DPS: A Novel Approach for Efficient Direction-Based Neighborhood Queries

Pedro Henrique Bergamo Bertolli and Marcela Xavier Ribeiro
Department of Computer Science
Universidade Federal de São Carlos (UFSCar)
e-mail: pbertolli@estudante.ufscar.br, marcelaxr@ufscar.br

***Abstract*—Current spatial search methods predominantly focus on distance-based metrics, while direction-based queries have emerged to address applications requiring diverse directional coverage. Existing direction-based approaches like the Direction-Based Surrounder (DBS) and Direction-Aware Nearest Neighbor (DNN) employ iterative algorithms that require examining multiple objects and their spatial relationships, leading to high computational costs particularly in dense datasets. These methods also suffer from either overly restricted results (DBS) or directionally clustered outcomes (DNN) due to their selection criteria. This paper introduces Direction Proximity Search (DPS), a novel approach that ensures directional diversity—defined as having at most one object per angular interval—while significantly reducing computational overhead. By employing geometric space partitioning to divide the search space into equal angular regions and a refinement phase that selects the nearest object per directional interval, DPS eliminates the need for extensive object-to-object comparisons. Experiments on both synthetic and real datasets show that DPS achieves processing time reductions of up to 99.9% specifically for high-density distributions (Bit and Sierpinski) with large datasets, while consistently maintaining the desired directional diversity property across all tested configurations.**

***Keywords*-Spatial databases; surrounding queries; efficient processing; directional diversity.**

I. Introduction

Spatial queries with directional diversity are essential for critical applications where distance alone cannot guarantee accessibility. In emergency response scenarios—such as fires, floods, or traffic incidents—the nearest facilities may be unreachable, making it crucial to identify alternatives distributed across different directions. The widespread adoption of mobile devices has made spatial data processing essential in various domains, including location-based recommendations, route planning, environmental monitoring, and urban mapping. These applications rely on spatial queries to retrieve and analyze geographic information, helping users make informed decisions based on their spatial context.

Spatial query processing typically relies on Geographic Information Systems (GIS) and spatial databases. These systems manage geometric objects (points, lines, and polygons) that represent entities in the real world. For example, a restaurant can be represented as either a simple point or, more precisely, as a polygon depicting its physical boundaries. The query point in these systems could represent various entities: a mobile user's location, a point of interest, or a vehicle's projected position.

Although distance-based queries are prevalent, incorporating directional diversity has become increasingly crucial. This is particularly evident in emergency scenarios, where the nearest service point may not be the most accessible. During a fire, for example, the closest hospitals or fire stations might be inaccessible due to smoke or the spread of the fire. Similarly, during floods, nearby shelters could be in areas prone to submersion or landslides. In urban settings, traffic congestion, road closures, or construction work can render the closest facilities temporarily unreachable, highlighting the need for directionally diverse alternatives.

To address the limitations of purely distance-based approaches, nearest surrounder queries [1] were introduced as queries that consider both distance and direction of objects in relation to a query point. Subsequently, the DBS [2] and DNN [3] queries emerged as variations of this approach. These queries employ a fundamental concept called "dominance relation", which uses direction and distance properties to determine which objects should be included in the result set. While DBS applies dominance relations between pairs of objects, resulting in more restricted results, DNN considers object triplets, potentially yielding more diversity, but sometimes spatially concentrated outcomes.

In critical applications, particularly emergency planning and response, the speed of information delivery is crucial. Decision-makers need instant access to results to plan their actions and execute the necessary procedures. However, current approaches face two main limitations: computational inefficiency due to iterative processing and suboptimal result distribution that is either too restrictive or lacks sufficient directional spread.

This paper makes three key contributions: (1) a novel geometric partitioning strategy that efficiently handles direction-based queries; (2) substantial computational efficiency improvements over existing methods; and (3) comprehensive experimental evaluation that demonstrates scalability across diverse datasets. The remainder of the paper is organized as follows: Section II reviews related work in spatial queries and direction-based methods. Section III introduces our novel Direction Proximity Search (DPS) approach, detailing its partitioning, processing, and refinement phases. Section IV describes our experimental evaluation methodology and datasets. Section V discusses the performance results and comparative analysis. Finally, Section VI concludes the paper and outlines future research directions.

