervals to DBS set S
6:     return S;
7: end function
```

5.1.4 Checking Termination Condition

Property 2 allows us to terminate the processing of the snapshot DBS query when all the partition angles are smaller than 2θ. In the dynamic scenario, a continuous DBS query is split into disjoint sub-queries q_i with each corresponding to *a certain time interval*. Similarly, we can safely terminate the processing of a sub-query q_i if all the partition angles are smaller than 2θ.

Consider Example 10 again. After evaluating subquery q_{46} corresponding to $(75, 100)$, if the four partition angles φ_{ba}, φ_{af}, φ_{fg} and φ_{gb} are all smaller than 2θ during $(75, 100)$, we can terminate the checking process on this branch.

A partition angle φ_{ab} formed by two objects $a = (x_a, y_a)'$ and $b = (x_b, y_b)'$ changes with the time parameter $t \in \mathcal{I}$. We want to decide whether φ_{ab} is always smaller than 2θ within the interval $\mathcal{I} = [I_s, I_e]$. To simplify the problem, we transform the coordinates by setting the user's start position $(\bar{x}_q, \bar{y}_q)'$ to the origin $(0, 0)'$, then we get $\overrightarrow{q} = (x_v, y_v)'t$. This does not change the essence of the problem. The vectors from q to a and b are given as follows:

$$\overrightarrow{a} = \begin{pmatrix} x_a \\ y_a \end{pmatrix} - \begin{pmatrix} x_v \\ y_v \end{pmatrix} t \quad (7)$$

$$\overrightarrow{b} = \begin{pmatrix} x_b \\ y_b \end{pmatrix} - \begin{pmatrix} x_v \\ y_v \end{pmatrix} t. \quad (8)$$

We analyze the variation of φ_{ab} by observing the properties of function $\cos\varphi_{ab}$. The detailed derivations are in Appendix A.3.

- *Case A*: It corresponds to the case when a and b are on the same side of the user's trajectory and $\overrightarrow{ab}$ is not parallel to $\overrightarrow{q}$. The condition is expressed as follows:

$$(\overrightarrow{ab} \times \overrightarrow{q} \neq 0) \wedge ((\overrightarrow{a} \times \overrightarrow{q})(\overrightarrow{b} \times \overrightarrow{q}) > 0). \quad (9)$$

  The notation $\overrightarrow{a} \times \overrightarrow{q}$ represents the *outer product* of $\overrightarrow{a}$ and $\overrightarrow{b}$[6]. The condition $(\overrightarrow{ab} \times \overrightarrow{q} \neq 0)$ represents that $\overrightarrow{ab}$ is not parallel to $\overrightarrow{q}$. The condition $((\overrightarrow{a} \times \overrightarrow{q})(\overrightarrow{b} \times \overrightarrow{q}) > 0)$ means that a and b are on the same side of the user's trajectory. In this case, $\cos\varphi_{ab}$ takes a local maximum 1 at

$$t = \frac{\overrightarrow{a} \times \overrightarrow{b}}{\overrightarrow{ba} \times \overrightarrow{q}}, \quad (10)$$

  and takes two local minima at

$$t = \frac{-|\overrightarrow{q}|(\overrightarrow{a} \times \overrightarrow{b}) \pm |\overrightarrow{ab}|\sqrt{(\overrightarrow{a} \times \overrightarrow{q})(\overrightarrow{b} \times \overrightarrow{q})}}{|\overrightarrow{q}|(\overrightarrow{ab} \times \overrightarrow{q})}. \quad (11)$$

- *Case B*: It is the case when a and b are on the opposite sides of the user's trajectory, namely,

$$(\overrightarrow{ab} \times \overrightarrow{q} \neq 0) \wedge ((\overrightarrow{a} \times \overrightarrow{q})(\overrightarrow{b} \times \overrightarrow{q}) < 0). \quad (12)$$

[6] The outer product of two vectors $\overrightarrow{v} = (v_x, v_y)'$ and $\overrightarrow{w} = (w_x, w_y)'$ in the two-dimensional case is defined as

$$\overrightarrow{v} \times \overrightarrow{w} = v_x w_y - v_y w_x = |\overrightarrow{v}||\overrightarrow{w}| \sin\eta,$$

where η is the angle between two vectors.

In this case, $\cos \varphi_{ab}$ takes a local minimum -1 at

$$t = \frac{\overrightarrow{a} \times \overrightarrow{b}}{\overrightarrow{ba} \times \overrightarrow{q}}. \tag{13}$$

– *Case C*: It corresponds to the case when the vector $\overrightarrow{ab}$ is parallel to $\overrightarrow{q}$, namely,

$$\overrightarrow{ab} \times \overrightarrow{q} = 0. \tag{14}$$

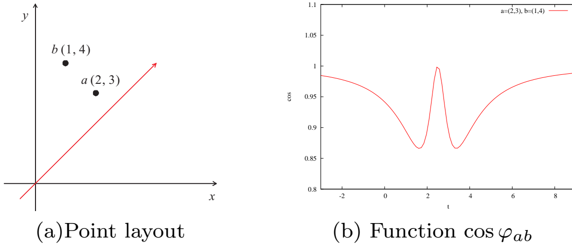
The function $\cos \varphi_{ab}$ takes a local minimum at

$$t = \frac{(|\overrightarrow{a}|^2 \overrightarrow{b} - |\overrightarrow{b}|^2 \overrightarrow{a}) \times \overrightarrow{q}}{2|\overrightarrow{q}|^2 (\overrightarrow{b} \times \overrightarrow{a})}. \tag{15}$$

Note that the situation that a (or b) is on the user's trajectory, where $(\overrightarrow{a} \times \overrightarrow{q})(\overrightarrow{b} \times \overrightarrow{q}) = 0$, does not happen in our problem setting.

Example 11 Let us consider examples of case A, B, and C. We assume that the query vector is given as $\overrightarrow{q} = (1, 1)'t$.

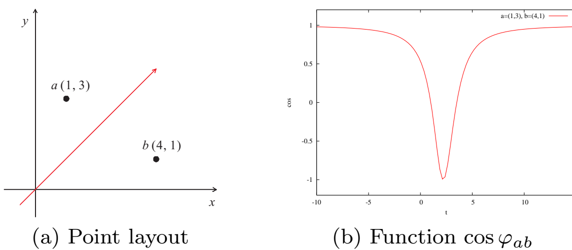
– *Case A*: In Fig. 15, $\cos \varphi_{ab}$ takes a local maximum at $t = 2.5$ and two local minima at $t = 1.63$ and $t = 3.67$.



(a)Point layout (b) Function $\cos \varphi_{ab}$

Fig. 15: Case A: $a = (2, 3)', b = (1, 4)'$

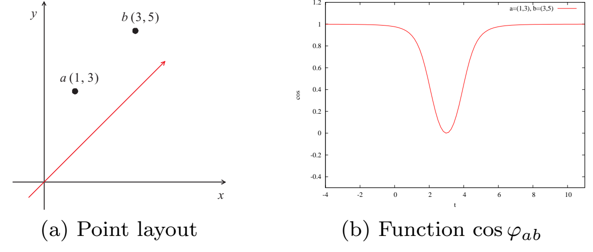
– *Case B*: In Fig. 16, $\cos \varphi_{ab}$ takes a local minimum $t = 2.2$.



