

Combining Multiple Interests in Decision Support Queries

GUO Xi

“Sometimes we are confronted with more data than we can really use, and it may be wisest to forget and to destroy most of it.”

Donald Knuth, *The Art of Computer Programming*

Abstract

Decision support queries can recommend a handful of appropriate objects to a user to help in decision-makings. The user specifies multiple interests as the query and the system outputs objects that may be interesting to the user as the answer. Among the objects recommended, the user can choose favorites. Typically, there are two types of queries to recommend objects: top- k queries and skyline queries.

Top- k queries retrieve objects based on their scores. The scores are used to evaluate the usefulness of the objects. The scoring function is defined by the user by combining multiple interests. Sorting the objects by their scores, the system outputs the k objects at the top of the ranking list as the answer. However, asking users to define scoring functions is not reasonable because users may not be experts for defining the functions. This is one of the main disadvantages of top- k queries.

To overcome this disadvantage, skyline queries were proposed. *Skyline queries* can recommend objects without relying on a scoring function. They use the notion of *dominance*; an object dominates another object if it is better in at least one attribute and not worse in any attributes. Given a set of objects with multiple attributes, an object would not be recommended if it is dominated by some other objects.

Generally speaking, the studies in this thesis fall into the skyline query category. In this thesis, two types of queries are proposed and investigated: direction-based surround (DBS) queries for spatial datasets and combination skyline queries for multi-attribute datasets.

A *DBS query* can be applied to the location-based services that recommend spatial objects to users. In the conventional location-based services, the most popular recommendation method is to select the nearest objects of the user. For spatial objects, however, not only their distances but also their directions are important. Motivated by this idea, a DBS query retrieves the nearest objects around the user from all directions.

In this thesis, two types of DBS queries are defined in terms of the two-dimensional Euclidean space and road networks. In the Euclidean space, we consider two objects a and b to be in the same direction with respect to the user's position q if their included angle $\angle aqb$ is bounded by a threshold specified by the user at the query time. In a road network, we consider two objects a and b to be in the same direction if their shortest paths to q overlap. We say object a dominates object b if they are in the same direction and a is closer to q than b . All objects that are not dominated by others, based on this dominance relationship, constitute the DBSs. The non-dominated objects found in the

Euclidean space are called \mathbb{E} -DBSs, while the ones found in a road network are called \mathbb{R} -DBSs.

In the thesis, DBS queries are studied in both snapshot and continuous settings, and extensive experiments are performed using both real and synthetic datasets to evaluate the proposed algorithms. A snapshot query finds the DBSs according to the current position of a user. A continuous query updates the DBSs when the user is moving. In order to answer snapshot queries in Euclidean spaces, two properties are observed to reduce the search space. For snapshot queries in road networks, we calculate the shortest path of every object and then determine whether it is dominated or not. Answering continuous queries is more difficult than answering snapshot queries because both the distances and the directions are changing due to the movement of the user. The basic idea of the algorithm is that we update the DBSs only at change moments rather than updating DBSs at every moment. The change moments can be predicated when the user starts to move. Experimental results demonstrate that the proposed algorithms can answer DBS queries efficiently.

The second study is for the combination skyline queries. A *combination skyline query* retrieves fixed-size combinations of objects that are not dominated by any other possible combinations. These selected combinations are called skyline combinations. Combination skyline queries are applicable to many application scenarios such as selecting desired stock portfolios. Answering such queries is technically challenging because the traditional skyline approaches work well only when objects are handled individually. In other words, they cannot handle an exponential number of combinations efficiently.

From the observations, it is shown that the number of skyline combinations is far smaller than that of the combinations that can be enumerated. The proposed pattern-based pruning (PBP) algorithm can retrieve the skyline combinations without enumerating and checking all of the combinations. Using an R-tree for indexing objects, the algorithm can prune the candidates efficiently. The solution is based on object-selecting patterns that indicate the number of objects to be selected from each minimum bounding rectangle (MBR) in the R-tree. Two major pruning conditions are proposed to avoid unnecessary expansions and enumerations, as well as a technique to reduce space consumption on storing the skyline for each rule in the object-selecting pattern. The efficiency of the proposed algorithms is demonstrated by the extensive experiments on both real and synthetic datasets.

Acknowledgements

First of all, I would like to express my deep gratitude to my supervisor, Prof. Yoshiharu Ishikawa. During my Ph.D. studies, he gave me valuable academic guidance and constant encouragement. His patience, knowledge, honesty, and diligence impressed me greatly.

I would also like to thank the rest of my thesis committee: Prof. Toyohide Watanabe, Prof. Kenji Mase, and Prof. Takami Yasuda, for their careful reading, useful comments, and fruitful discussions.

My sincere thanks also go to Prof. Baihua Zheng, Prof. Yunjun Gao, and Dr. Chuan Xiao, co-authors of papers I published during my Ph.D. studies, for their constructive advice and thoughtful discussions.

I would like to take this opportunity to express my gratitude to Prof. Hiroyuki Kitagawa, Prof. Cyrus Shahabi, Dr. Xing Xie, Prof. Wang-Chien Lee, and Prof. Bin Wang, for their unceasing encouragement and support. In particular, I am grateful to Prof. Xiaochun Yang for introducing me to research.

I am very thankful to the other current and past members of our laboratory at Nagoya University. I'll always remember their kindness and friendship.

Last but not least, I would like to thank my parents, for raising me and supporting me spiritually throughout my life. I would like to thank my friends who make my life merry and full of happiness.

Contents

Abstract	ii
Acknowledgements	iv
List of Figures	vi
List of Tables	vii
Abbreviations	viii
Symbols	ix
1 Introduction	1
1.1 Research Background	1
1.2 Research Objectives and Contributions	3
1.3 Related Work	8
1.4 Thesis Organization	10
2 Direction-Based Surrounder Queries in the Euclidean Space	11
2.1 Motivation	11
2.2 Problem	12
2.2.1 Snapshot \mathbb{E} -DBS Queries	12
2.2.2 Continuous \mathbb{E} -DBS Queries	14
2.3 Related Work	15
2.3.1 Direction-Based k NN Queries	15
2.3.2 Location-Based Skyline Queries	17
2.3.3 Quoted Work	20
2.3.4 Summary	20
2.4 Preliminaries	20
2.4.1 Dominance Relationship and \mathbb{E} -DBS Query	21
2.4.2 Directional Closeness	21
2.4.3 Two Minor Issues	22
2.5 Processing of Snapshot Queries	23
2.5.1 First Observation: Search Space Pruning	23
2.5.2 Second Observation: Early Termination	25

2.5.3	Algorithm	26
2.6	Processing of Continuous Queries	28
2.6.1	Basic Idea	30
2.6.2	Finding Adjacent Objects	34
2.6.3	Checking Dominance	38
2.6.4	Checking Termination Condition	39
2.7	Experiments	45
2.7.1	Settings	45
2.7.2	Performances of Snapshot Queries	46
2.7.3	Performances of Continuous Queries	48
3	Direction-Based Surrounder Queries in Road Networks	54
3.1	Problem	54
3.1.1	Snapshot \mathbb{R} -DBS Queries	54
3.1.2	Continuous \mathbb{R} -DBS Queries	55
3.2	Related Work	56
3.2.1	k NN Queries in Road Networks	56
3.2.2	Path Nearest Neighbor Query	57
3.2.3	Quoted Work	57
3.2.4	Summary	58
3.3	Preliminaries	58
3.3.1	Dominance Relationship and \mathbb{R} -DBS Query	58
3.3.2	Directional Closeness	59
3.4	Processing of Snapshot Queries	60
3.4.1	Property	60
3.4.2	Naïve Algorithm	60
3.4.3	Optimized Algorithm	61
3.5	Processing of Continuous Queries	63
3.5.1	Basic Idea	64
3.5.2	Properties	65
3.5.3	Algorithms	67
3.5.4	Discussion	68
3.6	Experiments	69
3.6.1	Settings	69
3.6.2	Performances of Snapshot Queries	70
3.6.3	Performances of Continuous Queries	72
3.6.4	Screenshots	73
4	Combination Skyline Queries	76
4.1	Motivation	76
4.2	Problem	77
4.3	Related Work	78
4.3.1	Combination Skyline Queries	78
4.3.2	Other Combination Queries	78
4.3.3	Quoted Work	80
4.3.4	Summary	80
4.4	Preliminaries	81

4.5	PBP Algorithm	82
4.5.1	Object-Selecting Pattern	82
4.5.2	Basic PBP Algorithm	85
4.6	Optimized PBP Algorithm	86
4.6.1	Pattern-Pattern Pruning	87
4.6.2	Pattern-Combination Pruning	88
4.6.3	Pattern Expansion Reduction	89
4.6.4	Complete Algorithm	93
4.7	Variations of PBP Algorithm	93
4.7.1	Incremental Combination Skyline	93
4.7.2	Constrained Combination Skyline	95
4.8	Experiments	97
4.8.1	Settings	97
4.8.2	Experiments on Synthetic Datasets	98
4.8.3	Experiments on Real Datasets	105
4.8.4	Summary	107
5	Conclusions and Future Work	110
5.1	Conclusions	110
5.1.1	DBS Queries	111
5.1.2	Combination Skyline Queries	112
5.2	Future Work	113
5.2.1	DBS Queries	113
5.2.2	Combination Skyline Queries	113
A	Appendix for DBS Queries	115
A.1	Processing of Continuous k NN Queries	115
A.2	Details of Dominance Checking	117
A.3	Details of Termination Checking	119
A.3.1	Function G	120
A.3.2	Function H	121
A.4	Proofs of Property 8 and Property 9	123
	Bibliography	124

List of Figures

1.1	Decision support query	2
1.2	An example of DBS queries	4
1.3	An example of combination skyline queries	6
1.4	The position of our work	9
2.1	Motivating example of DBS queries	12
2.2	Example of an \mathbb{E} -DBS query ($\theta = \pi/3$)	13
2.3	Example of a continuous \mathbb{E} -DBS query ($\theta = \pi/3$)	14
2.4	Visible nearest neighbor query	15
2.5	Suppositive obstacles of \mathbb{E} -DBS queries	16
2.6	Nearest surrounder query	17
2.7	Location-based skyline query	18
2.8	Two observations	25
2.9	Processing of snapshot DBS query ($\theta = \pi/3$)	26
2.10	Example of a continuous \mathbb{E} -DBS query	29
2.11	Process tree for continuous DBS query	31
2.12	Change of direction order	35
2.13	Change of dominance relationship	38
2.14	Case A: $a = (2, 3)', b = (1, 4)'$	42
2.15	Case B: $a = (1, 3)', b = (4, 1)'$	42
2.16	Case C: $a = (1, 3)', b = (3, 5)'$	43
2.17	Performance of snapshot queries w.r.t. θ	46
2.18	Screenshot of snapshot queries	47
2.19	Performance of snapshot queries for data sets with different distributions	49
2.20	Number of change moments of continuous queries w.r.t. θ	50
2.21	Tree sizes and CPU costs of continuous queries vs. θ w.r.t. time interval $[0, 30]$	51
2.22	Tree depths of continuous queries	52
2.23	Performance of continuous queries for data sets with different distributions	53
3.1	Example of an \mathbb{R} -DBS query	55
3.2	Example of a continuous \mathbb{R} -DBS query	55
3.3	Path nearest neighbour query	57
3.4	Naïve algorithm for snapshot \mathbb{R} -DBS queries	61
3.5	Optimized algorithm for snapshot \mathbb{R} -DBS queries	62
3.6	Example of a continuous \mathbb{R} -DBS query	64
3.7	Property 7 of continuous \mathbb{R} -DBS queries	66
3.8	Properties 8 and 9 of continuous \mathbb{R} -DBS queries	67

3.9	The special case for continuous \mathbb{R} -DBS queries	69
3.10	Performance of snapshot \mathbb{R} -DBS queries w.r.t. the number of objects . . .	70
3.11	Performance of snapshot \mathbb{R} -DBS queries w.r.t. the object category	71
3.12	Performance of continuous \mathbb{R} -DBS queries w.r.t. the number of objects . .	72
3.13	Performance of continuous \mathbb{R} -DBS queries w.r.t. the object category . . .	73
3.14	Screenshot of a snapshot \mathbb{R} -DBS query	74
3.15	Screenshots of a continuous \mathbb{R} -DBS query	75
4.1	Object layout and R-tree	83
4.2	Pattern tree	85
4.3	Pattern-pattern pruning (grey patterns are pruned using Theorem 1) . . .	87
4.4	Priority queue and query result	89
4.5	Pattern-combination pruning (grey patterns are pruned using Theorem 2 and the patterns beginning with \times are pruned using both Theorem 1 and Theorem 2)	90
4.6	Pattern expansion reduction matrix	91
4.7	Incremental combination skyline query	94
4.8	Constrained combination skyline	96
4.9	Constraint-based pruning	96
4.10	Distribution of combinations and skyline combinations	99
4.11	PBP versus BBS on small datasets	99
4.12	PBP performance for different distributions	101
4.13	PBP performance for different cardinalities	102
4.14	PBP performance for different number of attributes	103
4.15	PBP performance for different fanouts of R-tree	105
4.16	PBP performance for different cardinalities on real datasets	106
4.17	PBP performance for different number of attributes on real datasets . . .	107
A.1	Candidate area for the next nearest object	116

List of Tables

2.1	Related work of \mathbb{E} -DBS queries	20
2.2	Incremental maintenance of direction order lists	36
2.3	Decision table for case A	43
2.4	Decision table for case B	43
2.5	Decision table for case C	44
2.6	Datasets	45
3.1	Related work of \mathbb{R} -DBS queries	58
4.1	Related work of combination skyline queries	81

Abbreviations

DBS	D irection- B ased S urrounder
POI	P oint O f I nterest
E-DBS	D irection- B ased S urrounder in two-dimensional E uclidean spaces
R-DBS	D irection- B ased S urrounder in R oad networks
kNN	k N earest N eighbor
VNN	V isible N earest N eighbor
NS	N earest S urrounder
CNN	C ontinuous N earest N eighbor
MOO	M ulti- O bjective O ptimization
MOCO	M ulti- O bjective C ombinatorial O ptimization
MOKP	M ulti- O bjective K napsack P roblem
CSP	C onstraint S atisfaction P roblem
MBR	M inimum B ounding R ectangle
PBP	P attern- B ased P runing

Symbols

\vec{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 \vec{p}_i in \mathbb{E} ($d_{p_i} = \vec{p}_i $) or the length of $SP(q, p_i)$ in \mathbb{R} ($d_{p_i} = SP(q, p_i) $)
\mathcal{O}	Multi-attribute object set
\mathcal{A}	Attribute set
$c = \{o_1, \dots, o_k\}$	A k -item combination
$c.A_j$	Attribute value of combination c
f_j	Monotonic aggregate function
r_i	An MBR
$obj(r_i)$	The set of objects enclosed by r_i
(r_i, k_i)	A rule
$\langle p_1, p_2, \dots, p_k \rangle$	Direction order list of the objects $\{p_1, p_2, \dots, p_k\}$
$\omega_i (\omega_{p_i})$	Direction of p_i : it is set to the angle between \vec{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
$\lambda_{ij} (\lambda_{p_i p_j})$	Included angle between p_i and p_j
$\varphi_{ij} (\varphi_{p_i p_j})$	Partition angle between p_i and p_j
$(\cdot, \cdot)'$	Transposition of the vector $(\cdot, \cdot)'$

Chapter 1

Introduction

1.1 Research Background

Nowadays, we are facing a flood of data due to the development of computer technologies. This flood brings us much more information than ever before and changes our lives even when we are not aware of it. The volume of data grows dramatically and comes in various forms, such as scientific tables, commercial records, stock charts, hypertexts, and multimedia. In the 1940s, when the first electronic digital computer appeared, we learned not only to process data efficiently, but also to store data on smaller and smaller storage disks rather than heavy stacks of books. The development of the Internet, which began in the 1960s, created another information explosion. Through the Internet, we can obtain and share data in more convenient and faster ways than reading books in libraries or writing letters for communications. Ubiquitous computing, which started in about 1988, has come into full bloom and is pushing the data flood right under our noses. Ubiquitous computing techniques make it possible to capture and collect data from almost any physical object, for example, cars, mobile phones, or even animals, by using just a small sensor chip. In 2010, a report predicted that there will be more data generated in the next four years than in all of history [1]. Can we surf this data flood and acquire more knowledge than our predecessors?