II. Related Work

This section presents a systematic review of the literature on direction-based neighborhood queries and optimization techniques. The research was carried out in the major digital libraries (IEEE, Science Direct, Springer, ACM DL, and Google Scholar), resulting in 11 relevant studies after applying selection criteria. The analysis revealed six main categories of

approaches: spatial indexing (C1), formal query definitions (C2), dominance-based algorithms (C3), computational geometry techniques (C4), visibility-based direction methods (C5), and performance testing (C6). Most works span multiple categories, demonstrating the interconnected nature of these approaches. Regarding spatial indexing (C1), Lee et al. [1] introduced direction-based neighborhood queries with the *sweep* and *ripple* algorithms using R-tree structures. Zhang et al. [4] and Chung et al. [5] expanded this approach, while Nutanong et al. [6] developed R*-Tree pruning techniques to reduce disk access. For formal query definitions (C2), Lee et al.'s work [1] established the theoretical foundations that supported subsequent studies, notably the DBS and DNN queries presented by Guo [2][3]. In dominance-based algorithms (C3), the relationship between objects determines the result set. Table I summarizes the key characteristics and computational limitations of the main direction-based query methods: DBS and DNN.

TABLE I
CHARACTERISTICS OF EXISTING DIRECTION-BASED QUERY METHODS

Method	Time Complexity (worst case)	Dominance Relation	Result Distribution
DBS	$O(n^2)$	Pairwise (2θ interval)	Sparse (uniform coverage)
DNN	$O(n^2)$	Triplet-based (relaxed criteria)	Dense (potential clustering)

As shown in Table I, the DBS algorithm [2] requires $O(n^2)$ comparisons in the worst case to examine all object pairs. Its restrictive dominance relationship, where objects dominate within a 2θ angular interval, can lead to overly limited result sets, especially with larger θ values where a single object can eliminate many candidates within its dominance range. The DNN algorithm [3] provides better directional diversity through less restrictive dominance rules but still has $O(n^2)$ worst-case complexity, making it computationally expensive for large datasets. Additionally, its relaxed dominance criteria can result in directionally close objects being returned, potentially compromising the spatial distribution consistency despite producing larger result sets.

For computational geometry techniques (C4) and visibility-based methods (C5), Lee et al. [1], Nutanong et al. [6], and Chung et al. [5] grounded the direction aspect as a visibility field. Nutanong et al. introduced the concept of minimum visible distance (MinViDist), while Chung et al. relied on angle and direction calculations. Regarding performance testing (C6), Carniel [7], [8] focused on general spatial query definitions, discussing future optimization challenges. A significant gap exists in the literature: the absence of comparative performance analyses between different algorithms, and the fundamental trade-off between computational efficiency and directional diversity in existing methods.

III. Direction Proximity Search

This section presents the DPS method and its implementation. We detail its architecture and operation, introducing a novel direction-based neighborhood query that addresses key limitations in existing approaches.

We begin by formally defining the core concepts of DPS. The parameter θ determines the directional diversity of the result set - it ensures that returned objects are separated by at least θ degrees. For any two objects $o_i, o_j \in \mathcal{D}$ relative to query point q, their angular separation is the angle between vectors $\overrightarrow{qo_i}$ and $\overrightarrow{qo_j}$, denoted as $\angle(\overrightarrow{qo_i}, \overrightarrow{qo_j})$.

In DPS, an object o dominates an angular region R of size θ if: (i) o is the nearest object to q within R, and (ii) no closer object exists within $\theta/2$ degrees of o. This ensures directional diversity by allowing at most one object per θ interval, resulting in a maximum of $\lceil 360/\theta \rceil$ objects in the result set.

DPS employs geometric partitioning to divide the 360° space around q into $n = 360/(\theta/2)$ equal partitions. Each partition spans $\theta/2$ degrees, allowing two adjacent partitions to form a complete θ interval. This approach eliminates the $O(n^2)$ pairwise comparisons required by DBS and DNN. For a fixed θ, DPS achieves $O(n)$ time complexity, as the number of partitions $k = 360/(\theta/2)$ is constant.

A. Partitioning

The partitioning algorithm systematically divides the spatial domain around the query point into equal geometric regions. This geometric partitioning constitutes the initial phase of the DPS query, formally defined as *DPS = (t, q, θ, distMax)*, where *t* denotes the dataset, *q* represents the query point, θ specifies the angular constraint, and *distMax* determines the maximum search radius.

The number of partitions is defined by $\varphi_n = \frac{360°}{\theta/2}$. Each partition has an angular interval of $\theta/2$, allowing two adjacent partitions to form a complete θ interval. The first partition φ_1 is constructed using Algorithm 1.

Algorithm 1 First Partition Construction

Require: Query point q, dataset $\mathcal{D}$, angle θ, distance $distMax$
Ensure: First partition φ_1
1: $NN \leftarrow$ FindNearestNeighbor(q, $\mathcal{D}$)
2: $NN' \leftarrow$ Project(NN, $distMax$)
3: $\vec{v} \leftarrow \overrightarrow{qNN'}$
4: $l_s \leftarrow$ Rotate($\vec{v}$, $\theta/2$)
5: $\varphi_1 \leftarrow$ CreatePolygon(q, NN', l_s)
6: **return** φ_1

To illustrate this process, we use a sample dataset with 13 points and parameters *DPS = (sample, POINT(0 0),* $90°$*, 200000)*. With $\theta = 90°$, we obtain $\varphi_n = 8$ partitions. The nearest neighbor to query point POINT(0,0) is point *a*.