(a) Point layout (b) Function $\cos \varphi_{ab}$

Fig. 16: Case B: $a = (1, 3)', b = (4, 1)'$

– *Case C*: In Fig. 17, $\cos \varphi_{ab}$ takes a local minimum $t = 3$.



(a) Point layout (b) Function $\cos \varphi_{ab}$

Fig. 17: Case C: $a = (1, 3)', b = (3, 5)'$

Then we need to check whether $\cos \varphi_{ab} > \cos 2\theta$ holds during the given time interval $\mathcal{I} = [I_s, I_e]$. In order to check this condition, we calculate the minimum value of $\cos \varphi_{ab}$ for the given interval $\mathcal{I}$. If the minimum value is greater than $\cos 2\theta$, the condition holds. Otherwise, the condition does not hold. We summarize the minimum values and their corresponding conditions in decision tables shown in Tables 2, 3, and 4.

Table 2: Decision table for case A

Condition	Minimum Value
$I_e \leq t_1$	$\cos \varphi_{ab}\|_{t=I_e}$
$(I_s \leq t_1) \wedge (t_1 < I_e \leq t_2)$	$\cos \varphi_{ab}\|_{t=t_1}$
$(I_s \leq t_1) \wedge (t_2 < I_e \leq t_3)$	$\min\{\cos \varphi_{ab}\|_{t=t_1}, \cos \varphi_{ab}\|_{t=I_e}\}$
$(I_s \leq t_1) \wedge (t_3 < I_e)$	$\min\{\cos \varphi_{ab}\|_{t=t_1}, \cos \varphi_{ab}\|_{t=t_3}\}$
$(t_1 < I_s < t_2) \wedge (I_e = t_2)$	$\cos \varphi_{ab}\|_{t=I_s}$
$(t_1 < I_s < t_2) \wedge (t_2 < I_e \leq t_3)$	$\min\{\cos \varphi_{ab}\|_{t=I_s}, \cos \varphi_{ab}\|_{t=I_e}\}$
$(t_1 < I_s < t_2) \wedge (t_3 < I_e)$	$\min\{\cos \varphi_{ab}\|_{t=I_s}, \cos \varphi_{ab}\|_{t=t_3}\}$
$(t_2 \leq I_s < t_3) \wedge (I_e \leq t_3)$	$\cos \varphi_{ab}\|_{t=I_e}$
$(t_2 \leq I_s \leq t_3) \wedge (I_e > t_3)$	$\cos \varphi_{ab}\|_{t=t_3}$
$t_3 < I_s$	$\cos \varphi_{ab}\|_{t=I_s}$

Note: t_1 and t_3 are two t-values when $\cos \varphi_{ab}$ takes two local minima of $\cos \varphi_{ab}$ (Eq. (11)), and t_2 is the t-value when $\cos \varphi_{ab}$ takes the local maximum of $\cos \varphi_{ab}$ (Eq. (10)).

Table 3: Decision table for case B

Condition	Minimum Value
$I_e \leq t^*$	$\cos \varphi_{ab}\|_{t=I_e}$
$(I_s < t^*) \wedge (t^* < I_e)$	-1
$t^* \leq I_s$	$\cos \varphi_{ab}\|_{t=I_s}$

Note: t^* is the t-value when $\cos \varphi_{ab}$ takes a local minimum (Eq. (13))

Note that, the constitution of the partition angles changes when the user moves. In Example 9, the subquery q_{42} generates two direction lists $L_{(7,18)} = \langle a, b, d, f \rangle$ and $L_{(18,32)} = \langle b, a, d, f \rangle$ as shown in Fig. 13. Their partition angle sets are $\Phi_{(7,18)} = \{\varphi_{ab}, \varphi_{bd}, \varphi_{df}, \varphi_{fa}\}$ and $\Phi_{(18,32)} = \{\varphi_{ba}, \varphi_{ad}, \varphi_{df}, \varphi_{fb}\}$, respectively. The

Table 4: Decision table for case C

Condition	Minimum Value
$I_e \leq t^*$	$\cos \varphi_{ab}\|_{t=I_e}$
$(I_s < t^*) \wedge (t^* < I_e)$	$\cos \varphi_{ab}\|_{t=t^*}$
$t^* \leq I_s$	$\cos \varphi_{ab}\|_{t=I_s}$

Note: t^* is the t-value when $\cos \varphi_{ab}$ takes a local minimum (Eq. (15)).

procedure can terminate when all angles in $\Phi_{(7,18)}$ and $\Phi_{(18,32)}$ are bounded by 2θ during $(7, 18)$ and $(18, 32)$, respectively. Therefore, we need to check partition angles for every list in order to determine whether we have found out all DBS objects.

For checking, we examine whether all φ's in every direction order list are bounded by 2θ within the time interval attached to the tree node. Assume that we are given a direction list $\Phi_{\mathcal{I}}$ (e.g., $\Phi_{(7,18)} = \{\varphi_{ab}, \varphi_{bd}, \varphi_{df}, \varphi_{fa}\}$). Obviously, if $|\Phi_{\mathcal{I}}| < 2\pi/2\theta$, the termination condition is not satisfied because the angles in the list cannot cover 2π angles. If $|\Phi_{\mathcal{I}}| \geq 2\pi/2\theta$, we need to check whether each partition angle φ in the list satisfies $\varphi \leq 2\theta$ while the time interval $\mathcal{I} = [I_s, I_e]$.

Algorithm 5 Checking Termination Condition

```
1:  procedure CANNOTTERMINATE(p, I_p)
2:      foreach ⟨l, I⟩ ∈ D do
                                  ▷ Each list in list set D of node ⟨p, I_p⟩.
3:          cnt ← 0;                          ▷ Counter for valid φ's.
4:          foreach i ← 1 to |l| do
                                          ▷ Fetch every object o_i in l.
5:              if φ^I_{i,(i+1)} ≤ 2θ then
                                  ▷ φ^I_{i,(i+1)} is formed by o_i and o_{i+1}.
                              ▷ Assume that φ^I_{|l|,|l|+1} = φ^I_{|I|,1}.
6:                  cnt ← cnt + 1;                     ▷ Increment cnt.
7:              end if
8:          end for
9:          if cnt ≠ |l| then
10:             return true;
                              ▷ Not all φ's are valid. Cannot terminate.
11:         end if
12:     end for
13:     return false;
                              ▷ All φ's are valid. We can terminate.