Possessing an enormous amount of data is meaningless if we cannot acquire knowledge from it. Good knowledge can help us make correct decisions and further discover truths. One way to convert data to knowledge is to extract small quantities of useful data as

choices or knowledge for decision-making. Motivated by this purpose, over the past few decades a lot of work has focused on exploiting such useful data from the data flood [2–4]. As an example, search engines such as Google can find those documents most related to the keywords from a good number of Web pages [3]. Our proposed approaches can help people in a similar way. One application scenario of our work is that we can recommend nearby drug stores from among all known drug stores. Another scenario is that we can recommend a good portfolio created from all the issued stocks. Generally speaking, we retrieve the desired data from a large-scale data set in order to help users make decisions (Fig. 1.1).

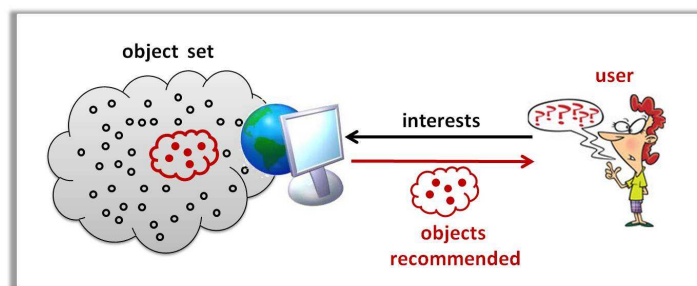


FIGURE 1.1: Decision support query

Whether data are desirable depends on the users query. Users convey their demands by submitting queries to computers. For search engines, the queries are the keywords input by users and the desirable data returned are relevant documents containing those keywords. Unlike search engines designed for exploiting text data, our first work exploits spatial data and our second exploits multi-attribute data. The spatial data, also called the points of interests (e.g., drug stores), are characterized by their two-dimensional physical coordinates. A typical query for the spatial data is “show me nearby drug stores.” The desirable data returned for the query would be several of the nearest drug stores, or those within walking distance. Most existing works focus on such spatial closeness (e.g., [5], [6], [7]); however, our research focuses on directions, which are also important features of the spatial data but have not been sufficiently studied [8, 9].

Our second work studies how to find the desired data from multi-attribute data. Multi-attribute data, that is, records or tuples, are characterized by two or more attributes. For example, stock data might have attributes including price, risk, and profit. The queries are the user’s demands pertaining to several attributes or all of the attributes. For example, “recommend me several stocks (or portfolios) with high profits but low

risks.” The desirable data returned would be those stocks (or portfolios) that can satisfy the demands regarding profits and risks. Most existing works (e.g., [10], [11]) focus on only selecting desirable records individually (e.g., stocks). Our research, however, focuses on selecting desirable combinations of records (e.g., portfolios) of which the studies are rather limited [12, 13].

1.2 Research Objectives and Contributions

The most popular query for the spatial data is the *nearest neighbor query* [5]. The nearest neighbor query is to find k nearest points of interest (POIs) given the user’s position. As an example, in Fig. 1.2, there are seven drug stores and user q . Since drug store a is the nearest one to the user, it is the answer to the nearest neighbor query with $k = 1$. Based on spatial closeness, there are many variations of the nearest neighbor query, such as *visible nearest neighbor queries* [14–16], the *nearest surrounder queries* [8, 9], and the *reverse nearest neighbor queries* [17–19]. Different from the existing works, we propose a new variation called the *direction-based surrounder (DBS) query* considering not only closeness but also direction. The directions to objects are also worth consideration. One reason is that the objects in all different directions can reflect neighborhood information. Another reason is that in some scenarios, for example, a user is driving on a one-way road, only the objects in a specific direction are useful. Example 1 illustrates the DBS query.

Example 1. The user wants to know the nearest drug stores in all directions in order to capture an overview of the surroundings. In Fig. 1.2, the arrows \vec{qa} and \vec{qb} show the directions of drug store a and drug store b with respect to user q . Since angle $\angle aqb$ is rather small, drug store a and drug store b would be regarded as being in the same direction. Comparing the two drug stores in the same direction, drug store a is more recommendable due to its closeness. Motivated by the user’s preference for closeness and direction, we would recommend the three drug stores a , d , and f . The reason is that drug store a (or d , f) is nearer than the ones $\{b, g\}$ (or $\{c\}$, $\{e\}$), located in the same direction.¹

¹Assume that the user regards two drug stores as being located in the same direction if their included angle is not larger than a threshold 60° as the dotted lines show. The threshold is a small angle that is a default setting or an angle input by the user.

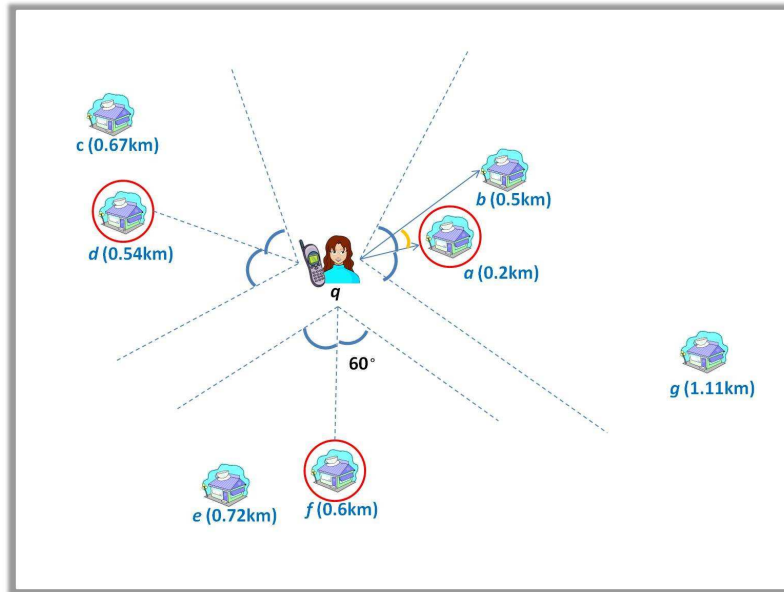


FIGURE 1.2: An example of DBS queries

Conventional nearest neighbor queries consider only the distance aspect, while DBS queries consider not only the distance aspect but also the direction aspect. Certainly, the computation cost of a DBS query is more than the cost of a nearest neighbor query, but our experimental results show that the DBS queries could be answered promptly enough. More importantly, the DBS queries have advantages over the nearest neighbor queries in two main aspects. A quite straightforward advantage is that DBS queries can provide multiple choices in all directions, instead of limited ones in some specific directions. DBS queries are especially important when the user is not willing to move in the direction of the nearest POI. Another advantage is that DBS queries can present neighborhood information around the user, as Example 1 showed. In short, our DBS queries can recommend adequate POIs when users desire to make decisions considering both distance and direction, whereas the existing queries cannot support such demands.

DBS queries are mainly applied to location-based services (LBSs) [20]. As an example, consider a tourist who wants to go sightseeing in an unfamiliar city. Using a nearest neighbor query, the tourist could find only the nearest spots that might be gathered in some specific direction. However, using a DBS query, the tourist could find all the nearest spots that lie in different directions. In this way, the tourist would obtain a full neighborhood view and have more options for sightseeing. As another example, let us consider a driver who wants to refuel. Using a nearest neighbor query, the driver could

find which gas stations are nearby, but they may not all be in an appropriate direction. An extreme case is when the gas stations suggested are all in the opposite direction of a one-way street. In contrast, using a DBS query the driver would obtain the nearest gas stations that lie in all directions. It is then possible to choose an appropriate one. Certainly, some may argue that we can issue a nearest neighbor query with a large k such that the objects returned could cover all different directions. However, this approach is not practical because it is difficult to decide an adequate k before issuing the query. Therefore, DBS queries can overcome the disadvantages of nearest neighbor queries.

Answering DBS queries seems quite simple in Example 1, but it is not trivial when there are a huge number of POIs. In practice, it is common that thousands of POIs are available to choose from. To decide which POIs should be recommended to the user, a naive approach is to check every POI by comparing it with all other POIs, but this is rather time-consuming. If we cannot answer DBS queries in a short time, users will be unsatisfied. For this reason, we propose fast algorithms to answer DBS queries by doing a very small number of comparisons.

With the development of the mobile computing technology [21], *continuous nearest neighbor queries* [6, 7, 22] along with its variations [14, 19] attract more and more attention. In such continuous queries, the user, or the POIs (e.g., cars), or both are moving. We only study DBS queries issued when the user is moving along a straight line. The challenge of answering continuous queries is that both the distances and directions of the POIs are changing due to the user's movement. For this challenge, we propose an algorithm to answer such *continuous DBS queries* by updating the result set when it changes.

Besides studying the DBS queries in two-dimensional Euclidean space, we also study the problem in road networks, about which practical spatial data is available. There are numerous works regarding nearest neighbor queries in road networks [23–25]. Like these studies, we also set the distance of a POI to the length of the shortest path from the user to the POI. Since our DBS queries also consider the direction of a POI, we set the direction to the shortest path itself. We study both the snapshot and continuous DBS queries in road networks.

For the spatial data, our contributions are summarized below.

- We propose a new query for exploiting spatial data, called a *DBS query*, which can recommend POIs considering not only spatial closeness but also direction.
- We define DBS queries both in two-dimensional Euclidean space (\mathbb{E} -DBS queries) and in road networks (\mathbb{R} -DBS queries).
- We propose efficient algorithms for answering both snapshot and continuous \mathbb{E} -DBS queries.
- We propose efficient algorithms for answering both snapshot and continuous \mathbb{R} -DBS queries.
- We conduct extensive experiments on both synthetic and real datasets to evaluate the algorithms for \mathbb{E} -DBS queries and \mathbb{R} -DBS queries.

There are two popular ways to exploit multi-attribute data: *top-k queries* and *skyline queries*. Top- k queries [11] retrieve the k records with the highest scores. The scores of the records are computed according to some aggregation function. What the aggregation function looks like depends on the user's preferences on the attributes. However, we exploit the multi-attribute data in another way, which is called a skyline query [10, 26–28].

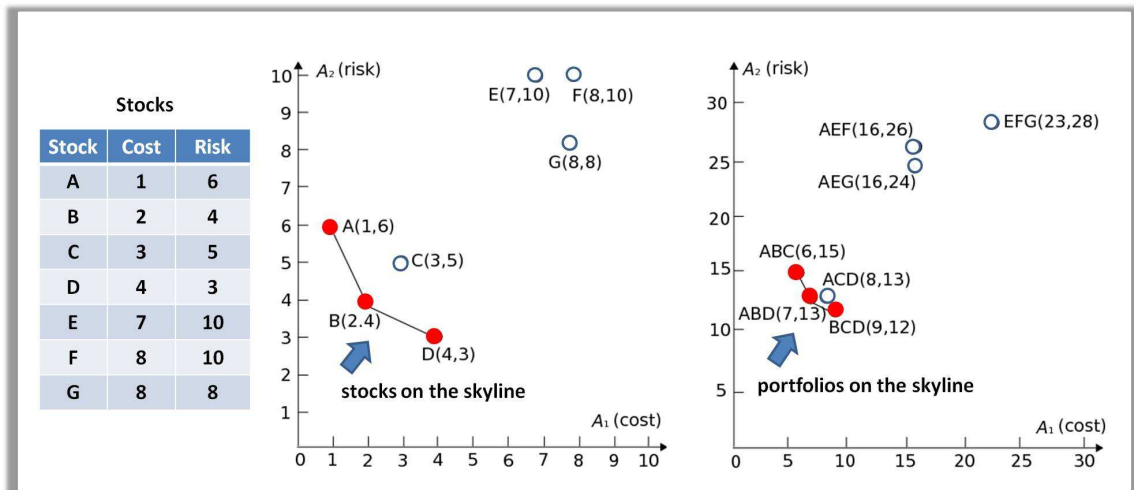


FIGURE 1.3: An example of combination skyline queries

Instead of comparing records by their scores, skyline queries compare the records by their dominance relationships. One record *dominates* another record if it is better in at least one attribute and not worse in any other attribute. A record is *on the skyline* if it cannot be dominated by any other records. Such non-dominated records are also called the skyline

records. According to the users preferences on attributes, the skyline records are selected and recommended to the user. Figure 1.3 shows that among seven stocks $\{A, \dots, G\}$, the stocks $\{A, B, D\}$ are on the skyline because they cannot be dominated by any other stocks for users who prefer low cost and low risk. On the other hand, stocks $\{C, E, F, G\}$ are not on the skyline, for example, because stock $B(2, 4)$ dominates stock $C(3, 5)$ because it is better in both of the two attributes. When users intend to make trade-off decisions considering their preferences on different attributes, they can use skyline queries to acquire all the records deserving their attention.

Skyline queries have attracted considerable interest [26–28] following the seminal paper by Börzsönyi et al. [10], but most studies focus on retrieving records individually (e.g., individual stocks). In contrast, our work retrieves combinations of records (e.g., portfolios composed of several stocks). We call this a *combination skyline query*, about which there are few related studies [12, 13]. Example 2 shows a simple query of this type.

Example 2. The user desires three-item portfolios with low risk and low cost. The risk and cost of the portfolios are the sums of their components' risks and costs. In the rightmost chart of Fig. 1.3, some three-item portfolios are shown as points in the cost-risk space.² Considering the user's preference, the portfolios $\{ABC, ABD, BCD\}$ cannot be dominated by any other portfolio, and thus will be recommended as the answers to the combination skyline query. ■

Since combinations of objects are common in practice, combination skyline queries could apply to various domains. Besides forming ideal portfolios, we can use combination skyline queries to form other good combinations that are on the skyline, for example, a basketball team consisting of the right players, an attractive gift basket consisting of appropriate gifts, and a well-balanced meal consisting of reasonable dishes. Note that in this paper we consider combinations made up of only a fixed number of objects. A combination should have the same attributes as its components. The attribute values are calculated by some monotonic aggregation function that takes the components as the inputs.

For the combination skyline query problem, the number of combinations is $\binom{|\mathcal{O}|}{k}$ for a data set containing $|\mathcal{O}|$ records when we select combinations of size k . This poses serious algorithmic challenges when compared with the traditional skyline problem. As Example 2 shows, $\binom{7}{3} = 35$ possible combinations are generated from only seven records. Even

²To make the chart concise and clear, the other points are omitted.

for a small database with thousands of entries, the number of combinations of records is prohibitively large. For this reason, we propose a pattern-based pruning (PBP) algorithm to answer the combination skyline queries without enumerating all the combinations. The PBP algorithm retrieves the skyline combinations following the object-selecting patterns. Indexing the objects using an R-tree [29], we can use an object-selecting pattern to represent the number of objects selected from each minimum bounding rectangle (MBR).

Our contributions are summarized below.

- We propose the combination skyline problem, a new variation of the skyline problem that prevalently exists in daily applications and poses technical challenges.
- We devise the PBP algorithm to tackle the major technical issue.
- We discuss two variations of the combination skyline problem, incremental combination skylines and constrained combination skylines, which can be solved by extending the PBP algorithm.
- We conduct extensive experimental evaluations both on synthetic and real data sets to demonstrate the efficiency of the proposed algorithm.

1.3 Related Work

In order to help users make decisions, typically we can recommend a handful objects in two ways. One way is to use top- k queries. The other way is to use multi-objective optimization queries (i.e., skyline queries).

In the top- k query processing research area, a common ranking method is to score the objects using a function $f : \vec{v} \mapsto \langle \vec{\alpha}, \vec{v} \rangle$, where \vec{v} is the attribute values of an object and $\vec{\alpha}$ weighs the attribute importance [11]. However, asking the users to define scoring functions is not reasonable because some users are not experts and are not good at defining functions. In light of this disadvantage of top- k queries, *skyline queries* were proposed by Börzsönyi et al. [10]. The skyline queries can retrieve all the objects that users might be interested in without asking them to input any scoring functions.