Following Algorithm 1, we project point *a* to create NN' at distance 200000, then rotate the vector $\overrightarrow{qNN'}$ by $\theta/2$ to obtain the upper boundary l_s. The resulting polygon forms the first partition φ_1, as illustrated in Figure 1.

Subsequent partitions are created in clockwise direction using Algorithm 2, which systematically generates all φ_n partitions.

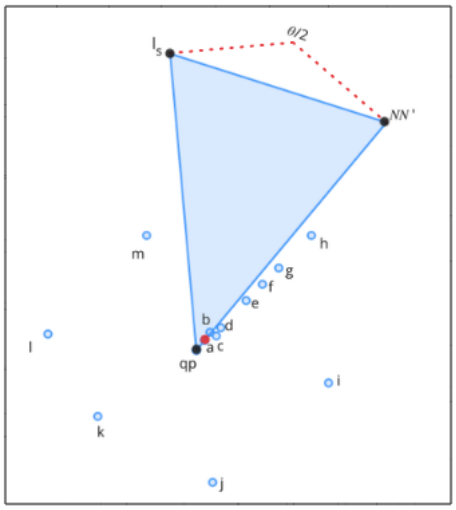

Figure 1. Construction of φ_1.

Algorithm 2 Complete Partitioning

Require: First partition φ_1, angle θ, number of partitions φ_n
Ensure: Set of partitions Φ

```
Φ ← {φ_1}
for i = 2 to φ_n do
    l_prev ← GetUpperBoundary(φ_{i−1})
    l_new ← Rotate(l_prev, −θ/2)
    φ_i ← CreatePartition(l_prev, l_new)
    Φ ← Φ ∪ {φ_i}
end for
return Φ
```

B. Processing

This step is responsible for finding the nearest object to q within each partition. The process requires identifying all objects that intersect with each partition and determining the one closest to q. The partitioning strategy enables an optimized processing approach by confining the search to individual partitions, where only a single nearest object needs to be identified. This significantly reduces computational overhead compared to traditional methods that require multiple object comparisons to establish dominance relationships.

The processing is performed sequentially φ_n-1 times, once for each partition except the first one, which already has its Nearest Neighbor (NN) calculated during the partitioning step. Following the geometric partitioning in our example, this step identifies the nearest objects to the query point *qp* for each partition. These objects, highlighted in red in Figure 2, are accompanied by a table that presents their distances in ascending order and their directions relative to *qp*.

The key advantage of this approach is that it reduces processing to φ_n-1 sequential operations, whereas traditional methods require multiple comparisons among objects until either meeting stopping conditions or, in the worst case, examining the entire dataset.

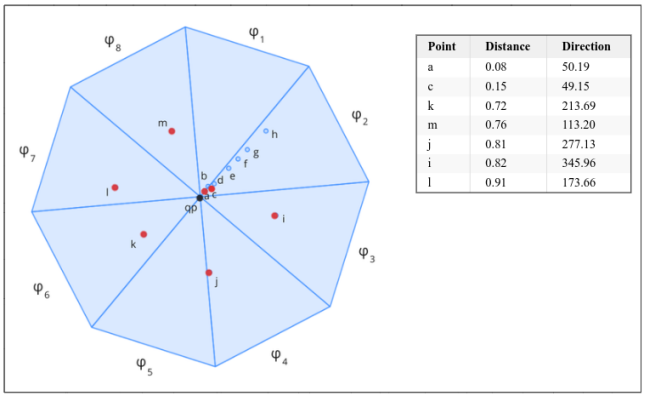

Figure 2. Objects identified as NN for each partition and their respective directions and distances relative to *qp*.

C. Refinement

After identifying all NN of q in their respective partitions, the refinement step ensures directional diversity. Objects are considered directionally close if their angular separation is less than $\theta/2$. The refinement merges adjacent partitions into composite partitions of size θ, selecting only the nearest object from each composite partition.

To formalize the refinement process, we introduce the following definitions:

Definition 1 (Ordered Processing List): The processing result is a list of tuples containing partition identifier, NN object, and distance from q to NN, $List_p$ = ($\varphi_{i_{\text{id}}}$, NN_i, *dist(q,*NN_i*)*), ... ($\varphi_{n_{\text{id}}}$, NN_n, *dist(q,*NN_n*)*), sorted by ascending distance.

Definition 2 (Adjacent Partition): Adjacent partitions comprise predecessor and successor partitions in an ordered partition list.