14: end procedure
```

The pseudo-code of the algorithm to check the termination condition (i.e., function CANNOTTERMINATE) is listed in Algorithm 5. We process every direction order list of the tree node $\langle p, \mathcal{I}_p \rangle$ using the outer loop (lines 2-12). The counter *cnt* is the number of valid partition angles which are bounded by 2θ during the time interval $\mathcal{I}$. The inner loop (lines 4 - 8) processes $(|l|-1)$ partition angles formed by every two adjacent objects from o_1 to $o_{|l|}$ in the list l. If some partition angles are larger than 2θ, we return *true* directly to indicate that we cannot terminate on this branch (line 10). After checking all direction lists and we do not stop beforehand, we return *false* to indicate that we can terminate on this branch.

The time complexity of Algorithm 5 depends on the number of direction lists of the tree node and the number of objects in every direction list. Assume that every direction list has l objects. In the worst case, the number of direction lists is $(\binom{l}{2} + 1)$ as analyzed in Section 5.1.2. Thus, the time complexity of the algorithm is $O((\binom{l}{2} + 1) \times l) = O(l^3)$ in the worst case. In fact, however, the number of lists in practice is very small; we can always consider it as a constant.

5.2 Processing Continuous $\mathbb{R}$-DBS Queries

We extend our running example in order to illustrate the concept of continuous $\mathbb{R}$-DBS queries.

Example 12 Let us extend our snapshot $\mathbb{R}$-DBS query example to the continuous case. Fig. 18 shows that the user q is moving along the edge $e(v_6, v_4)$ from v_6 to v_4 with a constant speed $(1, 0)'$. The user issues a continuous query for the time interval $[0, 6]$. The continuous DBS query predicts the changes of DBS during $[0, 6]$. The result will be:

$$DBS = \begin{cases} \{a, c, d\} & t \in [0, 3.5) \\ \{c, d\} & t \in [3.5, 6] \end{cases} \tag{16}$$

The result indicates that when started at v_6, objects a, c, and d are DBS objects. These three objects remain as DBS objects until the user reaches q' at $t = 3.5$. After that position, object a becomes dominated by object c and hence the result set is changed to $\{c, d\}$. It remains the same until the user reaches v_4. ■

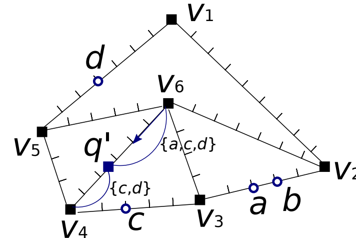


Fig. 18: Example of a continuous $\mathbb{R}$-DBS query

Snapshot $\mathbb{R}$-DBS queries are to retrieve DBS points based on a fixed query point. However, query points might move along road networks. For example, users who carry mobile devices may submit DBS queries even

when they are moving. Consequently, we propose continuous $\mathbb{R}$-DBS queries to support DBS query processing when the location of the query point q keeps changing in a road network. Given a road network $G = \{V, E\}$, and a set of POI objects $P = \{p_1, \ldots, p_n\}$ located on the edges of G, a continuous $\mathbb{R}$-DBS query specifies an edge $e(v_i, v_j)$ as the moving trajectory of a user, and wants to find out all the objects $p \in P$ that are not dominated by any other object w.r.t. any point $q \in e(v_i, v_j)$. To simplify our discussion, we assume q moves only along an edge e of the road network, and the algorithm developed can naturally support the case where q moves along multiple edges.

Similar to continuous $\mathbb{E}$-DBS queries, we need to find out the change moment where the DBS results change, and then convert the continuous DBS query to snapshot DBS queries corresponding to those change moments. However, unlike a continuous $\mathbb{E}$-DBS query which has a time interval parameter τ, a continuous $\mathbb{R}$-DBS query uses the position along the moving trajectory e to indicate the moment DBS results change, i.e., *change positions* to be distinguishable from *change moment* used by continuous $\mathbb{E}$-DBS queries. The change positions partition the moving trajectory e into disjoint sub-segments $e' \subseteq e$ with DBS queries corresponding to the points of one sub-segment sharing the same result. In other words, the answer set $DBS(q, G) = \cup_{e'(e'.l, e'.r) \subseteq e} \langle DBS(e'.l, G), e' \rangle$ where i) $e'.l$ and $e'.r$ refer to the left and right endpoints of the sub-segment e'; ii) $\cup e' = e(v_i, v_j)$; and iii) $\forall e' \wedge \forall q \in e'$, $DBS(q, G) = DBS(e'.l, G)$. In the following, we first identify three important properties related to continuous $\mathbb{R}$-DBS query, and then present the search algorithm.

As observed that as long as query point q moves along the edge $e(v_i, v_j)$, the shortest path from q to an object p located outside e definitely passes either v_i or v_j as the first vertex, i.e., $\forall q \in e(v_i, v_j) \wedge p \notin e(v_i, v_j)$, $SP(q, p).sp_1 \in \{v_i, v_j\}$. In other words, p changes its direction w.r.t. q only when $SP(q, p)$ changes its first vertex from v_i to v_j (or verse visa). Property 6 is thereafter developed to locate the position s_p along $e(v_i, v_j)$ that p changes its direction.

Property 6 Given a query point q moving on an edge $e(v_i, v_j)$ from vertex v_i to vertex v_j and an object p, p changes its direction only when q reaches point s_p along e with $dist(v_i, s_p)$ expressed in Equation (17)[7].

$$dist(v_i, s_p) = \frac{1}{2} \cdot (|e| + ||SP(v_j, p)| - |SP(v_i, p)||), \quad (17)$$

where $|e|$ is the length of edge e, and $|SP(v_i, p)|$ and $|SP(v_j, p)|$ are the shortest distances from v_i, v_j to p, respectively. Specifically, object p does not change its direction while q moves from v_i to s_p along e. □

Example 13 An example is depicted in Fig. 19. Assume q moves along edge $e(v_6, v_4)$. Since $|SP(v_6, a)| = 6$, $|SP(v_4, a)| = 7$, and $|e(v_6, v_4)| = 6$, $dist(v_6, s_a) = \frac{1}{2} \cdot (6 + 7 - 6) = 3.5$. Based on Property 6, we understand that when q moves along the sub-segment $(0, 3.5)$, the shortest path $SP(q, a)$ takes v_6 as the first vertex; when q moves along the sub-segment $(3.5, 6)$, the shortest path $SP(q, a)$ takes v_4 as the first vertex. In other words, a remains its direction w.r.t. q when q moves along the sub-segment $(0, 3.5)$, then it changes the direction when q reaches 3.5, and remains its direction w.r.t. q again when q moves along $(3.5, 6)$. ■

Fig. 19: Property 6 of continuous $\mathbb{R}$-DBS queries

We also observe that the dominance relationship has the continuity property as presented in Properties 7 and 8, respectively.

Property 7 Given an object p and a query point q moving on an edge $e(v_i, v_j)$, if p is dominated when q is located at v_i, and p is dominated when q is located at v_j, object p is guaranteed to be dominated when q is located at any point on the edge $e(v_i, v_j)$. □

Property 8 Given an object p and a query point q moving on an edge $e(v_i, v_j)$, if p is a DBS point when q is located at v_i, and p is a DBS point when q is located at v_j, object p is guaranteed to be a DBS point when q is located at any point on the edge $e(v_i, v_j)$. □

Due to the space limitation, the proofs for Property 7 and Property 8 are presented in Appendix A.4.