In database research, the *skyline query* has received considerable attention. It is also called the *multi-objective optimization (MOO)* problem [30, 31] in other fields. Since [10]

was published, many subsequent algorithms have been proposed to improve performance from different aspects. Well-known centralized algorithms include the branch-and-bound skyline algorithm (BBS) [28], sort-filter-skyline (SFS) [26], and linear elimination sort for skyline (LESS) [27]. Recently, distributed and parallel skyline processing algorithms (e.g., [32], [33]) have received growing interest with the improvements to mobile processing capabilities and the development of wireless networks. Many variations and extensions are derived from the classical skyline [10] with respect to different research aims, such as location-based skyline queries in spatial databases (e.g., [34], [35]). The main disadvantage of skyline queries is that the number of the results cannot be controlled, because skyline queries retrieve every object that may interest the user.

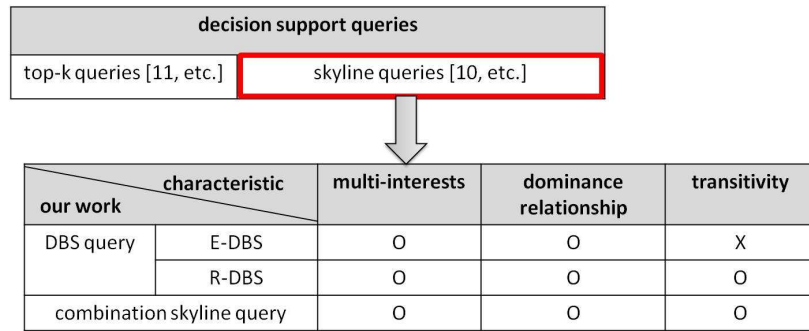


FIGURE 1.4: The position of our work

Generally speaking, our *DBS queries* and *combination skyline queries* belong to the skyline query category. The characteristics of such queries are as follows:

- The queries retrieve desired objects according to the *multiple interests* of a user in order to help make decisions.
- Objects that cannot be dominated are selected. The *dominance relationships* are determined by multiple interests.

DBS queries find POIs considering both geographical closeness and direction. Combination skyline queries find record combinations considering the users' demands regarding different attributes. Fig. 1.4 illustrates the position of our work on decision support queries. Note that the E-DBS query does not have the transitive property³, as do the other skyline queries.

³The transitive property means that if object o dominates object o' and o' dominates object o'' then o certainly dominates o'' .

1.4 Thesis Organization

The rest of the thesis is organized as follows. In Chapter 2, we introduce the DBS queries in two-dimensional Euclidean spaces. In Chapter 3, we introduce the DBS queries in road network spaces. In Chapter 4, we introduce the combination skyline queries. In Chapter 5, we summarize this work and propose several areas of future study.

Chapter 2

Direction-Based Surround Queries in the Euclidean Space

2.1 Motivation

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 [20] 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 the user wants to know the neighbourhood information.

An example is depicted in Fig. 2.1. Fig. 2.1(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 the 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 surround* (*DBS*) queries in this paper. As an example a DBS query depicted in Fig. 2.1(b) returns the three nearest objects surrounding q . 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.

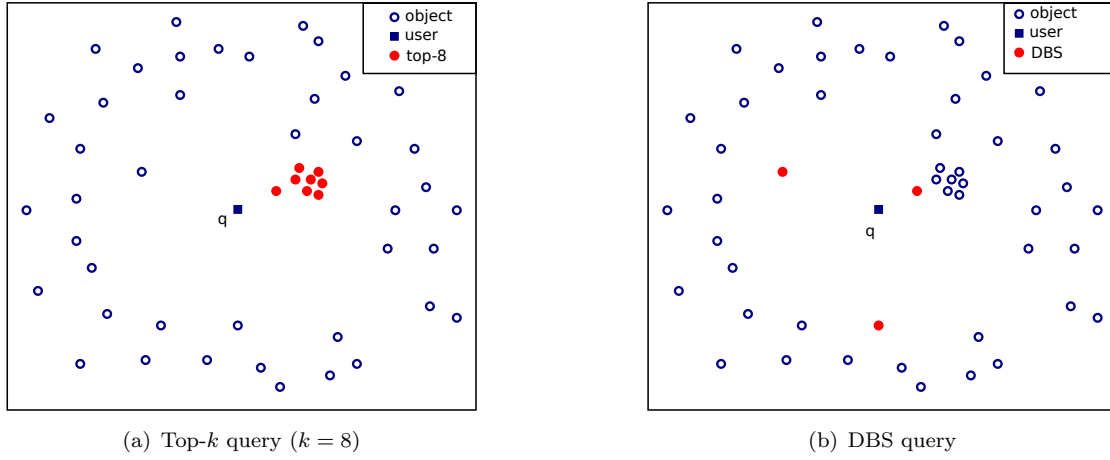


FIGURE 2.1: Motivating example of DBS queries

As shown in Fig. 2.1(b), a DBS query evaluates objects 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 .

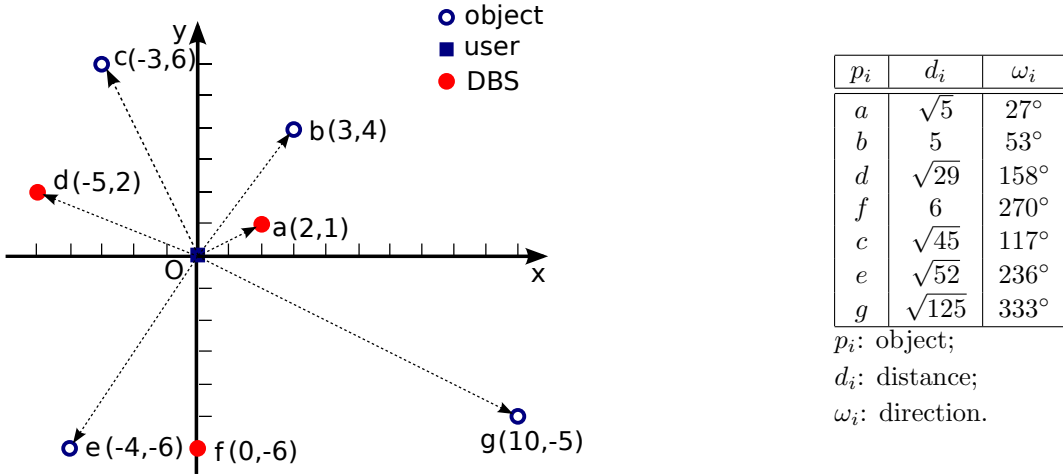
We consider DBS queries in a vector space \mathbb{E} . For the vector space \mathbb{E} , we assume that objects of interest are in the two dimensional Euclidean space and the corresponding queries are called \mathbb{E} -DBS queries. For \mathbb{E} -DBS queries, we measure the distance and the direction of an object p_i w.r.t. q using the vector which originates from q and ends in p_i .

2.2 Problem

2.2.1 Snapshot \mathbb{E} -DBS Queries

Before presenting the formal definition of a DBS query, we use Example 3 to illustrate DBS queries in the Euclidean space \mathbb{E} . It also serves as the running examples in this paper.

Example 3 (Snapshot \mathbb{E} -DBS Queries). In Fig. 2.2, there are seven POIs (a to g) around the user O . We use the vectors \vec{a}, \dots, \vec{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 $\theta (= \pi/3)$ specified by the user. For example, a and b are directional close as $|\omega_a - \omega_b| = 26^\circ < \pi/3$, but objects a

FIGURE 2.2: Example of an \mathbb{E} -DBS query ($\theta = \pi/3$)

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.

■

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 2.4. 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 3 presents examples of snapshot DBS queries in the Euclidean space \mathbb{E} . 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.

2.2.2 Continuous \mathbb{E} -DBS Queries

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 4 is extended from original Example 3 to illustrate the idea of continuous \mathbb{E} -DBS queries.

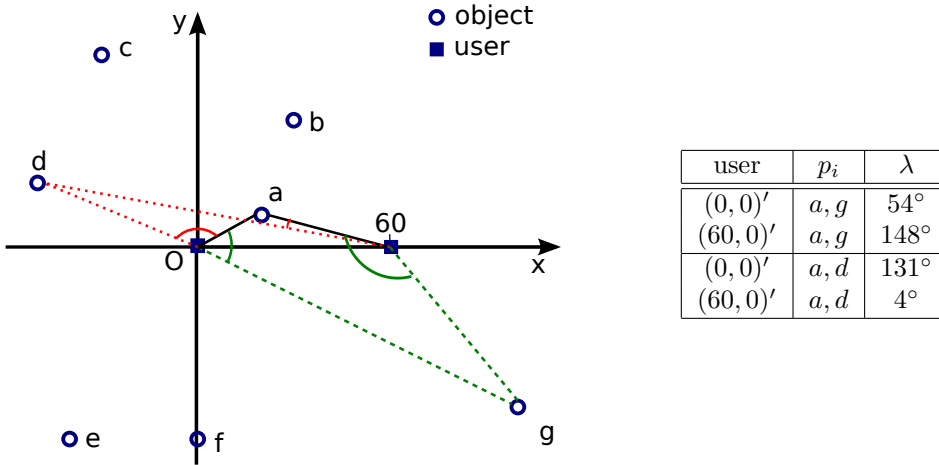


FIGURE 2.3: Example of a continuous \mathbb{E} -DBS query ($\theta = \pi/3$)

Example 4 (Continuous \mathbb{E} -DBS Queries). As shown in Fig. 2.3, 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. 2.3 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 $(0, 0)'$. ■

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 the Euclidean space \mathbb{E} , and present the corresponding query processing algorithms.

2.3 Related Work

2.3.1 Direction-Based k NN Queries

In the two-dimensional Euclidean space, most existing nearest neighbor queries focus on the geographical closeness (e.g., [5], [6], [7]), however, there are also several studies that consider the direction properties. Among them, *visible nearest neighbor (VNN)* queries [14–16] and *nearest surrounder (NS)* queries [8, 9] are most related to our DBS queries.

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 [14–16], as Fig. 2.4 shows. 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.

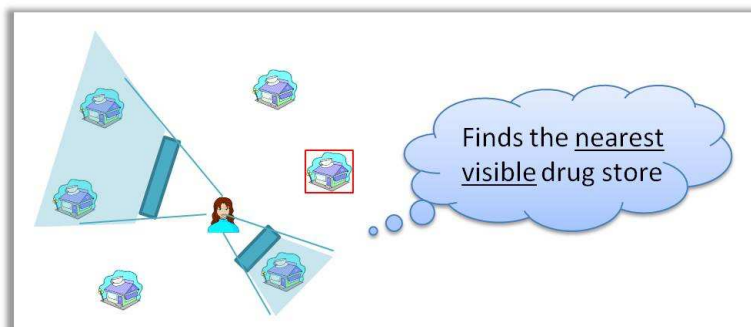


FIGURE 2.4: Visible nearest neighbor query

We can regard an \mathbb{E} -DBS query, proposed in this paper, as a special type of VNN queries. We consider each data point as an obstacle and derive its suppositive 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. 2.5(a). For point a , its dominance region is defined by the angle $\theta = \pi/3$ ¹. For the presentation purpose, assume that there is an suppositive 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 suppositive obstacle o_i in a similar manner. Thus, we can transform an \mathbb{E} -DBS query to a VNN query.

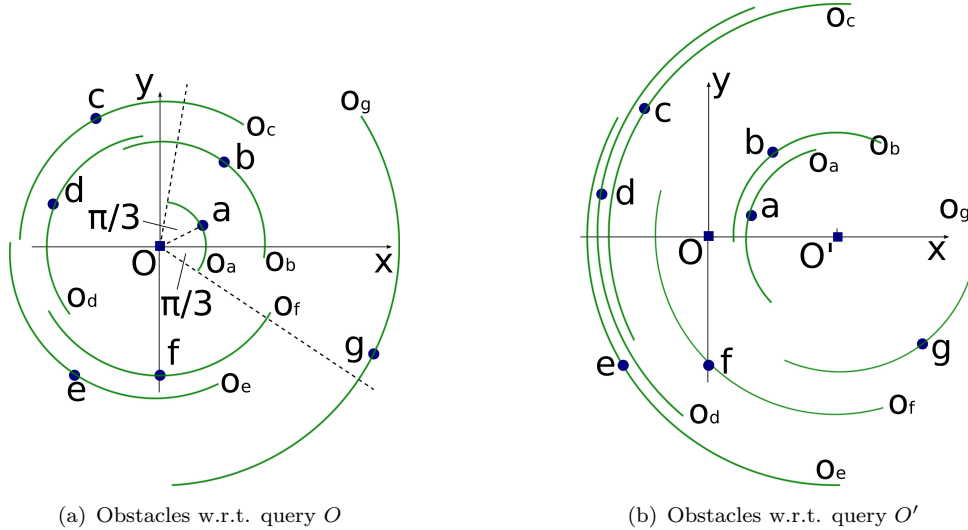


FIGURE 2.5: Suppositive obstacles of \mathbb{E} -DBS queries²

However, we cannot directly apply the algorithms proposed for VNN queries [14–16] to DBS queries. The main reason is that the existing VNN query methods consider rectangular (or polygonal) obstacles. In Section 2.5, we will explain how to exploit the properties of arcs to handle our problem efficiently. Additionally, in the continuous case, the shapes of the arc obstacles are changing due to the movement of the user. As shown in Fig. 2.5(b), the positions of suppositive 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 2.6.

Nearest Surrounder Queries. Lee et al. [8, 9] 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. 2.6 shows an example of an NS query with a query

¹Assume that the user sets the threshold θ to be $\pi/3$.

²The acronym “w.r.t.” is short for the phrase “with respect to”.

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. [8, 9] 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.

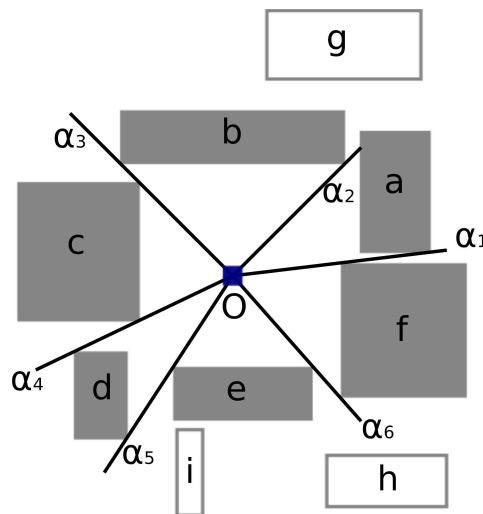


FIGURE 2.6: Nearest surrounder query

Other Direction-Based Queries. Patrumpas et al. proposed the notion of an *orientation-based query* which finds objects moving towards the query point [36]. 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 objects within the required direction and distance ranges. To the best of our knowledge, our work is the first study of direction-based surrounder queries considering distance and direction attributes.

2.3.2 Location-Based Skyline Queries

In spatial databases, where the geographical data are stored, 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. As Fig. 2.7 shows, the user wants to find hotels both near (i.e., spatial interest) and cheap (i.e., non-spatial interest). Spatial

interests 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.

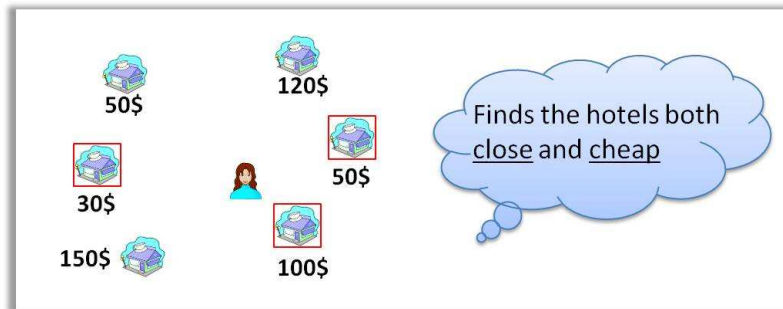


FIGURE 2.7: Location-based skyline query

Single Point. In the *single point* category, approaches [34, 35] focus on a dynamic spatial attribute (Euclidean distance) and static non-spatial attributes. For example, [37, 38] focus on both the distance information and some static non-spatial attributes, and [39] focuses on two static spatial attributes, i.e., spatial network distance and detour. Huang et al. [34] 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. [35] 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 [34]. 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 [34, 35] that consider distances only cannot be directly applied to answer DBS queries because we consider not only distances but also directions.

Single Point with Multiple Dynamic Attributes. The skyline problem becomes more complicated when we take multiple dynamic attributes into account [37, 38]. Chen et al. [37] 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 [40]. 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 [37] to update skylines while a query point moves.