Definition 3 (Ignored Partition): A partition is marked as ignored if its NN object is at an angular distance less than $\theta/2$ from the NN of a dominant partition.

The refinement algorithm (Algorithm 3) systematically processes the ordered list to determine the final result set.

Algorithm 3 DPS Refinement

Require: Ordered processing list $List_p$
Ensure: Result set R

```
ignored ← ∅
R ← ∅
for each (φ_i, NN_i, dist_i) in List_p do
    if φ_i ∉ ignored then
        R ← R ∪ {NN_i}
        adjacent ← GetAdjacentPartitions(φ_i)
        ignored ← ignored ∪ adjacent
    end if
end for
return R
```

In our example, Algorithm 3 starts with partition φ_1 containing object *a*. Since *a* dominates a complete θ interval, its successor partition is marked as ignored, as shown in Figure 3.

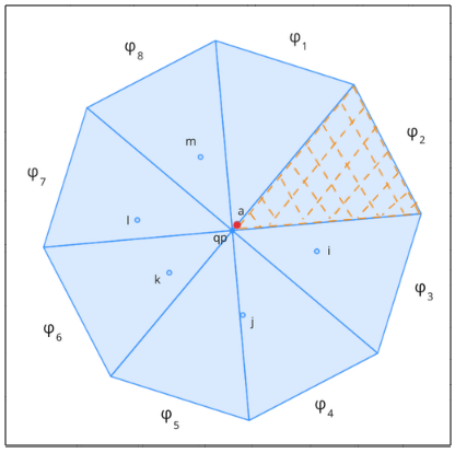

Figure 3. Ignored partition in the first iteration of the refinement step.

In the next iteration, the algorithm examines the next NN object that is not in an ignored partition. In this case, it is the point *k* in φ_6, which then marks its predecessor and successor partitions as ignored, as illustrated in Figure 4.

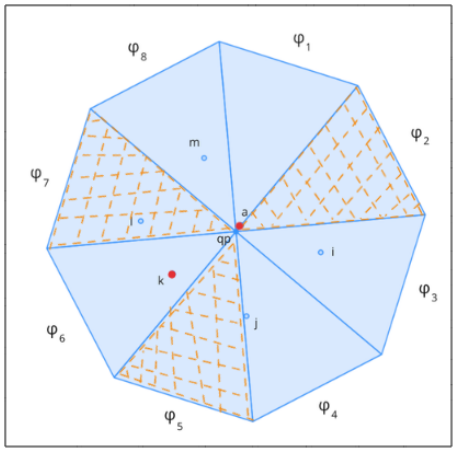

Figure 4. Adjacent partitions of φ_6 marked as ignored.

Subsequently, the object *m* in φ_8 does not mark any partitions as ignored, since its predecessor φ_7 was already ignored by φ_6. Being the last partition, φ_8 has no successor according to the definition of the adjacent partition. Finally, object *j* in φ_4 is verified and marks φ_3 as an ignored partition, as shown in Figure 5.

Definition 4 (Dominant Partition): Partitions containing the nearest object to *q* in an ordered processing list, not marked as ignored. These partitions contain objects for the DPS query result set.

Definition 5 (Composite Partition): A composite partition (PC) joins two consecutive partitions where k ranges from 1 to $\frac{n}{2}$:

$$\mathrm{PC}_k = (\varphi_{2k-1}, \varphi_{2k}) \tag{1}$$

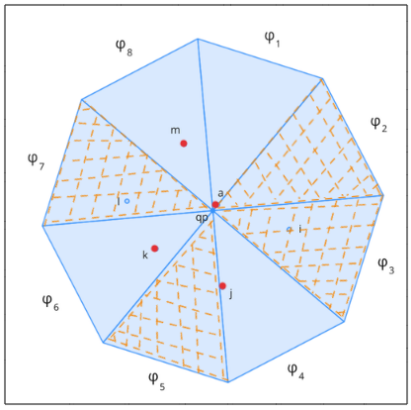

Figure 5. Final iteration of the refinement step.

From the upper limit (*ls*) of partition φ_1, we define the angular intervals (λ) for the composite partitions PC_k. For PC_1, the upper limit is as follows:

$$\mathrm{ls}_{\mathrm{PC}_1} = \overrightarrow{q\mathrm{NN}'} + \frac{\theta}{2} \tag{2}$$

The lower limit is calculated by subtracting θ from the upper limit:

$$\mathrm{li}_{\mathrm{PC}_1} = \overrightarrow{q\mathrm{ls}} - \theta \tag{3}$$

For subsequent composite partitions PCk (k > 1), the angular interval is calculated from the lower limit of the previous partition:

$$\lambda_{\mathrm{PC}_k} = \overrightarrow{q\mathrm{li}_{\mathrm{PC}_{k-1}}} - \theta \tag{4}$$

The final result set contains all NN objects from non-ignored partitions. Each object dominates the θ interval defined by a composite partition. Figure 6 shows the result set and identifies the composite partitions (PC), formed by consecutive partitions: PC_1 = φ_1 and φ_2, PC_2 = φ_3 and φ_4, PC_3 = φ_5 and φ_6, PC_4 = φ_7 and φ_8.