Example 14 An example is depicted in Fig. 20. When $q = v_6$, $SP(q, a) = (6, v_6 v_3)$, $SP(q, b) = (7, v_6 v_3)$ and thus $a \prec b$. When $q = v_4$, $SP(q, a) = (7, v_4 v_3)$, $SP(q, b) = (8, v_4 v_3)$ and hence $a \prec b$. Based on Property 7, b is dominated when q is located at any point on the edge

[7] To simplify the presentation, we assume edge $e(v_i, v_j)$ aligns with x-axis, and v_i is located at the origin. Consequently, the position of s_p can be represented by $dist(v_i, s_p)$.

$e(v_6, v_4)$, and hence b will not be a DBS point when q moves along $e(v_6, v_4)$. On the other hand, c is a DBS point w.r.t. $q = v_6$ and $q = v_4$. Consequently, based on Property 8, c is the DBS point w.r.t. q located at any position along $e(v_6, v_4)$. However, a is a DBS point w.r.t. $q = v_6$, but is not a DBS point w.r.t. $q = v_4$. ■

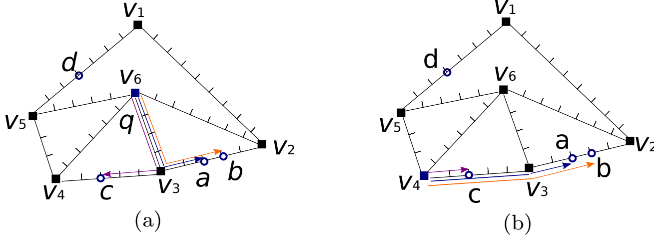


Fig. 20: Properties 7 and 8 of continuous $\mathbb{R}$-DBS queries

Given a continuous DBS query issued at edge $e(v_i, v_j)$, Property 7 guarantees that objects dominated w.r.t. $q = v_i$ and $q = v_j$ are excluded from the answer set, and Property 8 guarantees that DBS points w.r.t. both $q = v_i$ and $q = v_j$ are for sure DBS points w.r.t. q located at any position of $e(v_i, v_j)$. In other words, $DBS(v_i, G) \cup DBS(v_j, G)$ form the superset of the answer set, i.e., candidates of DBS points w.r.t. to $q \in e(v_i, v_j)$ must be in $DBS(v_i, G) \cup DBS(v_j, G)$. Consequently, we can first issue two snapshot $\mathbb{R}$-DBS queries on point v_i and v_j. Based on the returned results $DBS(v_i, G)$ and $DBS(v_j, G)$, two sets, denoted as S_{int} and S_{dif}, are derived. Here, set S_{int} refers to the intersection of $DBS(v_i, G)$ and $DBS(v_j, G)$, i.e., $S_{int} = DBS(v_i, G) \cap DBS(v_j, G)$; and set S_{dif} refers to the rest, i.e., $S_{dif} = DBS(v_i, G) \cup DBS(v_j, G) - S_{int}$. Based on Property 8, all the objects in S_{int} must be DBS points for any point along trajectory $e(v_i, v_j)$ and hence only objects in S_{dif} require evaluations. For each object $p \in S_{dif}$, we find the position s_p along $e(v_i, v_j)$ that p changes its direction based on Property 6. If $p \in DBS(v_i, G)$, p is a DBS point when q moves along subsegment $(0, s_p)$. Otherwise, $p \in DBS(v_j, G)$, and p is a DBS point when q moves along subsegment $e'(s_p, |e(v_i, v_j)|)$.

Let us consider our example again. We issue two snapshot $\mathbb{R}$-DBS queries at the endpoints of $e(v_6, v_4)$, and then obtain $DBS(v_6, G) = \{a, c, d\}$ and $DBS(v_4, G) = \{c, d\}$. Accordingly, we have $S_{int} = \{c, d\}$, and $S_{dif} = \{a\}$. As object a is the only point in S_{dif}, we derive $s_a = 3.5$ based on Eq (17) and thus a is a DBS point along subsegment $(0, 3.5)$. Consequently, the continuous DBS query issued at edge $e(v_6, v_4)$ has $\{\langle\{a, c, d\}, (0, 3.5)\rangle, \langle\{c, d\}, (3.5, 6)\rangle\}$ as the answer set. Algorithm 6 presents the procedure of continuous $\mathbb{R}$-DBS Queries.

Algorithm 6 Continuous $\mathbb{R}$-DBS Query

```
1: procedure ContinuousRDBSQuery(G,P,e^q(v1,v2))
2:     DBS_v1 ← SnapshotRDBSQuery(v1, G, P);
                                   ▷ DBS objects when q is at v1
3:     DBS_v2 ← SnapshotRDBSQuery(v2, G, P);
                                   ▷ DBS objects when q is at v2
4:     X ← ∅;   ▷ positions where an object's direction changes
5:     foreach p ∈ DBS_v1 ∪ DBS_v2 − DBS_v1 ∩ DBS_v2 do
6:         dist_v1,p ← Dist(v1, p);
                      ▷ the length of the shortest path from v1 to p
7:         dist_v2,p ← Dist(v2, p);
                      ▷ the length of the shortest path from v2 to p
8:         x ← (|e^q| + |dist_v2,p − dist_v1,p|)/2;
                          ▷ the position where p's direction changes
9:         X ← X ∪ x;                          ▷ Add x into X.
10:    end for
11:    SG ← GetSubSegments(e^q, X);
                                      ▷ sub-segments split by X
12:    foreach e_i ∈ SG do
13:        DBS_mid_i ← SnapshotRDBSQuery(mid_i, G, P);
              ▷ DBS objects when q is at the midpoint mid_i of e_i.
14:        DBS ← DBS ∪ {⟨DBS_mid_i, e_i⟩};
                                 ▷ Add {⟨DBS_mid_i, e_i⟩} to DBS.
15:    end for
16:    return DBS;
17: end procedure
```

6 Experiments

In this section, we report the experimental evaluation. In the following, we first explain the detailed settings of the experimental study, and then present the experimental results in the Euclidean space $\mathbb{E}$ and the road network $\mathbb{R}$, respectively.