Lee et al. [38] 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 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. [39] 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.

Multiple Points. In *multiple point* category, approaches [41, 42] focus on the context where there are several query points. In the Euclidean space, Sharifzadeh et al. [41] find out skyline points (e.g., meeting places) which can minimize traveling distances for several query points (e.g., a group of mobile users) and proposed algorithms B^2S^2 and VS^2 . Son

et al. [42] noticed the incompleteness of the VS^2 algorithm and proposed alternative algorithms to answer skyline queries.

2.3.3 Quoted Work

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 [7] and we also employ the basic idea of the work presented by Raptopoulou et al. in [43] to make use of the intersections of distance functions to find changes of nearest neighbors.

2.3.4 Summary

TABLE 2.1: Related work of \mathbb{E} -DBS queries

query point \ interest	spatial			non-spatial
	distance	+obstacle	+direction	+(price, etc.)
snapshot	[5], etc.	[14], etc.	[44] and [45]	[35], etc.
continuous	[7], etc.	[14], etc.	[44] and [45]	[35], etc.

Table 2.1 summarizes the related work of \mathbb{E} -DBS queries and illustrates the position of our work. The interests of the users can be divided into two categories, namely, spatial interests and non-spatial interests. According to the state of the query point, the queries can be divided into two categories, namely, the snapshot queries and the continuous queries. There are a lot of work considering only distances in both snapshot scenarios ([5], etc.) and continuous scenarios ([7], etc.). There are several papers considering not only distances but also spatial obstacles in both snapshot and continuous scenarios ([14], etc.). In addition, there are also some work considering distances as well as some non-spatial attributes in both snapshot and continuous scenarios ([35], etc.). Our study is the first one considering distances and directions in both snapshot and continuous scenarios ([44] and [45]).

2.4 Preliminaries

In this section, we formalize DBS queries. There is a set of target objects $P = \{p_1, \dots, p_n\}$ and a query object q in a two-dimensional Euclidean space \mathbb{E} . 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. We consider DBS queries in a vector space \mathbb{E} . In the vector space \mathbb{E} , we compare objects using Euclidean distances and directions.

2.4.1 Dominance Relationship and \mathbb{E} -DBS Query

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

Definition 1 (Dominance Relationship). In a two-dimensional Euclidean space \mathbb{E} , 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 $p_i \prec p_j$. \square

Note that objects that are not directional close are not comparable. We will define the *direction closeness* in the vector space \mathbb{E} (Section 2.4.2). Accordingly, *DBS queries* are defined in Definition 2.

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

2.4.2 Directional Closeness

Assume that a set of target objects $P = \{p_1, \dots, p_n\}$ and a query object q are in a two-dimensional Euclidean space \mathbb{E} . The vector \vec{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} ($= |\vec{p}_i|$) 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 \vec{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 \vec{a} in Fig. 2.2 has the distance $d_a = |\vec{a}| = \sqrt{5}$ and the direction $\omega_a = 27^\circ$.

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 \vec{p}_i and \vec{p}_j can capture their angular difference, mathematically defined as follows:

$$\lambda_{ij} = \arccos \frac{\vec{p}_i \cdot \vec{p}_j}{|\vec{p}_i| \cdot |\vec{p}_j|}. \quad (2.1)$$

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. \square

As we have shown in Example 3 (Fig. 2.2), 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 $\lambda_{ad} > \theta$.

2.4.3 Two Minor Issues

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.

2.5 Processing of Snapshot Queries

In this section, we present the snapshot \mathbb{E} -DBS query and its corresponding search algorithms. When a direction-based surrounders query is issued at a fixed query point q , it is a snapshot DBS query. As mentioned in Section 2.1, 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 $n = |P|$ where P is the set of target objects. 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 next subsections, we present some efficient search algorithms to support \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 earlier³.

2.5.1 First Observation: 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 certainly 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

³To 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.

to facilitate the checking of directional closeness, we introduce the notions of a *direction order list* and *adjacent objects* defined in Definition 4 and Definition 5, respectively.

Definition 4 (Direction Order List). Given a set of objects P' and a query point q , its *direction order list* $L_{P'} = \langle p_1, p_2, \dots, p_k \rangle$ is formed by the objects of P' 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) $|L_{P'}| = |P'|$. \square

Definition 5 (Adjacent Object). Given a set of objects $P' = \{p_1, p_2, \dots, p_k\}$ 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. \square

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. 2.2 again. Assume objects are evaluated based on ascending order of their distances to q , and those objects that have been evaluated constitute the set $P' = \{a, b, d\}$ with $L_{P'} = \langle a, b, d \rangle$, as shown in Fig. 2.8(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 certainly 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 certainly 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' , which 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. \blacksquare

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

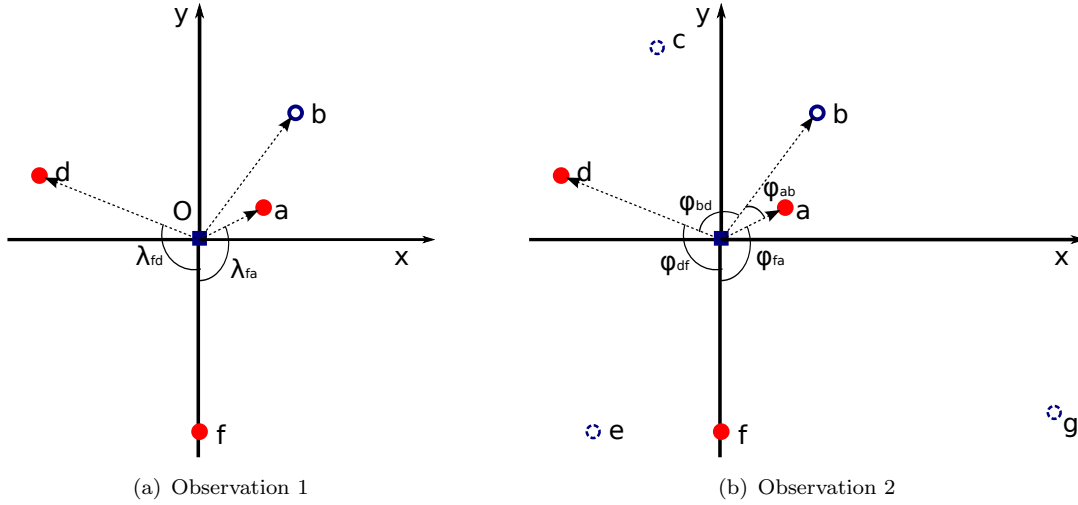


FIGURE 2.8: Two observations

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 . \square

2.5.2 Second Observation: 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 6.

Definition 6 (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. (2.2):

$$\varphi_{ij} = (\omega_j - \omega_i) \bmod 360^\circ. \quad (2.2)$$

\square

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. 2.8(b), four partition angles are formed. They are $\varphi_{ab} = 26^\circ$, $\varphi_{bd} = 105^\circ$, $\varphi_{df} = 112^\circ$, and $\varphi_{fa} = 117^\circ$. In other words, the 2π angular range is partitioned by the objects of P' into four sub-angular regions. For a new object, it certainly 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 $2\theta = 120^\circ$, 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. □

2.5.3 Algorithm

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.

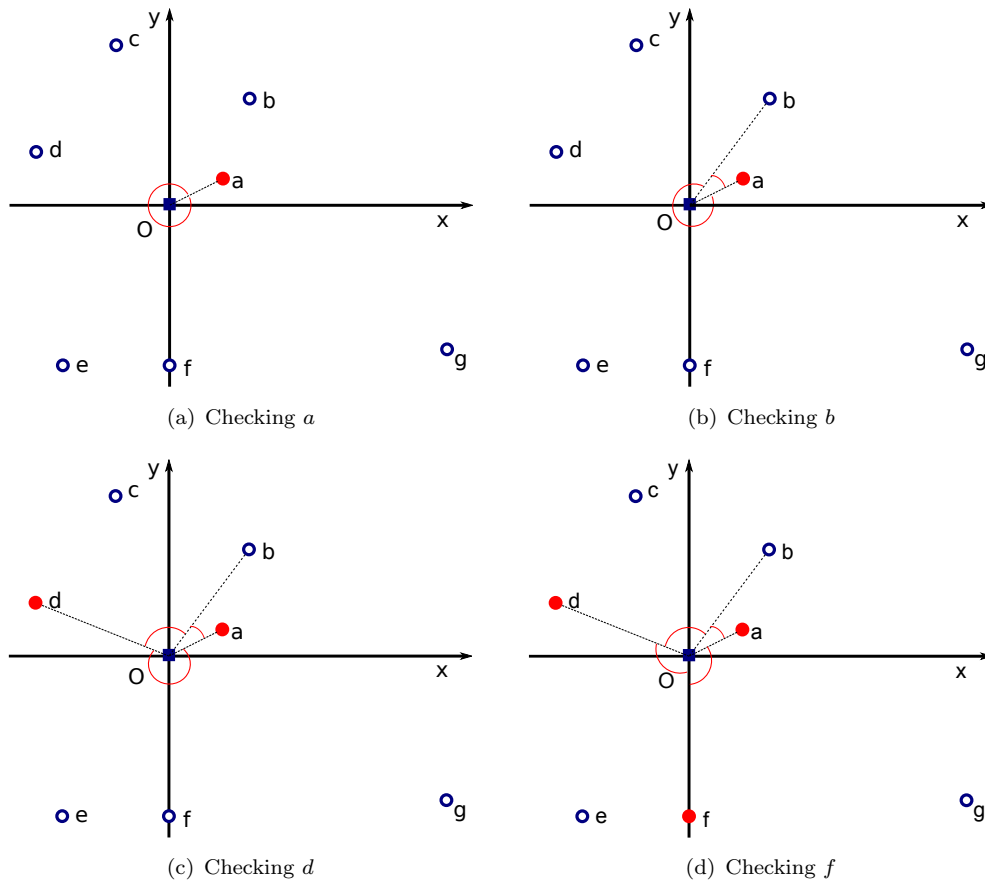


FIGURE 2.9: Processing of snapshot DBS query ($\theta = \pi/3$)

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. 2.9(a). After the evaluation of a , $P' = \{a\}$, and $\varphi_{aa} = 0^4$. Next, the algorithm checks the second-nearest object b , as illustrated in Fig. 2.9(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, the algorithm examines object d , the third nearest object to O , as depicted in Fig. 2.9(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 $\varphi_{ab} (\leq 2\theta)$, $\varphi_{bd} (\leq 2\theta)$, and $\varphi_{da} (> 2\theta)$ are formed. The procedure continues and we examine object f , as shown in Fig. 2.9(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 certainly 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 objects 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).

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 2.7.2 also demonstrate the efficiency of Algorithm 1 in supporting snapshot DBS queries.

⁴Notice that we use Property 2 as a termination condition when we have checked more than one object.

Algorithm 1: SnapshotDBSQuery (q, θ)

```

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

2.6 Processing of Continuous 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 that 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 2.2.2, 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 his/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 $\vec{q} = (x_q, y_q)'$, is mathematically expressed

in Eq. (2.3).

$$\vec{q} = \begin{pmatrix} x_q \\ y_q \end{pmatrix} = \begin{pmatrix} x_v \\ y_v \end{pmatrix} t + \begin{pmatrix} \bar{x}_q \\ \bar{y}_q \end{pmatrix}, \quad (2.3)$$

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 7. 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 7 (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 $[0, \tau]$. \square

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

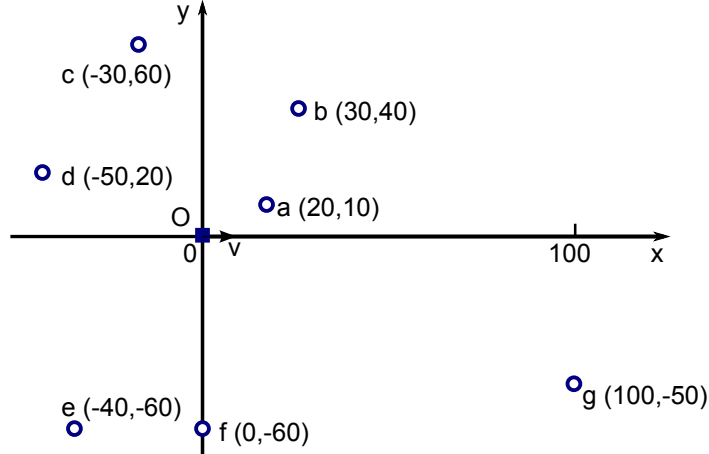


FIGURE 2.10: Example of a continuous \mathbb{E} -DBS query

Example 8. Let us extend our example of snapshot \mathbb{E} -DBS queries to the continuous case. Fig. 2.10 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:

$$DBS = \begin{cases} \{a, d, f\} & t \in [0, 4) \\ \{a, d, f, g\} & t \in [4, 23) \\ \{a, f, g\} & t \in [23, 59) \\ \{a, g\} & t \in [59, 100]. \end{cases} \quad (2.4)$$

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 in the same state 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 his/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. ■

2.6.1 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.

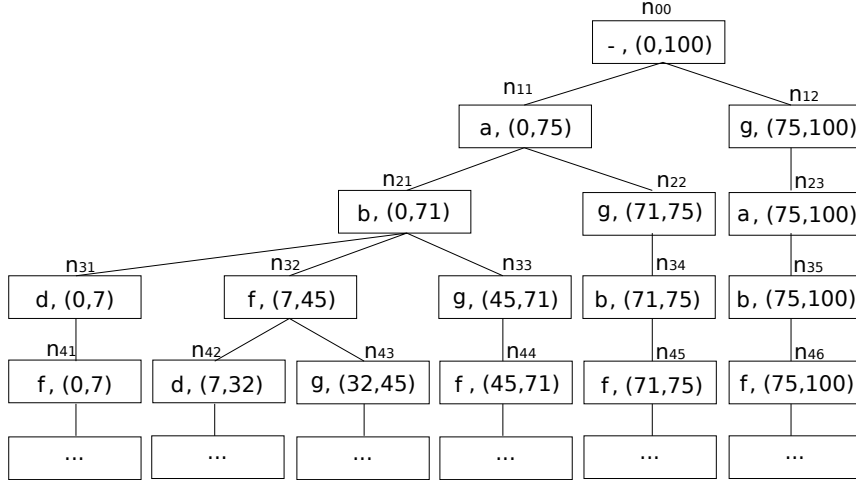


FIGURE 2.11: Process tree for continuous DBS query

As illustrated in Fig. 2.11, 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). Let the root node be at level 0, its immediate child nodes (e.g., n_{11}) be at level 1, its immediate grand-child nodes (e.g., n_{21}) be at level 2, and so on. Each child node n_{ij} at level i actually corresponds to an i^{th} 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 i^{th} 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. 2.11, 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 the interval 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 j^{th} 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 below.

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 different time intervals), they are DBS objects. As shown in Fig. 2.11, 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 objects⁵. 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.

⁵How to locate the adjacent objects for a given object when q keeps changing will be explained next.

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: ContinuousEDBSQuery $(\vec{q}, \theta, \mathcal{I})$