The refinement algorithm transforms the processing results into an ordered list, determines the dominant partitions, and combines them into composite partitions, ensuring that each object in PC_k is dominant over a complete interval θ.

IV. Experimental Evaluation

A. Datasets

To vary the distribution and complexity of spatial objects, synthetic and real datasets were constructed for the experiments.

The Spider spatial data generator [9] was used to generate synthetic data within the [0,1] interval, containing different volumes and distributions. The generated volumes were defined into three distinct categories: small, medium, and large, containing 20,000, 200,000, and 2,000,000 records, respectively.

The data distribution was generated considering the 5 types of distributions available for point objects in the generator: uniform, diagonal, Gaussian, Sierpinski, and bit. A dataset was

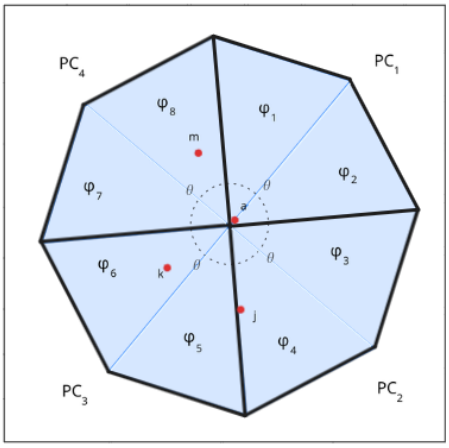

Figure 6. DPS query result with answer objects in their respective dominance intervals.

generated for each combination of volume and distribution, totaling 15 datasets.

Real-world data was collected from the OpenStreetMap platform [10], resulting in three datasets extracted from the Brazil map: a small dataset with points representing schools, a medium dataset with street intersections, and a large dataset with all point-type objects. These datasets vary in volume, representing small, medium, and large datasets.

B. Experimental Design

The experimental design was structured to comprehensively evaluate the algorithm performance under various conditions by systematically varying the query parameters. The primary parameter, θ, was tested using four distinct values: 20, 45, 60, and 90 degrees, applied consistently across all databases. Although most queries shared the same input parameters, the proposed DPS query required an additional parameter, *distMax*, which defines the maximum partition length in meters. This parameter was adjusted between real and synthetic databases to account for differences in data variation.

The experiment encompassed a total of 18 databases, 15 synthetic and 3 real. Each database was tested against four values of θ, resulting in 72 unique query scenarios. These scenarios were then doubled to compare performance between indexed and non-indexed databases, creating 144 distinct test configurations. Each configuration was evaluated using three different algorithms (DBS, DNN and DPS), culminating in 432 total test loads. Of these, 360 test loads were executed on synthetic data, while the remaining 72 were performed on real data.

C. Experimental Setup

The experiments were conducted on a physical machine with the following specifications: Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz with 12 cores, 16 GB of RAM, 1 TB SSD, running Ubuntu 20.04.1 LTS (64-bit). The spatial database was implemented using PostgreSQL 12.4 with PostGIS 3.0.2 extension.

For indexed experiments, we employed the Generalized Search Tree (GiST) indexing method provided by PostGIS, which implements a variant of the R-Tree structure. All queries (DBS, DNN, and DPS) were executed systematically, with results stored in a dedicated *results* table. To ensure consistency and prevent caching effects, the system cache was cleared before each test execution using standard Linux cache clearing procedures.

D. Performance Analysis

Although statistical tests were not performed, the performance differences are substantial enough to demonstrate DPS superiority. DPS completed some queries on large datasets in under 25 seconds, while DBS and DNN were unable to complete the same queries even after 24 hours—representing a performance improvement of at least 3,456x. Such extreme differences, consistent across multiple configurations, clearly indicate algorithmic advantages beyond measurement uncertainties.

DPS query demonstrated superior efficiency in a significant portion of the test scenarios, outperforming other methods in 52.8% of cases for non-indexed databases and 61.1% of cases for indexed databases, as illustrated in Figure 7. Specifically, it achieved better performance in 38 out of 72 query scenarios for non-indexed databases and 44 out of 72 scenarios for indexed databases, indicating robust performance across both database types.