6.1 $\mathbb{E}$-DBS Queries

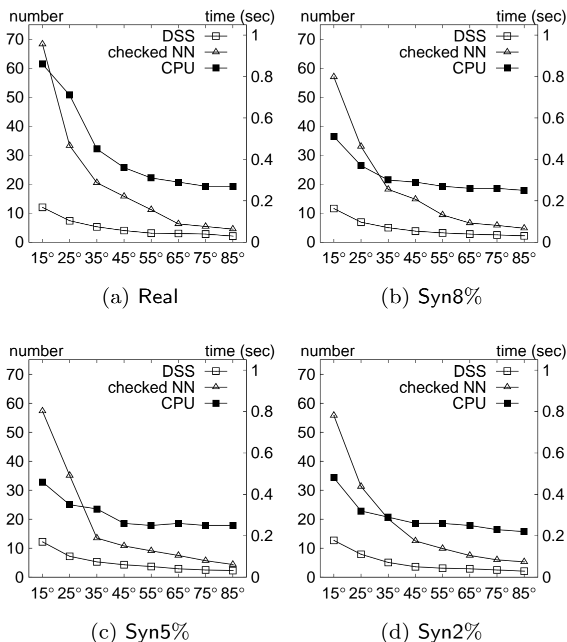
In the Euclidean space $\mathbb{E}$, we use both real and synthetic datasets, with their properties summarized in Table 5. For the real dataset, denoted as Real, we consider the road line segments of *Long Beach* in the TIGER database [17], and extract the midpoint for each road line segment to form a point dataset. It in total consists of 50,747 points normalized in $[0, 1000] \times [0, 1000]$ space. The synthetic datasets, denoted as Synρ%, are generated based on the uniform distribution in the $[0, 1000] \times [0, 1000]$ spaces, with density ρ indicating the average number of points falling into $[0, 1] \times [0, 1]$ unit. All the datasets are indexed in R*-trees [1] with the page size set to 8,192 bytes. All the algorithms are implemented in GNU C++ and conducted on an Intel Core2 Duo 2.40 GHz PC with 2.0 GB RAM running Ubuntu Linux 2.6.31.

Table 5: Datasets

Dataset	Cardinality	Density (ρ)
Real	50,747	–
Syn8%	80,000	0.08
Syn5%	50,000	0.05
Syn2%	20,000	0.02

6.1.1 Performance of Snapshot $\mathbb{E}$-DBS Queries

First, we evaluate the performance of snapshot $\mathbb{E}$-DBS queries. We consider the number of DBS objects, the number of checked nearest neighbors, and the CPU costs, denoted as DSS, checked NN, and CPU, as the performance metrics. The performance of snapshot queries under different θ values for different datasets is depicted in Fig. 21.



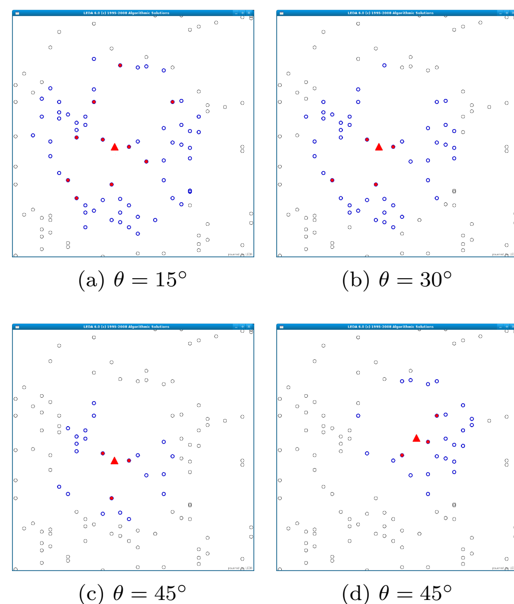
(a) Real (b) Syn8% (c) Syn5% (d) Syn2%

Fig. 21: Performance of snapshot queries w.r.t. θ

As shown in Fig. 21, the total number of DBS objects changes when θ varies in the range of $[15°, 85°]$. The number decreases while θ increases. The reason behind is that an object can dominate larger angle ranges given a larger θ and hence more objects are dominated and excluded from the DBS result. On the other hand, we also observe that the number of DBS objects is not affected by the densities of datasets.

The number of checked NN also changes with different θ's. It decreases when θ increases because it is easier to reach the early termination condition with a larger θ. Consequently, fewer nearest neighbors are approached to obtain the final results. We also observe that the number of NNs evaluated is much smaller than the total number of the objects in the dataset, as roughly only 0.14% of data points are evaluated. These results show that our algorithms can respond to snapshot queries promptly and have good performance. Additionally, the number reduces fast when θ is small ($\theta < \pi/4$) and reduces steadily when θ is relatively large ($\theta > \pi/4$). It means that our algorithms can achieve more stable performance with relatively larger θ's.

The CPU cost depends on the number of checked nearest neighbors. It decreases when θ increases, but the CPU cost is independent on the densities of the datasets.



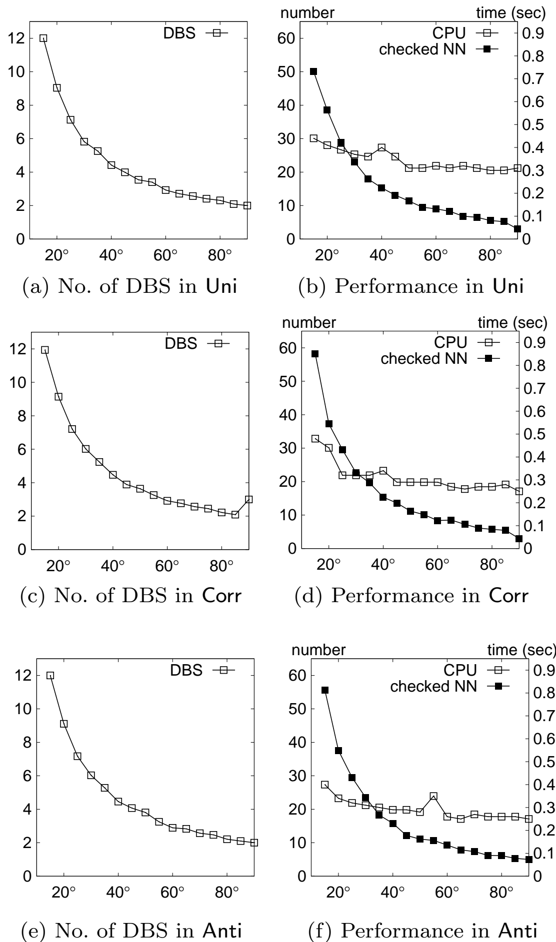
(a) $\theta = 15°$ (b) $\theta = 30°$ (c) $\theta = 45°$ (d) $\theta = 45°$

Fig. 22: Images of snapshot queries

We also capture some images of the DBS points corresponding to different θ's and user position for the snapshot case, as depicted in Fig. 22. The triangular shape point in the center refers to the user's position (i.e., q), the red solid points are the DBS objects, and the blue hollow ones are the checked nearest neighbors. Fig. 22(a), Fig. 22(b) and Fig. 22(c) refer to the case where the user position is fixed but θ value changes. We can observe that as θ increases, both the number of DBS objects and the number of checked NN objects reduce. On the other hand, Fig. 22(c) and Fig. 22(d) demonstrate the case where θ value is fixed at 45° but user positions change.