```

//  $\mathcal{I}$  is the target time interval:  $\mathcal{I} = [0, \tau]$ .
1 begin
2    $r \leftarrow \text{CreateRootNode}()$ ; // Create a root node
3    $S \leftarrow \emptyset$ ;
4   FindDBS( $\vec{q}, \mathcal{I}, 1, r, S$ );
5   return  $S$ ;
6 end
7 FindDBS( $\vec{q}, \mathcal{I}, k, n, S$ )
8 begin
9   forall  $\langle p, \mathcal{I}_p \rangle \in \text{CNNQuery}(\vec{q}, \mathcal{I}, k)$  do
    // Find  $k$ -th NN object(s) while  $\mathcal{I}$ 
10     $\mathcal{A} \leftarrow \text{FindAdjacentObj}(p, \mathcal{I}_p)$ ;
    // Find  $p$ 's adjacent objects while  $\mathcal{I}_p$ .
11    forall  $\langle p^-, p^+, \mathcal{I}' \rangle \in \mathcal{A}$  do
12      DomCheck( $p, \langle p^-, p^+, \mathcal{I}' \rangle, S$ ); // Check dominance.
13       $S \leftarrow \text{UpdateDBS}(S)$ ;
14    end for
15    if CannotTerminate( $p, \mathcal{I}_p$ ) then
    // Termination condition is not satisfied.
16      FindDBS( $\vec{q}, \mathcal{I}_p, k + 1, S$ ); // Expand the child nodes.
17    end if
18  end for
19 end

```

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 \vec{q} within the time interval \mathcal{I} and expands the process tree accordingly⁶ (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).

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.

In Algorithm 2, we partition intervals three times. First, we partition the intervals according to the process tree as Fig. 2.11 shows. Second, we split one interval into sub-intervals if the direction order list changes on the interval (Section 2.6.2). Third, we split a sub-interval further if the dominance relationship changes on the sub-interval (Section 2.6.3). On every sub-interval, we find DBSs until the termination condition is satisfied (Section 2.6.4).

2.6.2 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.

⁶In Appendix A.1, we explain how to implement the function `CNNQUERY` by extending the existing incremental NN algorithm proposed by Tao et al. in [7].

Example 9. Let us consider our example in Fig. 2.11 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 $L_{t=7} = \langle a, b, d, f \rangle$ and d 's adjacent objects are b and f , as depicted in Fig. 2.12(a). When $t = 32$, the direction order list $L_{t=32} = \langle b, a, d, f \rangle$ and d 's adjacent objects are a and f , as shown in Fig. 2.12(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 $\langle b, a, d, f \rangle_{(18,32)}$. ■

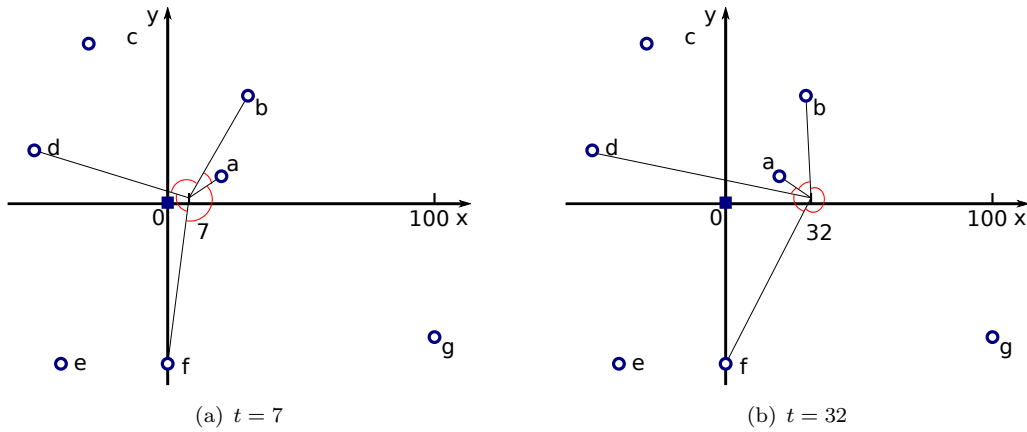


FIGURE 2.12: Change of direction order

Based on the above observation, we develop Property 3 to guide the detection of the moment where objects in the direction order list switch their positions.

Property 3. 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 \vec{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 \vec{q} , and let $line(a, b)$ represent a line passing by both a and b . The intersection between $line(a, b)$ and \vec{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. 2.2 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. ■

TABLE 2.2: Incremental maintenance of direction order lists

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

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 8). 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 10). The process repeats until all the

Algorithm 3: FindAdjacentObj(p, \mathcal{I}_p)

```

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

```

co-linear objects are evaluated. We deal with the last subinterval (t', I_e) at line 13 and line 14. 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} .

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 3 to 15) 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 5 and 6 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 to a very deep level due to the early termination strategy in Section 2.6.4.

2.6.3 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 4 is developed.

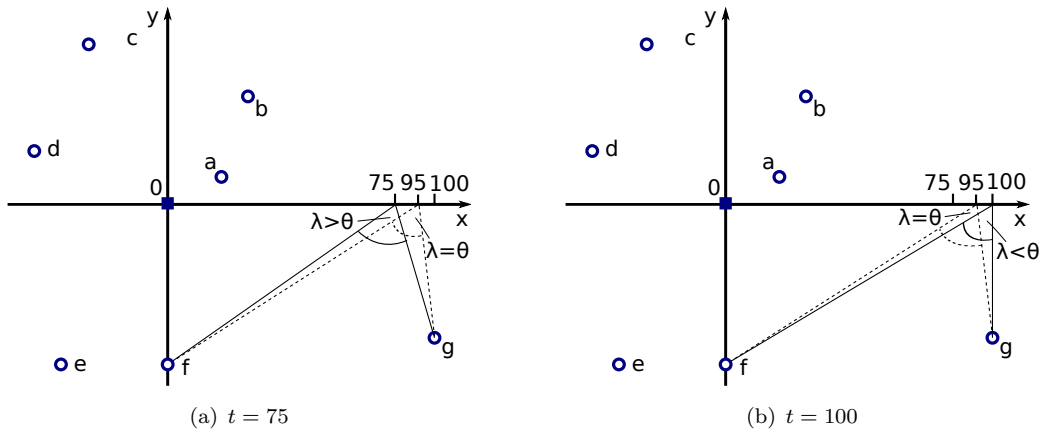


FIGURE 2.13: Change of dominance relationship

Example 10. Take Fig. 2.11 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. 2.13(a), when $t = 75$, $\lambda_{fg} > \theta = \pi/3$. However, as shown in Fig. 2.13(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 4. 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.,

$$\lambda_{ij} = \arccos \frac{\vec{p}_i \cdot \vec{p}_j}{|\vec{p}_i| |\vec{p}_j|} \geq \theta \quad (0 \leq \lambda_{ij} \leq \pi). \quad (2.5)$$

□

Note that \vec{p}_i and \vec{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 $t_i \in [t_s, t_e]$ ($i = 1, \dots, 4$) of the equation returned by GNU Scientific Library [46] (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,

$$\lambda_{ab}(t)|_{t=[t_s^j, t_e^j] \in \mathcal{I}_j} \begin{cases} \geq \theta, & \text{if } \cos \lambda_{ab}(\frac{t_s^j + t_e^j}{2}) \leq \cos \theta \\ < \theta, & \text{if } \cos \lambda_{ab}(\frac{t_s^j + t_e^j}{2}) > \cos \theta \end{cases}, \quad (2.6)$$

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.

Algorithm 4: DomCheck($p, \langle p^-, p^+, \mathcal{I} \rangle$)

```

1  $\mathcal{I}^- \leftarrow \text{UndomInterval}(p, p^-, \mathcal{I});$ 
   // Set of intervals where  $p$  is not dominated by  $p^-$ 
2  $\mathcal{I}^+ \leftarrow \text{UndomInterval}(p, p^+, \mathcal{I});$ 
   // Set of intervals where  $p$  is not dominated by  $p^+$ 
3  $\mathcal{I}_{dbs} \leftarrow \mathcal{I}^- \cap \mathcal{I}^+;$  // Set of intervals where  $p$  is on the DBS
4  $S \leftarrow S \cup \{ \langle p, \mathcal{I}_{dbs} \rangle \};$  // Add  $p$  and its intervals to DBS set  $S$ 
5 return  $S;$ 

```

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 $O(1)$.

2.6.4 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 sub-query 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 $\vec{q} = (x_v, y_v)'t$. This does not change the essence of the problem. The vectors from q to a and b are given as follows:

$$\vec{a} = \begin{pmatrix} x_a \\ y_a \end{pmatrix} - \begin{pmatrix} x_v \\ y_v \end{pmatrix} t \quad (2.7)$$

$$\vec{b} = \begin{pmatrix} x_b \\ y_b \end{pmatrix} - \begin{pmatrix} x_v \\ y_v \end{pmatrix} t. \quad (2.8)$$

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 \vec{ab} is not parallel to \vec{q} . The condition is expressed as follows:

$$(\vec{ab} \times \vec{q} \neq 0) \wedge ((\vec{a} \times \vec{q})(\vec{b} \times \vec{q}) > 0). \quad (2.9)$$

The notation $\vec{a} \times \vec{q}$ represents the *outer product* of \vec{a} and \vec{q} ⁷. The condition $(\vec{ab} \times \vec{q} \neq 0)$ represents that \vec{ab} is not parallel to \vec{q} . The condition $((\vec{a} \times \vec{q})(\vec{b} \times \vec{q}) > 0)$ means that a and b are on the same side of the user's trajectory. In this

⁷The outer product of two vectors $\vec{v} = (v_x, v_y)'$ and $\vec{w} = (w_x, w_y)'$ in the two-dimensional case is defined as

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

where η is the angle between two vectors.

case, $\cos \varphi_{ab}$ takes a local maximum 1 at

$$t = \frac{\vec{a} \times \vec{b}}{\vec{ba} \times \vec{q}}, \quad (2.10)$$

and takes two local minima at

$$t = \frac{-|\vec{q}|(\vec{a} \times \vec{b}) \pm |\vec{ab}| \sqrt{(\vec{a} \times \vec{q})(\vec{b} \times \vec{q})}}{|\vec{q}|(\vec{ab} \times \vec{q})}. \quad (2.11)$$

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

$$(\vec{ab} \times \vec{q} \neq 0) \wedge ((\vec{a} \times \vec{q})(\vec{b} \times \vec{q}) < 0). \quad (2.12)$$

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

$$t = \frac{\vec{a} \times \vec{b}}{\vec{ba} \times \vec{q}}. \quad (2.13)$$

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

$$\vec{ab} \times \vec{q} = 0. \quad (2.14)$$

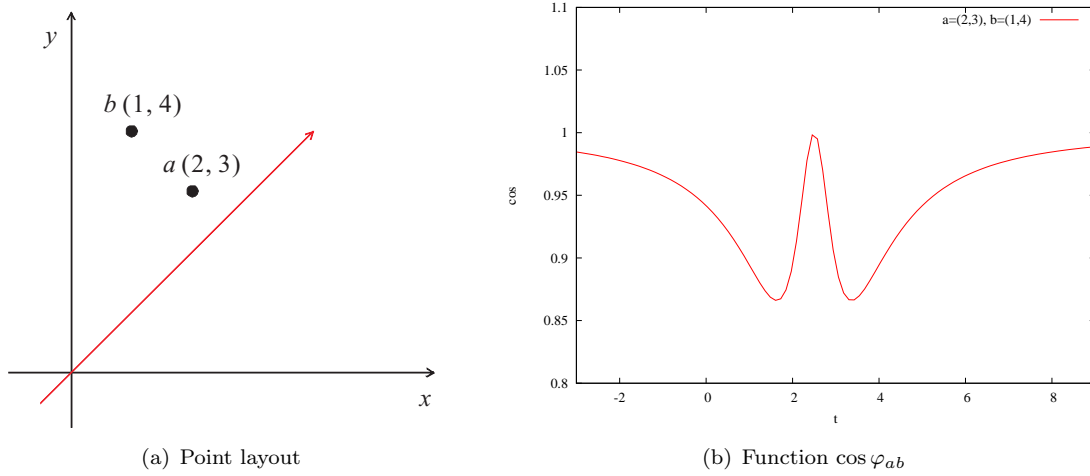
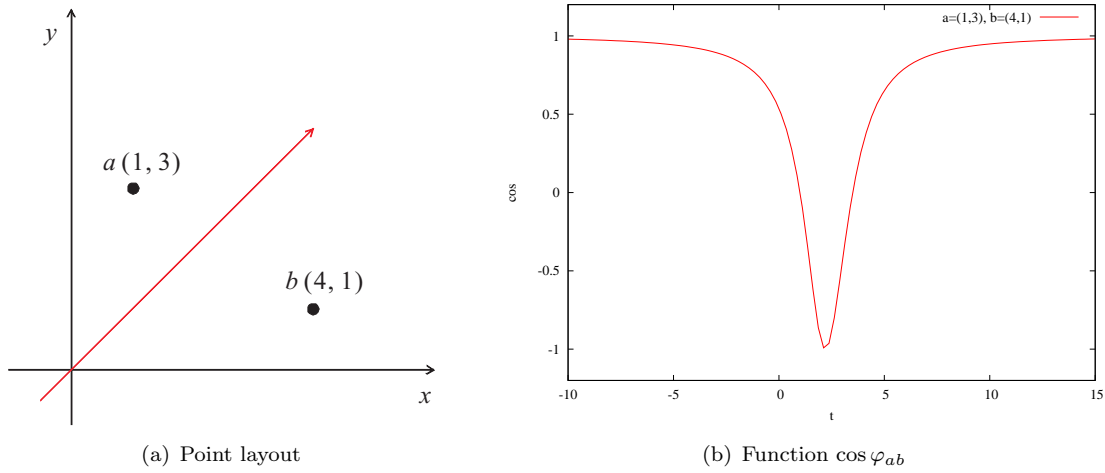
The function $\cos \varphi_{ab}$ takes a local minimum at

$$t = \frac{(|\vec{a}|^2 \vec{b} - |\vec{b}|^2 \vec{a}) \times \vec{q}}{2|\vec{q}|^2(\vec{b} \times \vec{a})}. \quad (2.15)$$

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

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

- *Case A*: In Fig. 2.14, $\cos \varphi_{ab}$ takes a local maximum at $t = 2.5$ and two local minima at $t = 1.63$ and $t = 3.67$.
- *Case B*: In Fig. 2.15, $\cos \varphi_{ab}$ takes a local minimum $t = 2.2$.
- *Case C*: In Fig. 2.16, $\cos \varphi_{ab}$ takes a local minimum $t = 3$.

FIGURE 2.14: Case A: $a = (2, 3)', b = (1, 4)'$ FIGURE 2.15: Case B: $a = (1, 3)', b = (4, 1)'$

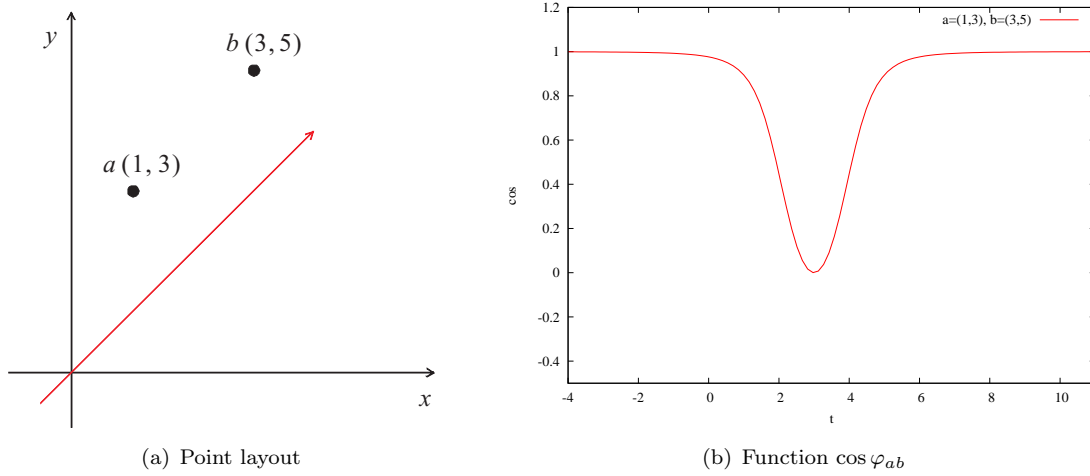
Then we need to check whether $\cos \varphi_{ab} > \cos 2\theta$ holds during the given time interval $\mathcal{I} = [I_s, I_e]$. In order to check this condition, we calculate the minimum value of $\cos \varphi_{ab}$ for the given interval \mathcal{I} . If the minimum value is greater than $\cos 2\theta$, 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, 2.4, and 2.5.

Note that, the constitution of the partition angles changes when the user moves. In Example 9, the sub-query q_{42} generates two direction lists $L_{(7,18)} = \langle a, b, d, f \rangle$ and $L_{(18,32)} =$

⁸Note: t_1 and t_3 are two t -values when $\cos \varphi_{ab}$ takes two local minima of $\cos \varphi_{ab}$ (Eq. (2.11)), and t_2 is the t -value when $\cos \varphi_{ab}$ takes the local maximum of $\cos \varphi_{ab}$ (Eq. (2.10)).

⁹Note: t^* is the t -value when $\cos \varphi_{ab}$ takes a local minimum (Eq. (2.13)).

¹⁰Note: t^* is the t -value when $\cos \varphi_{ab}$ takes a local minimum (Eq. (2.15)).

FIGURE 2.16: Case C: $a = (1, 3)'$, $b = (3, 5)'$ TABLE 2.3: Decision table for case A⁸

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

TABLE 2.4: Decision table for case B⁹

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

$\langle b, a, d, f \rangle$ as shown in Fig. 2.2. 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 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.

TABLE 2.5: Decision table for case C¹⁰

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

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: CannotTerminate(p, \mathcal{I}_p)