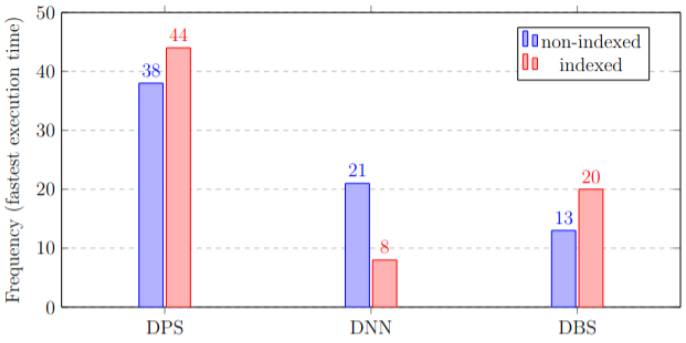

Figure 7. Frequency of algorithms (DPS, DBS, DNN) achieving fastest query execution across indexed and non-indexed databases.

Figure 8 presents a detailed breakdown of the results by data distribution, revealing significant performance patterns. The DPS algorithm demonstrated remarkable effectiveness on both synthetic and real datasets. In Bit and Sierpinski distributions, it consistently achieved optimal performance across all configurations, with a maximum frequency of 12 best results in both indexed and non-indexed scenarios. For real data, the algorithm also showed strong performance, achieving 9 and 10 best results in non-indexed and indexed configurations respectively. Although performance was more modest with the Gaussian distribution, the algorithm still maintained consistent effectiveness across diagonal and uniform distributions, demonstrating its versatility across different data patterns.

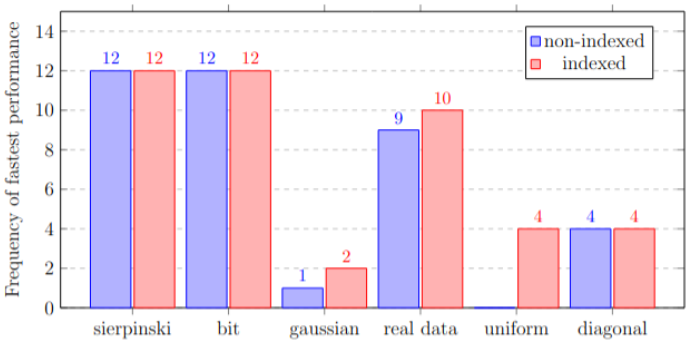

Figure 8. Frequency of DPS achieving fastest query execution across data distributions.

A deeper analysis of query execution times for Bit and Sierpinski distributions revealed significant differences among the algorithms. Both DBS and DNN algorithms encountered considerable challenges, particularly when processing large databases. In most cases involving large-scale datasets, these algorithms failed to complete execution even after 24 hours of processing time. The only exception occurred with the Sierpinski distribution, where both DBS and DNN converged to a solution in approximately 77 minutes using a θ parameter of 90°. In contrast, the DPS algorithm demonstrated remarkable efficiency. For angles of 20°, the execution time remained under 25 seconds, and for larger angles, it further decreased to less than 11 seconds. This performance improvement highlights the algorithm's scalability and optimization capabilities. To better illustrate this performance contrast, Figure 9 presents the execution time in seconds for the DPS algorithm. The graph shows results for both Bit and Sierpinski distributions in large-scale databases, comparing different values of the θ parameter across indexed and non-indexed databases.

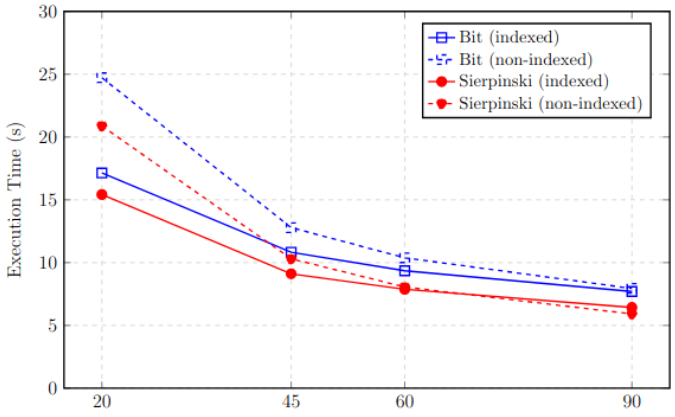

Figure 9. DPS algorithm execution time as a function of θ for larges indexed and non-indexed databases.

In the analysis of real-world data distributions, the DPS query demonstrated superior performance across most tested scenarios. The algorithm showed less favorable results for 90° angles in both indexed and non-indexed large-volume databases, as well as for 60° angles in smaller non-indexed databases. Despite these exceptions, DPS achieved excellent execution time results. Table II presents the execution times for different configurations of DBS, DNN, and DPS queries on real databases.

TABLE II
EXECUTION TIME COMPARISON BETWEEN DBS, DNN AND DPS ALGORITHMS.