In addition, we also evaluate the impact of data distributions on the search performance. Three synthetic datasets (denoted as Corr, Anti and Uni) are generated

with each consisting of 10,000 objects in $[0, 1000] \times [0, 1000]$ space. Corr simulates correlated distribution with a correlated coefficient 0.6, Anti simulates the anti-correlated distribution with a correlated coefficient -0.6, and data set Uni simulates the uniform distribution. 100 random queries are issued and the average performance is reported in Fig. 23. We observe that the average number of DBSs, the number of checked NNs, and the CPU cost are not affected by the object distributions.



(a) No. of DBS in Uni (b) Performance in Uni

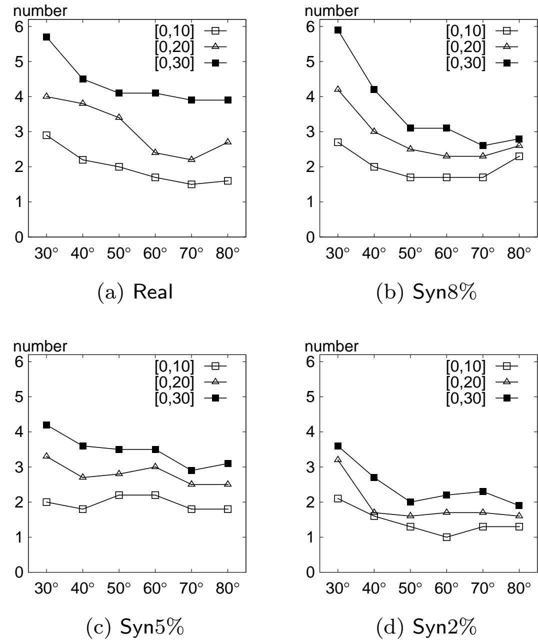
(c) No. of DBS in Corr (d) Performance in Corr

(e) No. of DBS in Anti (f) Performance in Anti

Fig. 23: Performance of snapshot queries for data sets with different distributions

6.1.2 Performance of Continuous $\mathbb{E}$-Queries

In the experiments of continuous DBS queries, we evaluate the number of change moments, the size and the depth of the process tree, and the CPU cost under different θ values. We consider the scenario such that the user randomly select a position as the starting point and then keeps moving with a constant speed of 0.06 unit distance per unit time[8] along the positive x-axis during different time intervals (i.e., $[0, 10]$, $[0, 20]$ and $[0, 30]$).



(a) Real (b) Syn8%

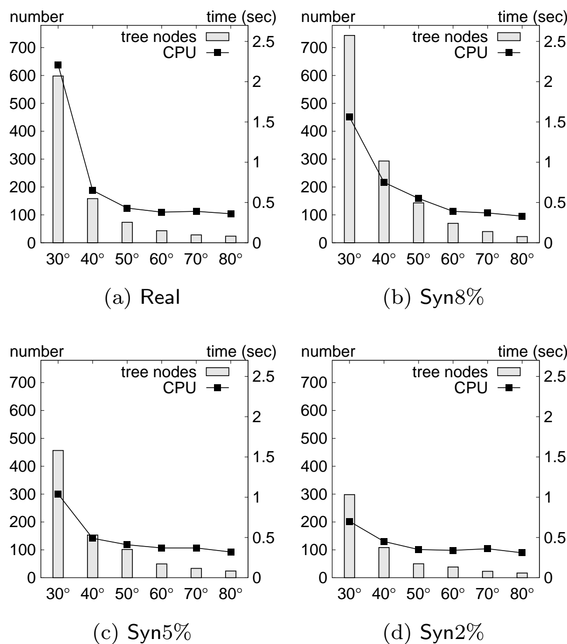
(c) Syn5% (d) Syn2%

Fig. 24: Number of change moments of continuous queries w.r.t. θ

Fig. 24 shows the number of change moments under different θ's and different time intervals of different datasets. It is observed that in general the number of change moments decreases while θ increases in a small θ-range, but it keeps steady once θ reaches a large value. This is because an object p_i can dominate all the objects p_j with $p_j \in [w_i - \theta, w_i + \theta]$ and $d_j > d_i$. Given a large θ, the range $[w_i - \theta, w_i + \theta]$ does not change much when the user moves. It also means that result to a continuous DBS query becomes stable for a large θ. We also observe that the number of change moments becomes smaller when the dataset has a lower density. When the dataset has less objects, each DBS point dominates less points and hence the user's movement causes less changes on the objects dominated by p. Last but not least, the number of change moments also decreases while the time interval becomes shorter.

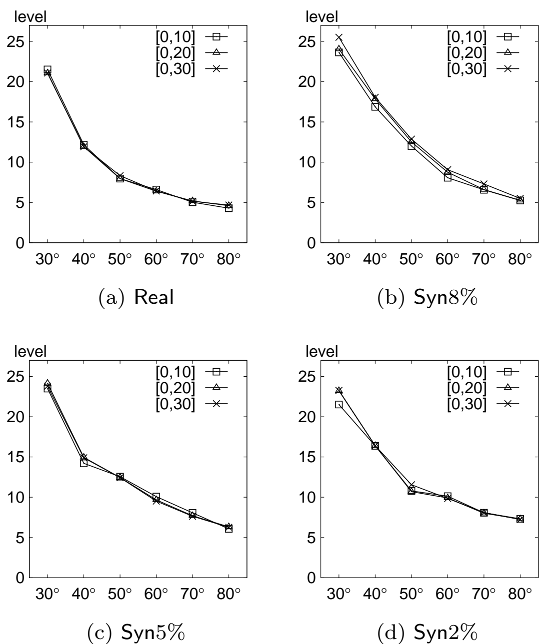
Fig. 25 illustrates the sizes of the process trees and the CPU costs under different θ's when the time interval is fixed at $[0, 30]$. In general, the process tree reduces

[8] We simulate the user's moving speed as human's average walking speed 1 m/s. In the space of our datasets, 1 unit distance equals to 1 kilometer approximately and we regard 1 unit time as 1 minute.



(a) Real (b) Syn8% (c) Syn5% (d) Syn2%

Fig. 25: Tree sizes and CPU costs of continuous queries vs. θ w.r.t. time interval $[0, 30]$



(a) Real (b) Syn8% (c) Syn5% (d) Syn2%

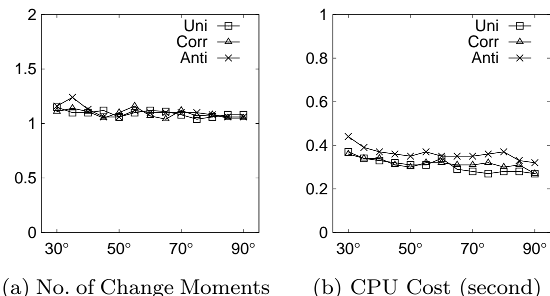
Fig. 26: Tree depths of continuous queries

its size when θ increases; because the larger the θ is, the easier the termination condition is achieved. It is observed that only a small number of objects (in average around 1.4% of the dataset) require evaluation. It demonstrates that our termination strategy works well for continuous queries and it is possible to respond to continuous queries promptly. Additionally, the object density has a direct impact on the size of the process tree. This is because the process tree is constructed based on the distance order of objects, and the order changes less frequently when the dataset density is smaller.