```

1 forall  $\langle l, I \rangle \in \mathcal{D}$  do
    // Each list in list set  $\mathcal{D}$  of node  $\langle p, \mathcal{I}_p \rangle$ .
2    $cnt \leftarrow 0$ ; // Counter for valid  $\varphi$ 's.
3   forall  $i \leftarrow 1$  to  $|l|$  do // Fetch every object  $o_i$  in  $l$ .
4     if  $\varphi_{i,(i+1)}^I \leq 2\theta$  then //  $\varphi_{i,(i+1)}^I$  is formed by  $o_i$  and  $o_{i+1}$ .
5       // Assume that  $\varphi_{|l|,|l|+1}^I = \varphi_{I,1}^I$ .
6        $cnt \leftarrow cnt + 1$ ; // Increment  $cnt$ .
7     end if
8   end for
9   if  $cnt \neq |l|$  then // Not all  $\varphi$ 's are valid. Cannot terminate.
10    return true;
11  end if
12 end for
13 return false; // All  $\varphi$ 's are valid. We can terminate.

```

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 1-11). 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 3 - 7) 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 9). After checking all direction lists and we do not return in the outermost loop (line 9), 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 2.6.2. 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.

2.7 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} .

2.7.1 Settings

In the Euclidean space \mathbb{E} , we use both real and synthetic datasets, with their properties summarized in Table 2.6. For the real dataset, denoted as *Real*, we consider the road line segments of *Long Beach* in the TIGER database [47], 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 *Syn* $\rho\%$, 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 [48] 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 2.6: Datasets

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

2.7.2 Performances of Snapshot 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. 2.17.

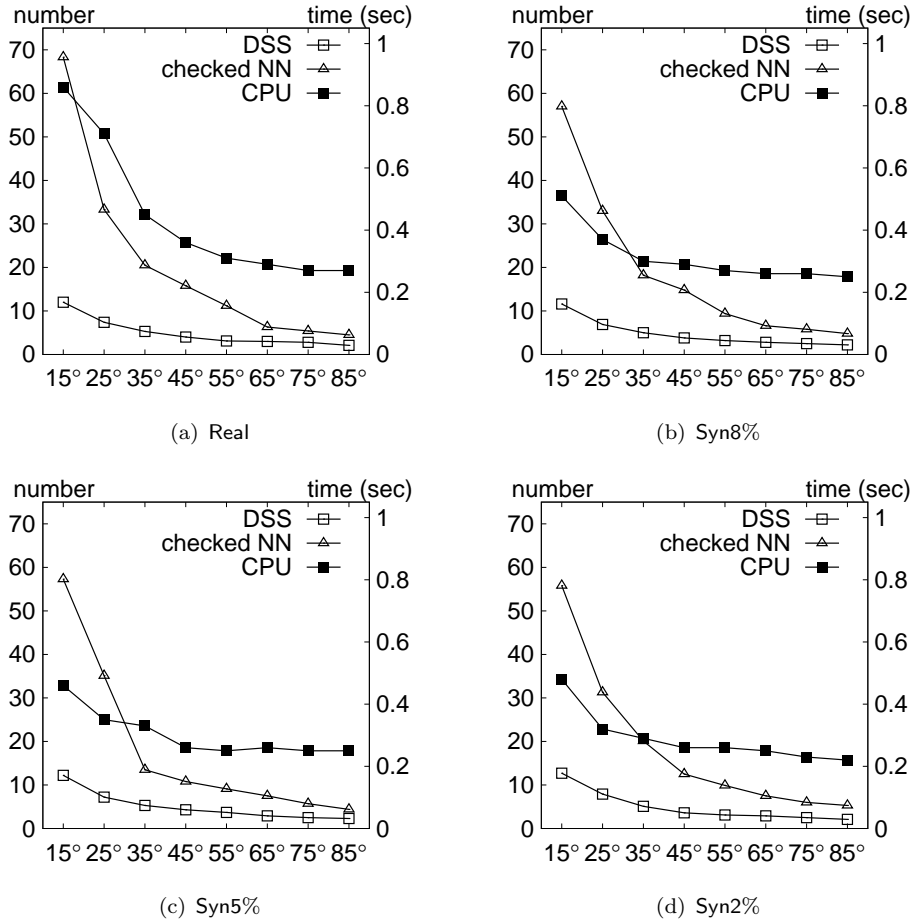


FIGURE 2.17: Performance of snapshot queries w.r.t. θ

As shown in Fig. 2.17, the total number of DBS objects changes when θ varies in the range of $[15^\circ, 85^\circ]$. 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 of the densities of the datasets.

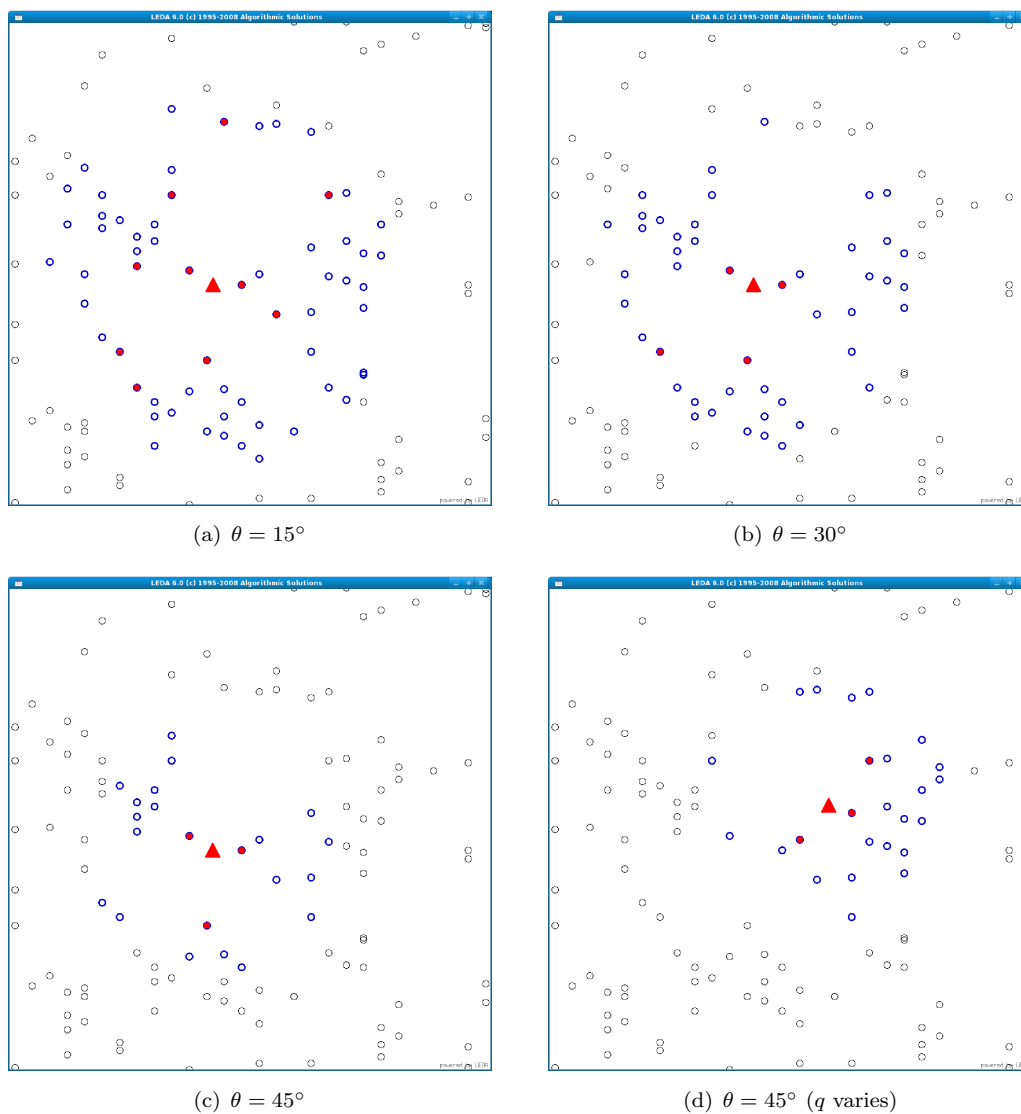


FIGURE 2.18: Screenshot of snapshot queries

We also capture some screenshots of the DBS points corresponding to different θ 's and user position for the snapshot case, as depicted in Fig. 2.18. 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. 2.18(a), Fig. 2.18(b) and Fig. 2.18(c) refer to the case that 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 are reduced. On the other hand, Fig. 2.18(c) and Fig. 2.18(d) demonstrate the case that θ value is fixed at 45° but user positions change.

In addition, we also evaluate the influence 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. 2.19. We observe that the average number of DBSs, the number of checked NNs, and the CPU cost are not affected by the object distributions.

To sum up, we have the following findings after the experiments on the snapshot queries:

- The DBS objects are few in number, i.e., less than twenty.
- The number of DBS objects is influenced by the threshold θ , i.e., it gets fewer when the θ gets larger, but not influenced by the size nor the density of the dataset.
- The proposed algorithm can answer a snapshot query promptly, i.e., in less than one second.
- The query cost is influenced by the threshold θ , i.e., it decreases when θ gets larger, but not influenced by the size nor the density of the dataset.

2.7.3 Performances of Continuous 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 selects a position as the starting point

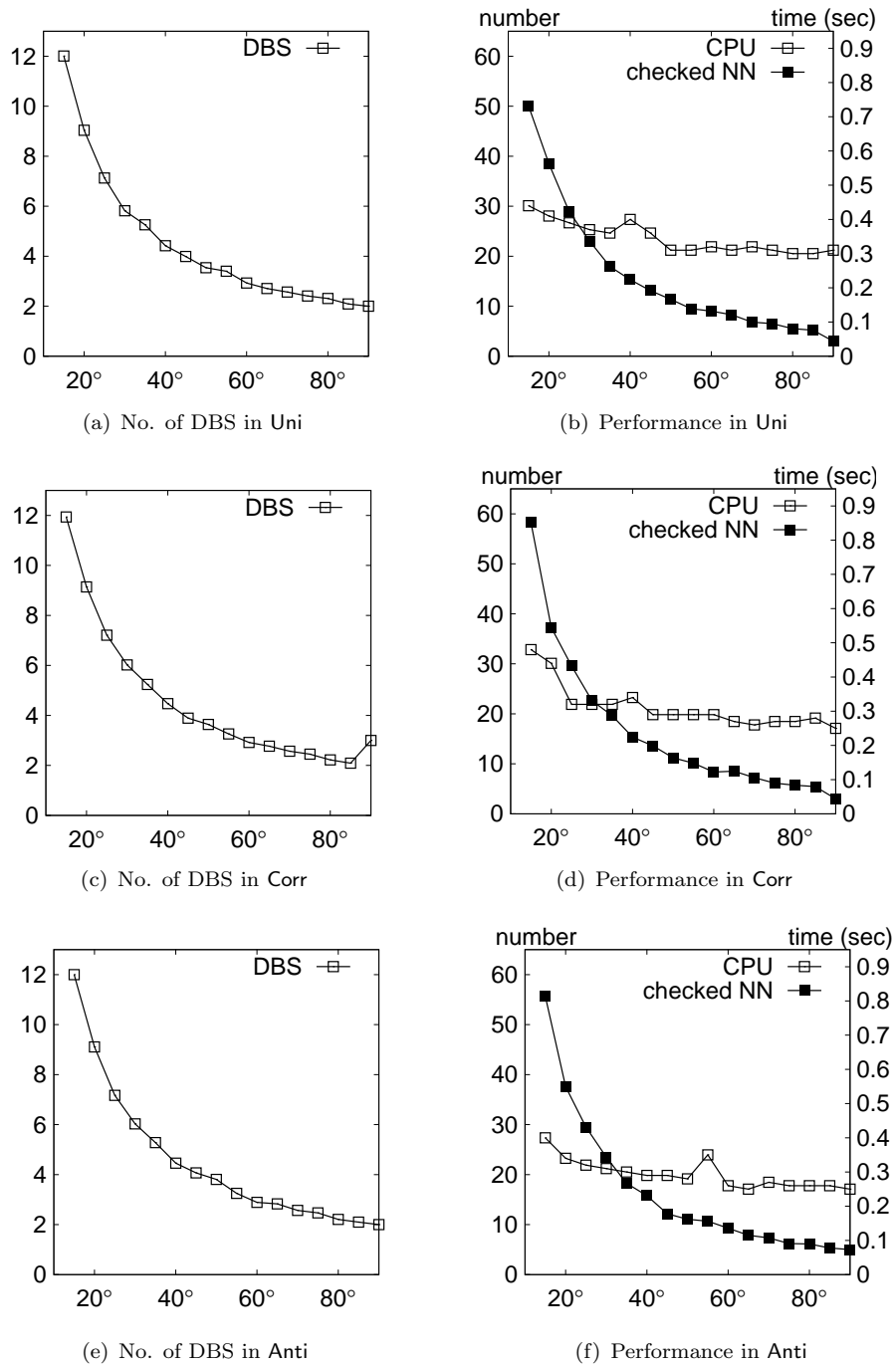
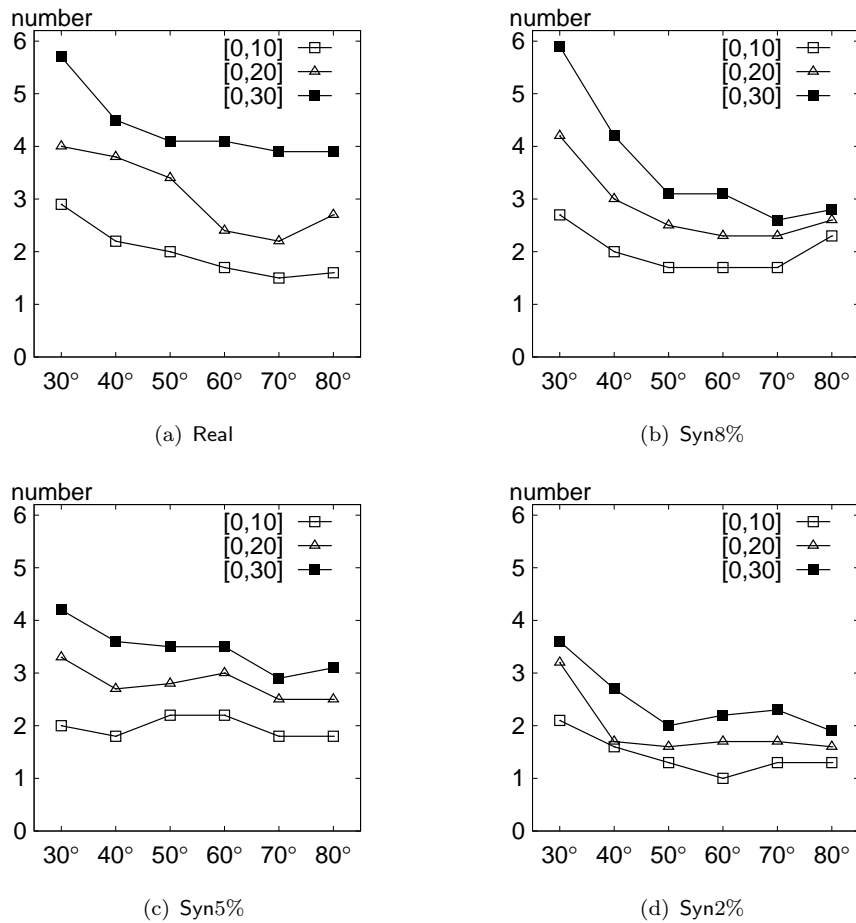


FIGURE 2.19: Performance of snapshot queries for data sets with different distributions

and then keeps moving with a constant speed of 0.06 unit distance per unit time¹¹ along the positive x -axis during different time intervals (i.e., $[0, 10]$, $[0, 20]$ and $[0, 30]$).

Fig. 2.20 shows the number of change moments under different θ 's and different time

¹¹We 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.

FIGURE 2.20: Number of change moments of continuous queries w.r.t. θ

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 the result of 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. 2.21 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 its size when θ increases; because the larger the θ is, the easier the termination condition is

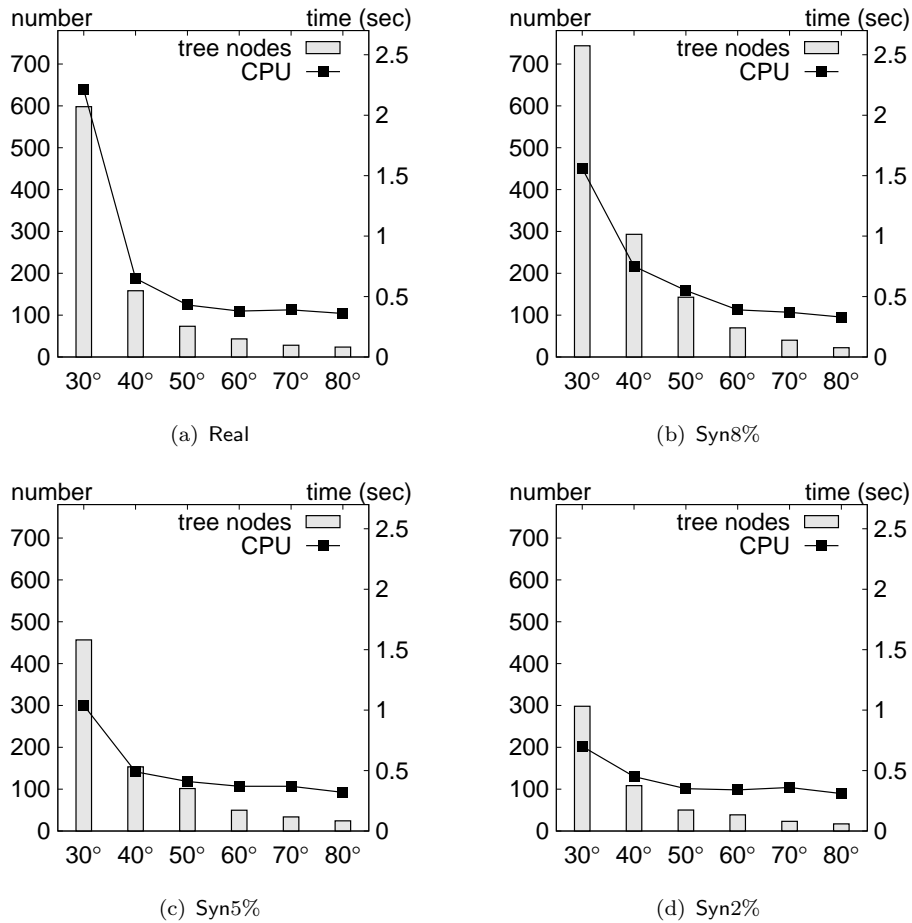


FIGURE 2.21: Tree sizes and CPU costs of continuous queries vs. θ w.r.t. time interval $[0, 30]$

achieved. It is observed that only a small number of objects (in average around 1.4% of the dataset) require evaluations. 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 influence 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. 2.22, we also present tree depths for continuous 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

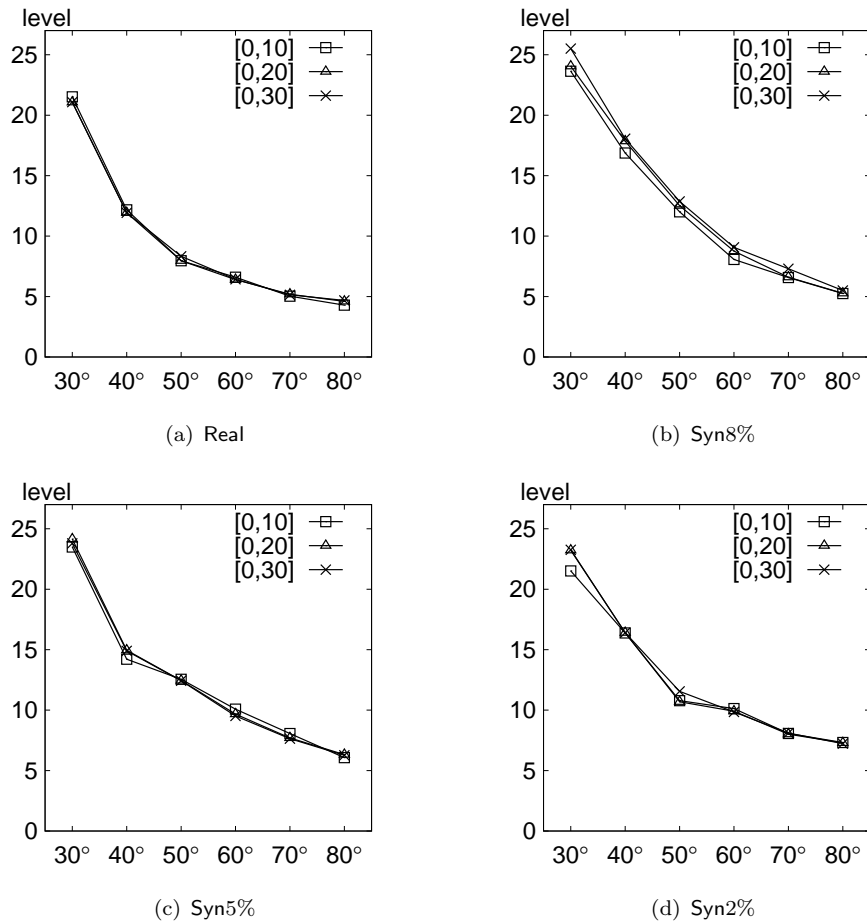


FIGURE 2.22: Tree depths of continuous queries

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. 2.23 (a) shows the numbers of change moments for the three data sets with different object distributions. Fig. 2.23 (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.

To sum up, we have the following findings after the experiments on the continuous queries:

- The change moments are few in number, i.e., less than three within 600 meters.

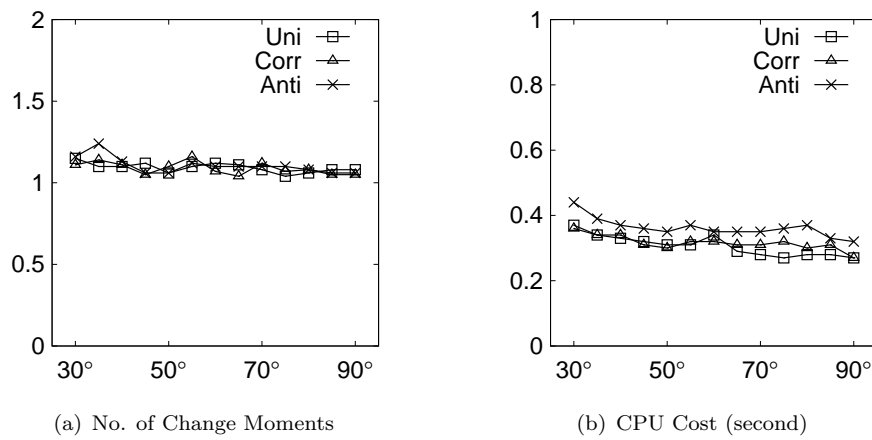


FIGURE 2.23: Performance of continuous queries for data sets with different distributions

- The number of change moments is influenced by the threshold θ , i.e., it gets fewer when θ gets larger, and also influenced by the density of the dataset, i.e., it gets fewer when the density gets lower.
- The proposed algorithm can answer a continuous query promptly, i.e., in less than 2.5 second.
- The query cost is influenced by the threshold θ , i.e., it decreases when θ gets larger, and also influenced by the density of the dataset, i.e., it decreases when the density gets lower.

Chapter 3

Direction-Based Surround Queries in Road Networks

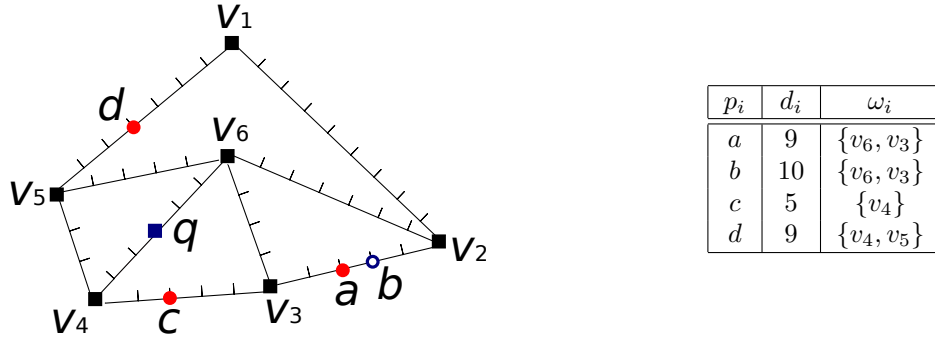
In this chapter, we assume that the objects are in a road network and the corresponding queries are called \mathbb{R} -DBS queries. For \mathbb{R} -DBS queries, we measure the distance and the direction of an object p_i w.r.t. q based on the shortest path from q to p_i . Typically, when an exact road 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.

3.1 Problem

3.1.1 Snapshot \mathbb{R} -DBS Queries

Before presenting the formal definition of an \mathbb{R} -DBS query, we use Example 12 to illustrate DBS queries in a road network \mathbb{R} . It also serves as the running example in this chapter.

Example 12 (Snapshot \mathbb{R} -DBS Queries). In Fig. 3.1, a road network is represented by a graph with six vertices $V = \{v_1, \dots, v_6\}$ 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., $q \rightarrow v_6 \rightarrow v_3 \rightarrow a$, 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. 3.1.

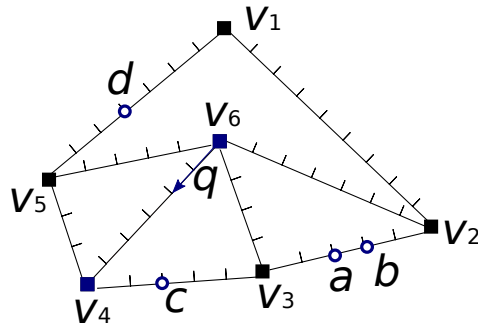
FIGURE 3.1: Example of an \mathbb{R} -DBS query

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. ■

3.1.2 Continuous \mathbb{R} -DBS Queries

Example 13 is an extension of Example 12 to illustrate the idea of continuous \mathbb{R} -DBS queries.

Example 13 (Continuous \mathbb{R} -DBS Queries). As shown in Fig. 3.2, 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 . ■

FIGURE 3.2: Example of a continuous \mathbb{R} -DBS query

In the following, we formalize the snapshot DBS query and continuous DBS query in the road network \mathbb{R} , and present the corresponding query processing algorithms.

3.2 Related Work

3.2.1 k NN 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.