Database Size	Angle	Non-Indexed			Indexed		
		DBS	DNN	DPS	DBS	DNN	DPS
Small	20°	4.37	4.36	**3.14**	4.19	4.22	**1.26**
	45°	2.18	2.13	**1.59**	2.00	2.05	**0.80**
	60°	1.25	**1.24**	1.34	1.13	1.14	**0.78**
	90°	1.28	1.18	**0.87**	1.11	1.19	**0.66**
Medium	20°	187.87	183.04	**4.51**	177.45	182.37	**5.05**
	45°	21.84	21.81	**2.39**	21.30	22.16	**2.51**
	60°	6.04	6.05	**1.92**	5.92	6.01	**2.09**
	90°	**0.78**	**0.76**	1.50	**0.71**	**0.71**	1.63
Large	20°	93.44	98.22	**37.41**	91.63	98.58	**32.13**
	45°	74.59	73.90	**19.06**	72.57	73.02	**19.44**
	60°	33.97	33.85	**15.27**	32.80	32.50	**16.75**
	90°	4.22	**3.97**	11.61	3.52	**3.49**	13.49

The analysis of the variation of the query angle, shown in Figure 10, demonstrates that DPS achieved better performance with smaller angles, particularly at $\theta = 20$. From a total of 18 queries per angle (15 on synthetic datasets and 3 on real datasets), the algorithm achieved the best 14 results on indexed bases (77.78%) and 12 on non-indexed bases (66.7%) for $\theta = 20$.

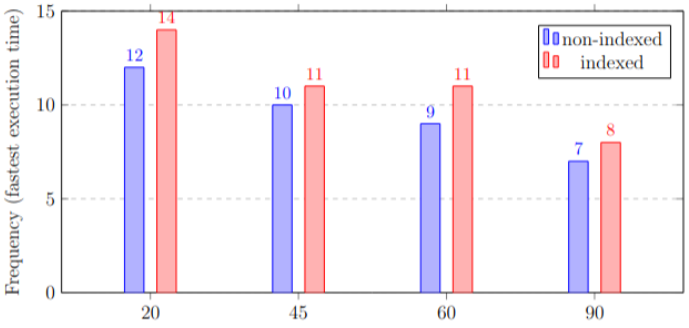

Figure 10. DPS performance comparison at different angles with and without index.

The diversity of objects returned by the DPS query reinforces this work's objective of providing consistent and homogeneous diversity in the response set, regardless of query parameters, distribution, and volume. This characteristic is demonstrated in 11, which presents a comparison of the number of objects returned in the response set among DPS, DBS, and DNN algorithms, considering only different configurations of real databases. As explained in the refinement step, if all partitions contain data, the maximum number of objects found is equal to the number of composite partitions; that is, it has a maximum of $\theta/360$ response objects.

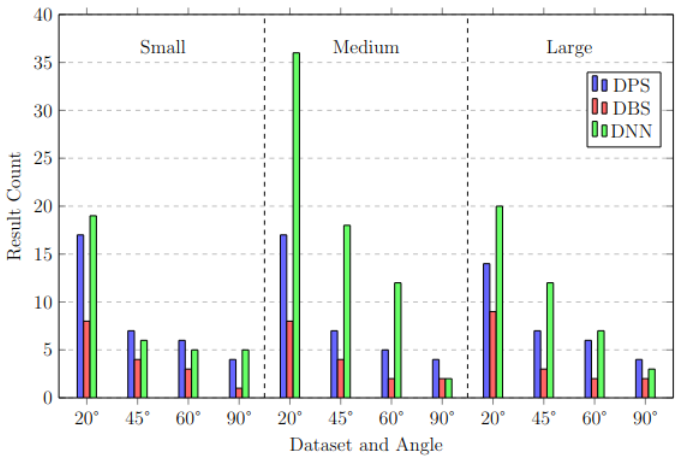

Figure 11. Comparative analysis of retrieved object counts for DPS, DBS, and DNN on real data.

V. Results Discussion

Overall, the DPS algorithm demonstrated superior performance compared to DBS and DNN algorithms, excelling in 52.8% of queries on non-indexed databases and 61.1% on indexed databases. These results demonstrate its versatility and efficiency in different application contexts.

In synthetic databases with Bit and Sierpinski distributions, DPS achieved exceptional performance, showing a 99.9% improvement in execution time for all queries on large databases. This result can be attributed to the high density of objects concentrated in specific directions, a characteristic that benefits the algorithm's geometric partitioning approach. The processing step efficiently identifies the NN point in each partition, significantly reducing the number of comparisons needed to determine the result set.

The same pattern of high object density in specific directions was observed in real-world data. This characteristic of spatial distribution explains the algorithm's excellent performance on real databases, as geometric partitioning proves to be particularly efficient when objects are concentrated in specific directions.