The CPU cost depends on the size of the process tree and thus it has the same tendency as the tree size—the query cost also decreases when θ grows and/or the object density decreases.

In Fig. 26, we also present tree depths for continous queries under different θ's and different time intervals. The tree depth decreases when θ grows because we can terminate the query procedure earlier when θ is larger. On the other hand, the tree depth is not affected by object density. It means that the growth of the tree size is caused by the increase of the branches. In addition, the tree depth is not affected by the length of the time interval. Therefore, our algorithms for continuous DBS queries are stable enough for different object densities and different time interval lengths.

We also evaluate performance of continuous DBS queries using data sets Uni, Corr and Anti with different object distributions. Fig. 27 (a) shows the numbers of change moments for the three data sets with different object distributions. Fig. 27 (b) shows the CPU costs for the three data sets. The number of change moments and the CPU costs are not influenced by object distributions obviously.



(a) No. of Change Moments (b) CPU Cost (second)

Fig. 27: Performance of continuous queries for data sets with different distributions

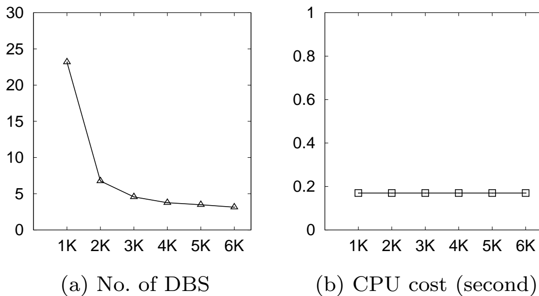
6.2 ℝ-DBS Queries

In evaluating the performance of ℝ-DBS queries, we adopt the map of Oldenburg (OL) with 6105 nodes and 7035 edges as the road network which is obtained from [32] and also used in [31]. Six object sets with different cardinalities are generated via randomly extracting the points from the line segments of the road

network. All the algorithms are implemented in GNU C++ and the experiments are conducted on an Intel Core2 Duo 2.40 GHz PC (2.0 GB RAM) with a Fedora Linux 2.6.32.

6.2.1 Snapshot $\mathbb{R}$-DBS Queries

In the experiments of snapshot $\mathbb{R}$-DBS queries, 100 snapshot DBS queries are generated randomly on the road network, and the average performance under different object set cardinalities is reported in Fig. 28. As observed, the number of DBS objects decreases while the size of object set increases. This is because an object is more likely to be dominated if there are more objects. On the other hand, the cardinality does not have a significant impact on the CPU cost. We use the Dijkstra algorithm to check the shortest paths for candidate objects rather than all. After finding out the shortest path for an vertex, we only consider the nearest objects on each adjacent edge because the further ones are definitely dominated by the nearest one. In order to illustrate the DBS objects in road network, we show an image of a snapshot query at an edge in the OL road network in Appendix A.5.

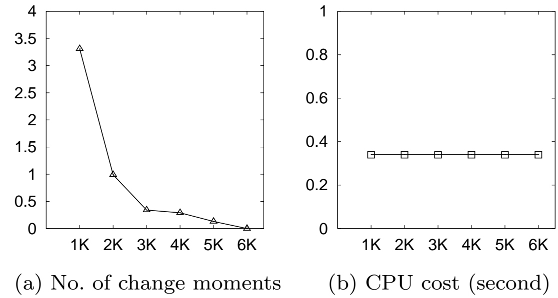


(a) No. of DBS (b) CPU cost (second)

Fig. 28: Performance of snapshot $\mathbb{R}$-DBS queries w.r.t. the cardinality of the object set

6.2.2 Continuous $\mathbb{R}$-DBS Queries

In the experiments of continuous queries in the OL road network, we also issue 100 random queries and compare the average number of change moments in Fig. 29 (a) and the average CPU cost in Fig. 29 (b). The horizontal axis represents the cardinalities of object set varying in the range of $\{1K, 2K, \ldots, 6K\}$ and the vertical axis represents the number of change moments and the CPU cost, respectively. It is observed that the change moments decreases while the object set size increases, and the CPU cost is independent of the object set size. We also show images of a continuous query at an edge in the OL road network in Appendix A.5.



(a) No. of change moments (b) CPU cost (second)

Fig. 29: Performance of continuous $\mathbb{R}$-DBS queries

7 Conclusions

In this paper, we have proposed a new type of spatial queries called *direction-based surrounder* (*DBS*) queries. In particular, we have studied two classes of DBS queries, including *snapshot DBS queries* and *continuous DBS queries*. We developed efficient algorithms for processing snapshot DBS queries and continuous DBS queries, respectively. Based on extensive experiments with both real and synthetic datasets, we demonstrated the performance of our proposed algorithms. The experimental results confirm that the proposed algorithms work well for both the snapshot case and the continuous case.

In the future, we would like to extend our work in the following four interesting directions. First, we intend to explore other approaches (e.g., using the MBR-trimming method in [45]) to tackle the DBS query. Second, we plan to extend our work considering non-spatial attributes. For example, we can consider a query such as “find near and less expensive hotels around me”. In this case, we should consider the non-spatial attribute “price” as well as the distance and the direction. The third interesting one is the situation of a moving user along with several moving objects. Assuming that a football game is going on, a player wants to pass the ball to his teammate. His teammates and adversaries have different directions and distances according to his current position. In this situation all the objects are moving including the user. We can help this football player to make a good decision of passing the ball by considering distances and directions. The last one is to construct a prototype system to provide DBS query services for mobile users based on the proposed ideas.

Acknowledgements Xi Guo and Yoshiharu Ishikawa were partly supported by the Grant-in-Aid for Scientific Research (#21013023, #22300034) from the Japan Society for the Promotion of Science (JSPS) and the FIRST Program, Japan. Yunjun Gao was supported in part by NSFC Grant 61003049, ZJNSF Grant Y1100278, the Fundamental Research Funds for the Central Universities under Grant No.2010QNA5051, the Key Project of Zhejiang University Excellent Young Teacher Fund (Zijin Plan), and the Re-

turned Scholar Funds for the Personnel Office of Zhejiang Province. The authors are grateful to the anonymous reviewers and Prof. Dr. Ralf Hartmut Güting and Prof. Dr. Nikos Mamoulis, the guest editors, for their constructive and insightful advice. They also appreciate the valuable comments from Prof. Dr. Cyrus Shahabi, Prof. Dr. Hiroyuki Kitagawa, and Dr. Xin Xie.

References

1. N. Katayama, "R*-tree Library," Dec. 1997; `http://research.nii.ac.jp/~katayama/homepage/research/srtree/English.html`.
2. S. Börzsönyi, D. Kossmann, and K. Stocker, "The Skyline Operator," ICDE, pp. 421-430, 2001,
3. Z. Chen, H. T. Shen, X. Zhou, and J. X. Yu, "Monitoring Path Nearest Neighbor in Road Networks," SIGMOD, pp. 591-602, 2009.