Snapshot k NN Queries in Road Networks The snapshot \mathbb{R} -DBS query is related to k NN queries in road networks. Papadias et al. [49] proposed a flexible architecture for spatial network databases in order to answer spatial queries including k NN queries. Their IER/INE algorithms find out k NN objects by performing network expansions which are inspired with the Dijkstra's algorithm [50]. Kolahdouzan et al. [51] proposed VN³ algorithms to answer k NN queries by partitioning a spatial network into smaller Voronoi polygons over objects and pre-computing some network distances. Hu et al. [25, 52] built indexes to facilitate k NN search in road networks. In [25], they performed k NN searches by retrieving a set of interconnected trees which are generated from the road network. In [52], they use distance signatures to maintain approximate network distances and build an index based on the distance signatures in order to speed up k NN searches. Lee et al. [53] pruned the k NN search space by skipping *Rnet* which is subspaces containing no objects.

Continuous k NN Queries in Road Networks The continuous \mathbb{R} -DBS query is related to continuous k NN queries for a moving query position on a query path [23, 54]. Kolahdouzan et al. [54] proposed IE/UBA algorithms to find out k NN candidates first and then split the query path into sub-paths where the k NN objects are the same. Cho et al. [23] 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 k NNs are the same no matter where the query position is. The continuous \mathbb{R} -DBS query is different from the continuous k NN queries for a query path.

Dynamic k NN Queries in Road Networks There are many studies for dynamic k NN queries which are different from the continuous k NN queries. The dynamic k NN queries have dynamic objects of interest or even dynamic road networks as well as a dynamic query position. Shahabi et al. [55] focused on k NN 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. [56] proposed a framework and also implemented a prototype to answer k NN queries for moving objects and a moving position. Mouratidis et al. [57] proposed IMA/GMA algorithms which update results when the changes on objects, query positions, and edges may influence the current results. Demiryurek et al. [58] proposed more efficient algorithms to solve the same problems in [57]. Their ER- Ck NN algorithms avoid blind network expansions in [58] by finding candidates which are selected based on their Euclidean distances to the query position. Samet et al. [59] proposed efficient algorithms to answer k NN queries when many different queries are issued or different sets of objects are used for static road networks. They proposed the shortest path quad tree to avoid repeatedly calculating shortest paths between two vertices for different query positions.

3.2.2 Path Nearest Neighbor Query

Chen et al. identified the *path nearest neighbor query* which retrieves the nearest neighbor along the user's moving path [24]. As Fig. 3.3 shows, the route of the user is from the start point to the end point. Along her route, the nearest object is object b . Thus, object b is the answer to the query.

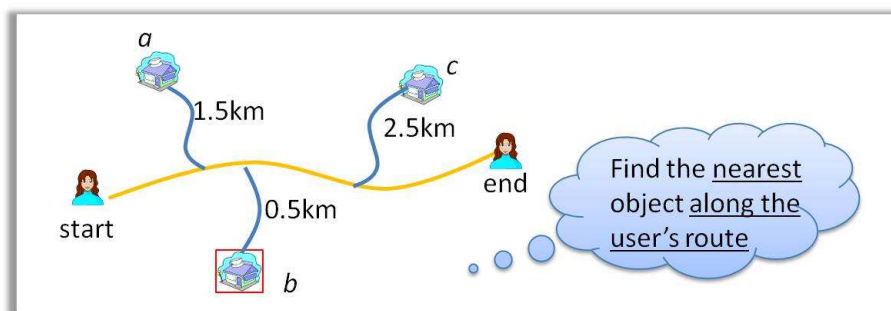


FIGURE 3.3: Path nearest neighbour query

3.2.3 Quoted Work

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 [50, 60, 61].

3.2.4 Summary

Table 3.1 summarizes the related work of \mathbb{R} -DBS queries and illustrates the position of our work. In road networks, we can also divide the interests of the users into two categories, namely, spatial interests and non-spatial interests. Considering the state of the query point, we also divide the queries into two types, namely, snapshot queries and continuous queries. There are a lot of work considering distances only in both snapshot queries ([49], etc.) and continuous queries ([23], etc.). Especially, there is one paper considering not only distances but also the route of the user [24]. Our work [45] is the first study considering distances as well as directions in road networks.

TABLE 3.1: Related work of \mathbb{R} -DBS queries

query point \ interest	spatial			non-spatial
	distance	+route	+direction	+(price, etc.)
snapshot	[49], etc.	-	[45]	
continuous	[23], etc.	[24]	[45]	

3.3 Preliminaries

In the metric space \mathbb{R} , we compare objects using network distances and directions. Before defining the *DBS* problem formally, we would like to define the *dominance relationship* first.

3.3.1 Dominance Relationship and \mathbb{R} -DBS Query

Definition 8 (Dominance Relationship). In a road network space \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 $p_i \prec p_j$. \square

We will define the *direction closeness* in the road network space \mathbb{R} (Section 3.3.2). Accordingly, *DBS queries* in road networks are defined in Definition 9.

Definition 9 (\mathbb{R} -DBS Query). Given a set of POI objects $P = \{p_1, \dots, p_n\}$ and a query point q in a road network space \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 $DBS(q, G)$ in \mathbb{R} , is to find all the direction-based surrounder points, i.e., $\{p_i \mid p_i \in P, \nexists p_j (\neq p_i) \in P, p_j \prec p_i\}$. \square

3.3.2 Directional Closeness

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 $P = \{p_1, \dots, p_n\}$ are all located at some edges, i.e., $\forall p \in q \cup P, \exists e \in E \wedge p \in e$. Let $SP(q, p_i) = \{sp_1, \dots, sp_j\}$ be the shortest path from user q to a POI object p_i with $\{sp_1, \dots, sp_j\}$ 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, \dots, sp_j , and finally reaches the destination p_i . In Fig. 3.1, $SP(q, a) = \{v_6, v_3\}$ 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, \dots, sp_j). Therefore, $SP(q, a) = (9, v_6v_3)$, and $SP(q, b) = (10, v_6v_3)$ in Fig. 3.1.

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 10. Based on this definition, in Fig. 3.1, 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 10 (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 $SP(q, p_j).\omega_j \subseteq SP(q, p_i).\omega_i$. \square

¹In 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.

3.4 Processing of Snapshot Queries

In this section, we first present a naïve algorithm to answer snapshot \mathbb{R} -DBS queries, and then propose an optimized algorithm based on the naïve one.

3.4.1 Property

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. We will first define an important property to facilitate the process, and then explain the search algorithm.

Property 5. 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 $p_i \in SP(q, p_j)$, $p_i \prec p_j$. \square

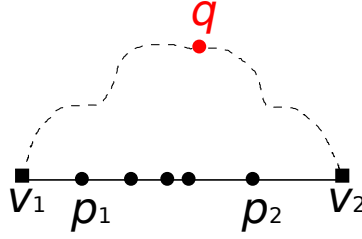
In the following, we use Example 14 to illustrate the property.

Example 14. In Fig. 3.1, 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 object a locates on the path $SP(q, b)$ 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, $DBS(q, G) = \{a, c, d\}$. \blacksquare

3.4.2 Naïve Algorithm

Property 5 inspires a simple approach to answer a snapshot DBS query on a road network. Given the fact that shortest path searches have been well studied in [50, 60, 61], we assume the shortest path from q to each vertex $v \in V$ is identified by some existing search algorithm (e.g., the Dijkstra algorithm [50]).

Given an object $p \in P$ on an edge $e \in E$, the shortest path from q to p either passes v_1 (i.e., $\{q, \dots, v_1, p\}$) or passes v_2 (i.e., $\{q, \dots, v_2, p\}$), where v_1 and v_2 are the end nodes of the edge e . Comparing the lengths of the two paths, the shorter one should be the shortest path from q to p . If there is no other object on the shortest path, object p is a DBS, otherwise, it is dominated. In order to find all DBSs, we go through all the edges

FIGURE 3.4: Naïve algorithm for snapshot \mathbb{R} -DBS queries

and check the objects on them. The non-dominated objects would be selected as the answers.

Additionally, we can check only a set of promising objects rather than all. As Fig. 3.4 shows, when checking the objects on an edge $e(v_1, v_2)$, we only need to consider the nearest object p_1 to the end node v_1 and the nearest object p_2 to the end node v_2 . The reason is that p_1 and p_2 are on the shortest paths from q to the other objects on the edge e , thus, the other objects are definitely dominated either by p_1 or by p_2 . Algorithm 6 summarizes this naïve approach.

Algorithm 6: SnapshotRDBSQueryNaïve($G(V, E)$, P , q)