DPS showed better performance with smaller angle parameters, such as 20° and 45°. This behavior can be explained by the fact that smaller angles impose less strict dominance restrictions for DBS and DNN queries, meaning more objects must be evaluated before the stopping condition is reached.

Regarding the diversity of results, DPS consistently maintains that the maximum number of returned objects will be equal to $360°/\theta$, which means that there will be at most one dominant object for each θ interval.

VI. Conclusion and Future Work

This paper presented DPS, a geometric partitioning approach for direction-based queries. DPS reduces execution time by up to 99.9% compared to existing methods—completing queries in under 25 seconds that previously took over 24 hours. It also ensures directional diversity by returning at most one object per θ interval.

While our experiments focused on geographic datasets, DPS has potential applications beyond traditional GIS systems. The algorithm's ability to efficiently identify directionally diverse neighbors could benefit autonomous navigation systems when detecting surrounding obstacles, or assist IoT networks in selecting well-distributed sensor nodes. The consistent directional coverage guaranteed by DPS makes it particularly suitable for emergency response scenarios where alternative routes in different directions are critical.

Our evaluation was limited to datasets of up to 2 million points. Although DPS performed well at this scale, real-world applications with larger datasets may present additional challenges requiring further investigation.

Future research directions include:

- **Intelligent Query Selection:** Develop models to automatically choose between DPS, DBS, or DNN based on dataset characteristics and query parameters.
- **Scalability Analysis:** Evaluate DPS performance with datasets exceeding 10 million objects and identify optimization opportunities.
- **Dynamic Environments:** Adapt DPS for scenarios with frequently changing data, such as real-time traffic or mobile sensor networks.
- **Extended Domains:** Explore applications beyond spatial queries, including similarity searches in high-dimensional spaces.

These directions will help establish the practical scope and limitations of the DPS approach.

References

[1] K. C. K. Lee, W. C. Lee, and H. V. Leong, "Nearest Surrounder Queries", in *Proceedings of the 22nd International Conference on Data Engineering (ICDE)*, Atlanta, GA, USA: IEEE, Apr. 2006, pp. 85–85. DOI: 10.1109/ICDE.2006.104.

[2] X. Guo, B. Zheng, Y. Ishikawa, and Y. Gao, "Direction-based surrounder queries for mobile recommendations", *The VLDB Journal*, vol. 20, no. 5, pp. 743–766, 2011. DOI: 10.1007/s00778-011-0241-y.

[3] X. Guo and X. Yang, "Direction-aware nearest neighbor query", *IEEE Access*, vol. 7, pp. 30 285–30 301, 2019. DOI: 10.1109/ACCESS.2019.2902130.

[4] H. Zhang *et al.*, "Group Visible Nearest Surrounder Query in Obstacle Space", in *Proceedings of the 2019 IEEE International Conference on Computer Science and Educational Informatization (CSEI)*, Guangzhou, China: IEEE, 2019, pp. 345–350. DOI: 10.1109/CSEI47661.2019.8939019.

[5] J. Chung, H. J. Jang, K. H. Jung, and S. Y. Jung, "Nearest surrounder searching in mobile computing environments", *International Journal of Communication Systems*, vol. 26, no. 6, pp. 770–791, 2013. DOI: 10.1002/dac.2409.

[6] S. Nutanong, E. Tanin, and R. Zhang, "Visible Nearest Neighbor Queries", in *Proceedings of the 11th International Conference on Database Systems for Advanced Applications (DASFAA)*, Bangkok, Thailand: Springer, 2007, pp. 876–883. DOI: 10.1007/978-3-540-71703-4_73.

[7] A. C. Carniel, “Spatial Information Retrieval in Digital Ecosystems: A Comprehensive Survey”, in *Proceedings of the 12th International Conference on Management of Digital EcoSystems (MEDES ’20)*, New York, NY, USA: ACM, 2020, pp. 10–17. DOI: 10.1145/3415958.3433038.

[8] A. C. Carniel, “Defining and designing spatial queries: the role of spatial relationships”, *Geo-spatial Information Science*, vol. 26, no. 1, pp. 1–25, 2023. DOI: 10.1080/10095020.2022.2163924.

[9] P. Katiyar, T. Vu, S. Migliorini, A. Belussi, and A. Eldawy, “SpiderWeb: A Spatial Data Generator on the Web”, in *Proceedings of the 28th International Conference on Advances in Geographic Information Systems (SIGSPATIAL ’20)*, Seattle, WA, USA: ACM, Nov. 2020, pp. 465–468. DOI: 10.1145/3397536.3422351.

[10] OpenStreetMap contributors, *Planet dump retrieved from https://planet.osm.org*, https://www.openstreetmap.org, 2017.