4. J. Chomicki, P. Godfrey, J. Gryz, and D. Liang, "Skyline with Presorting," ICDE, pp. 717-719, 2003.
5. S. Nutanong, E. Tanin, and R. Zhang, "Visible Nearest Neighbor Queries," DASFAA, pp. 876-883, 2007.
6. Y. Gao, B. Zheng, G. Chen, W.-C. Lee, and G. Chen, "Continuous Visible Nearest Neighbor Queries," EDBT, pp. 144-155, 2009.
7. GNU, "GNU Scientific Library," `http://www.gnu.org/software/gsl/`.
8. Z. Huang, H. Lu, B. C. Ooi, and A. K. H. Tung, "Continuous Skyline Queries for Moving Objects," TKDE, vol.18, no.12, pp. 1645-1658, 2006.
9. K. C. K. Lee, W.-C. Lee, and H. V. Leong, "Nearest Surrounder Queries," ICDE, pp. 85, 2006.
10. D. Papadias, Y. Tao, G. Fu, and B. Seeger, "An Optimal and Progressive Algorithm for Skyline Queries," SIGMOD, pp. 467-478, 2003.
11. K. Patroumpas and T. Sellis, "Monitoring Orientation of Moving Objects Around Focal Points," SSTD, pp. 228-246, 2009.
12. K. Raptopoulou, A. Papadopoulos, and Y. Manolopoulos, "Fast Nearest-Neighbor Query Processing in Moving-Object Databases," GeoInformatica, vol.7, no.2, pp. 113-137, 2003.
13. P. Godfrey, R. Shipley, and J. Gryz, "Maximal Vector Computation in Large Data Sets," VLDB, pp. 229-240, 2005.
14. J. Schiller and A. Voisard, *Location-Based Services*, first ed., Morgan Kaufmann, 2004.
15. Y. Tao, D. Papadias, and Q. Shen, "Continuous Nearest Neighbor Search," VLDB, pp. 287-298, 2002.
16. X. Guo, Y. Ishikawa, and Y. Gao, "Direction-Based Spatial Skylines," MobiDE, pp. 73-80, 2010.
17. "TIGER, U.S. Census Bureau," `http://tiger.census.gov/`.
18. B. Zheng, K. C. K. Lee, and W.-C. Lee, "Location-Dependent Skyline Query," MDM, pp. 148-155, 2008.
19. N. Chen, L. Shou, G. Chen, Y. Gao and J. Dong, "Predictive Skyline Queries for Moving Objects," DASFAA, pp.278-282, 2009.
20. S. Šaltenis, C.S. Jensen, S.T. Leutenegger, and M.A. Lopez, "Indexing the Positions of Continuously Moving Objects," SIGMOD, pp.331-342, 2000.
21. M.-W. Lee and S.-W. Hwang, "Continuous Skylining on Volatile Moving Data," DBRank, pp.1568-1575, 2009.
22. X. Huang and C.S. Jensen, "'In-Route Skyline Querying for Location-Based Services," W2GIS, pp.120-135, 2004.
23. W.-T. Balke and U. Güntzer, "Multi-objective Query Processing for Database Systems," VLDB, pp. 936-947, 2004.
24. A. Vlachou, C. Doulkeridis and Y. Kotidis, "Angle-based Space Partitioning for Efficient Parallel Skyline Computation," SIGMOD, pp.227-238, 2008.
25. Z. Huang, C.S. Jensen, H. Lu, and B.C. Ooi, "Skyline Queries Against Mobile Lightweight Devices in MANETs," ICDE, pp.66, 2006.
26. K. C. K. Lee, J. Schiffman, B. Zheng, W.-C. Lee, and H. V. Leong, "Tracking Nearest Surrounders in Moving Object Environments," ICPS, pp. 3-12, 2006.
27. K. C. K. Lee, J. Schiffman, B. Zheng, W.-C. Lee, and H. V. Leong, "Round-Eye: A system for tracking nearest surrounders in moving object environments," JSS, vol.80, issue 12, pp. 2063-2076, 2007.
28. E. W. Dijkstra, "A Note on Two Problems in Connexion with Graphs," Numerische Mathematik, vol. 1, pp. 269-271, 1959.
29. N. Jing, Y.-W. Huang, and E. A. Rundensteiner, "Hierarchical Encoded Path Views for Path Query Processing: An Optimal Model and Its Performance Evaluation," TKDE, vol.10, no.3, pp. 409- 432, 1998.
30. F. Wei, "TEDI: efficient shortest path query answering on graphs," SIGMOD, pp. 99-110, 2010.
31. M. Yiu and N. Mamoulis, "Clustering Objects on a Spatial Network," SIGMOD, pp. 443-454, 2004.
32. T. Brinkhoff, "A Framework for Generating Network-Based Moving Objects," GeoInformatica, vol.6, no.2, pp.153-180, 2002.
33. H. Hu, D.L. Lee, and J. Xu, "Fast Nearest Neighbor Search on Road Networks, " EDBT, pp. 186-203, 2006.
34. C. Shahabi, M.R. Kolahdouzan, and M. Sharifzadeh, "A Road Network Embedding Technique for K-Nearest Neighbor Search in Moving Object Databases, " GIS, pp. 94-100, 2002.
35. K.C.K. Lee, W.-C. Lee, and B. Zheng, "Fast Object Search on Road Networks, " EDBT, pp. 1018-1029, 2009.
36. C. Jensen, J. Kolářvr, T. Pedersen and I. Timko, "Nearest Neighbor Queries in Road Networks," GIS, pp. 1-8, 2003.
37. H.-J. Cho, and C.-W. Chung, "An efficient and scalable approach to CNN queries in a road network, " VLDB, pp. 865-876, 2005.
38. H. Samet, Hanan, J. Sankaranarayanan, and H. Alborzi, "Scalable network distance browsing in spatial databases, " SIGMOD, pp. 43-54, 2008.
39. U. Demiryurek, F. Banaei-Kashani, and C. Shahabi, "Efficient Continuous Nearest Neighbor Query in Spatial Networks Using Euclidean Restriction, " SSTD, pp. 25-43, 2009.
40. D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao, "Query processing in spatial network databases," VLDB, pp. 802-813, 2003.
41. M. Kolahdouzan and C. Shahabi, "Voronoi-based K nearest neighbor search for spatial network databases," VLDB, pp. 840-851, 2004.
42. H. Hu, D.L. Lee, and V.C.S. Lee, "Distance indexing on road networks," VLDB, pp. 894-905, 2006.
43. K. Mouratidis, M. Yiu, D. Papadias, and N. Mamoulis, "Continuous nearest neighbor monitoring in road networks," VLDB, pp. 43-54, 2006.
44. M.R. Kolahdouzan and C. Shahabi, "Continuous K Nearest Neighbor Queries in Spatial Network Databases," STDBM, pp.44-50, 2004.
45. Y. Tao, D. Papadias, and X. Lian, "Reverse kNN Search in Arbitrary Dimensionality," VLDB, pp. 744-755, 2004.