```

1 for each  $v \in V$  do
2   Calculate the shortest path  $sp_v$  from  $q$  to  $v$  using Dijkstra algorithm;
3 end for
4 for each  $e(v_1, v_2) \in E$  do
5    $p_1 \leftarrow$  the nearest object to  $v_1$ ;
6    $p_2 \leftarrow$  the nearest object to  $v_2$ ;
7    $sp_1 \leftarrow$  the shortest path from  $q$  to  $p_1$ ;
8    $sp_2 \leftarrow$  the shortest path from  $q$  to  $p_2$ ;
9   if  $\nexists p \in P - \{p_1\}$  locates on  $sp_1$  then  $DBS \leftarrow DBS \cup \{p_1\}$ ;
10  if  $\nexists p \in P - \{p_2\}$  locates on  $sp_2$  then  $DBS \leftarrow DBS \cup \{p_2\}$ ;
11 end for
12 return  $DBS$ ;
```

3.4.3 Optimized Algorithm

In the naïve algorithm, we have to calculate the shortest paths from q to all the vertices on the road network, however, it is not necessary. The reason is that some objects can be determined to be definitely dominated without calculating their shortest paths. Let us define the *dominance of a vertex* first.

Definition 11. Given a vertex $v \in V$, if there is any object on the shortest path from q to v , the vertex v is *dominated*. \square

As Fig. 3.5 shows, there is an object on the shortest path from q to v_1 , thus, the vertex v_1 is dominated. In the same way, the vertex v_2 is also dominated. Consider an edge

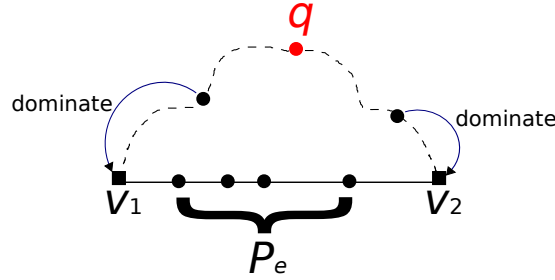


FIGURE 3.5: Optimized algorithm for snapshot \mathbb{R} -DBS queries

$e(v_1, v_2)$ with objects P_e on it. We can determine whether the objects P_e are dominated using Property 6 below.

Property 6. Given an edge $e \in E$ with the end nodes v_1 and v_2 , the objects P_e on e are dominated if both v_1 and v_2 are dominated. \square

As Fig. 3.5 shows, the objects P_e are dominated because both v_1 and v_2 are dominated.

In Dijkstra algorithm [50], we commonly use a set Q to maintain the vertices that have been visited. In the initialization step, the distance of the initial node q^2 is set to zero, the distances of the other vertices are set to infinity, and the initial node is added to the empty set Q . Next, we process the vertex $v \in Q$ with the minimum distance everytime. The procedure includes to determine the shortest path from q to v , to update the distances of v 's neighbors, and to add the neighbors to Q if they are not in Q . The process continues until the set Q becomes empty.

In our approach, in order to make the Dijkstra algorithm terminate earlier, when processing a vertex v , we also determine whether it is dominated or not. In addition, if its neighbor's distance has been updated, the neighbor's dominance status should be set to the same status as v 's. The Dijkstra algorithm will terminate when all the vertices in Q are dominated. It also means that the unvisited vertices are definitely dominated. Thus, we only need to check the objects that are on the edges having at least one non-dominated end node. Algorithm 7 summarizes the optimized approach.

²For simplicity, we assume that the user is at a vertex on the road network.

Algorithm 7: SnapshotRDBSQueryOpt($G(V, E), P, q$)

```

1  $v \leftarrow q$ ;
2  $v.dist \leftarrow 0.0$ ; // Set the distance of  $v$ .
3  $Q \leftarrow \{v\}$ ; // Add  $v$  to the empty set  $Q$ .
4  $CE \leftarrow \emptyset$ ; // Use  $CE$  to maintain the checked edges.
5 while  $\exists v \in Q$  is not dominated do
6    $sp_v \leftarrow$  the shortest path from  $q$  to  $v$ ;
7   if  $\exists p \in P$  locates on  $sp_v$  then // Set the dominance status of  $v$ .
8      $v.domed \leftarrow true$ ;
9   else
10     $v.domed \leftarrow false$ ;
11   end if
12   for each neighbor  $u$  of  $v$  do // Update neighbors
13      $alt \leftarrow v.dist + |e(u, v)|$ ;
14     if  $alt < u.dist$  then
15        $u.dist \leftarrow alt$ ; // Update the distance.
16        $u.prev \leftarrow v$ ;
17        $u.domed \leftarrow v.domed$ ; // Update the dominance status.
18     end if
19      $CE \leftarrow CE \cup \{e(v, u)\}$ ; // Add the checked edge to  $CE$ .
20     if  $u \notin Q$  then  $Q \leftarrow Q \cup \{u\}$ ; // Add the neighbor to  $Q$ .
21   end for
22    $Q \leftarrow Q - \{v\}$ ; // Remove  $v$  from  $Q$ .
23    $v \leftarrow$  the vertex in  $Q$  with the minimum distance;
24 end while
25 for each  $e(v_1, v_2) \in CE$  do
26   Set the dominance status of  $p_1$  and  $p_2$ ; //  $p_1$  and  $p_2$  are the NNs to  $v_1$  and  $v_2$ .
27   if  $p_1$  is not dominated then  $DBS \leftarrow DBS \cup \{p_1\}$ ;
28   if  $p_2$  is not dominated then  $DBS \leftarrow DBS \cup \{p_2\}$ ;
29 end for
30 return  $DBS$ ;

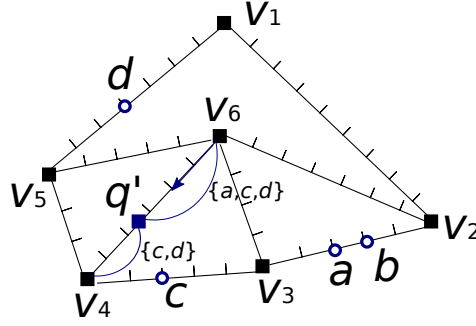
```

3.5 Processing of Continuous Queries

In this section, we propose the algorithms to answer the \mathbb{R} -DBS queries. At first, we extend our running example in order to illustrate the concept of continuous \mathbb{R} -DBS queries.

Example 15. Let us extend our snapshot \mathbb{R} -DBS query example to the continuous case. Fig. 3.6 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:

$$DBS = \begin{cases} \{a, c, d\} & t \in [0, 3.5) \\ \{c, d\} & t \in [3.5, 6] \end{cases} \quad (3.1)$$

FIGURE 3.6: Example of a continuous \mathbb{R} -DBS query

The result indicates that when the user at his/her start position 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 . ■

3.5.1 Basic Idea

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 $P = \{p_1, \dots, p_n\}$ 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 moments 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 moments when DBS results change, i.e., *change positions* to be distinguished 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) $\cup e' = e(v_i, v_j)$;
- (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.

3.5.2 Properties

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 sp_1 , 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 , vice versa. Property 7 is thereafter developed to locate the position s_p along $e(v_i, v_j)$ that p changes its direction.

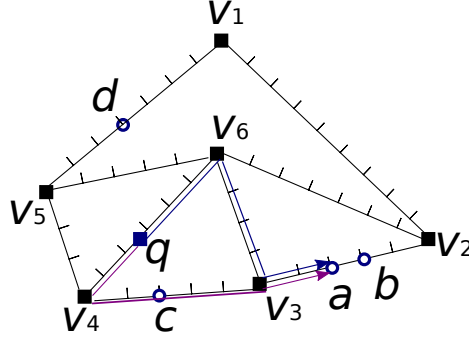
Property 7. Given a query point q moving on an edge $e(v_i, v_j)$ from vertex v_i to vertex v_j and an object p , p changes its direction only when q reaches point s_p along e with $dist(v_i, s_p)$ expressed in Equation (3.2)³.

$$dist(v_i, s_p) = \frac{1}{2} \cdot (|e| + ||SP(v_j, p)| - |SP(v_i, p)||), \quad (3.2)$$

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 . \square

Example 16. An example is depicted in Fig. 3.7. 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 7, 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

³To 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 $dist(v_i, s_p)$.

FIGURE 3.7: Property 7 of continuous \mathbb{R} -DBS queries

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 $(3.5, 6)$. ■

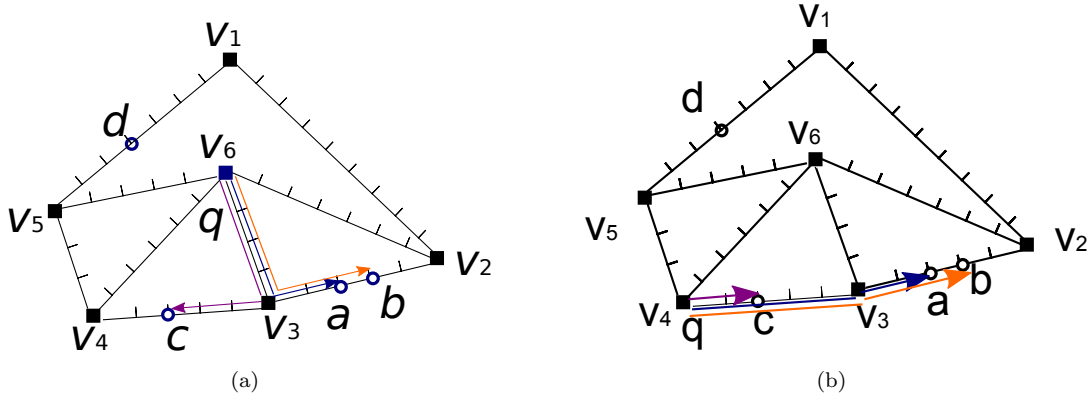
We also observe that the dominance relationship has the continuity property as presented in Properties 8 and 9, respectively.

Property 8. 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 9. 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 8 and Property 9 are presented in Appendix A.4.

Example 17. An example is depicted in Fig. 3.8. When $q = v_6$, $SP(q, a) = (6, v_6v_3)$, $SP(q, b) = (7, v_6v_3)$ and thus $a \prec b$. When $q = v_4$, $SP(q, a) = (7, v_4v_3)$, $SP(q, b) = (8, v_4v_3)$ and hence $a \prec b$. Based on Property 8, b is dominated when q is located at any point on the edge $e(v_6, v_4)$, 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 9, 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. $q = v_4$. ■

FIGURE 3.8: Properties 8 and 9 of continuous \mathbb{R} -DBS queries

3.5.3 Algorithms

Given a continuous DBS query issued at edge $e(v_i, v_j)$, Property 8 guarantees that objects dominated w.r.t. $q = v_i$ and $q = v_j$ are excluded from the answer set, and Property 9 guarantees that DBS points w.r.t. both $q = v_i$ and $q = v_j$ are certainly DBS points w.r.t. q located at any position of $e(v_i, v_j)$. In other words, $DBS(v_i, G) \cup DBS(v_j, G)$ forms the superset of the answer set, i.e., candidates of DBS points w.r.t. $q \in e(v_i, v_j)$ must be in $DBS(v_i, G) \cup DBS(v_j, G)$. Consequently, we can first issue two snapshot \mathbb{R} -DBS queries on points 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 9, 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 7. 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 (3.2) 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 8 presents the procedure of continuous \mathbb{R} -DBS Queries.

Algorithm 8: ContinuousRDBSQuery($G, P, e^q(v_1, v_2)$)

```

1  $DBS_{v_1} \leftarrow \text{SnapshotRDBSQuery}(v_1, G, P);$  // DBS objects when  $q$  is at  $v_1$ 
2  $DBS_{v_2} \leftarrow \text{SnapshotRDBSQuery}(v_2, G, P);$  // DBS objects when  $q$  is at  $v_2$ 
3  $X \leftarrow \emptyset;$  // positions where an object's direction changes
4 forall  $p \in DBS_{v_1} \cup DBS_{v_2} - DBS_{v_1} \cap DBS_{v_2}$  do
5    $dist_{v_1, p} \leftarrow \text{Dist}(v_1, p);$  // the length of the shortest path from  $v_1$  to  $p$ 
6    $dist_{v_2, p} \leftarrow \text{Dist}(v_2, p);$  // the length of the shortest path from  $v_2$  to  $p$ 
7    $x \leftarrow (|e^q| + |dist_{v_2, p} - dist_{v_1, p}|)/2;$  // the position where  $p$ 's direction changes
8    $X \leftarrow X \cup x;$  // Add  $x$  into  $X$ .
9 end for
10  $SG \leftarrow \text{GetSubSegments}(e^q, X);$  // sub-segments split by  $X$ 
11 forall  $e_i \in SG$  do
12    $DBS_{mid_i} \leftarrow \text{SnapshotRDBSQuery}(mid_i, G, P);$  // DBS objects when  $q$  is at the midpoint  $mid_i$  of  $e_i$ .
13    $DBS \leftarrow DBS \cup \{(DBS_{mid_i}, e_i)\};$  // Add  $\{(DBS_{mid_i}, e_i)\}$  to  $DBS$ .
14 end for
15 return  $DBS;$ 

```

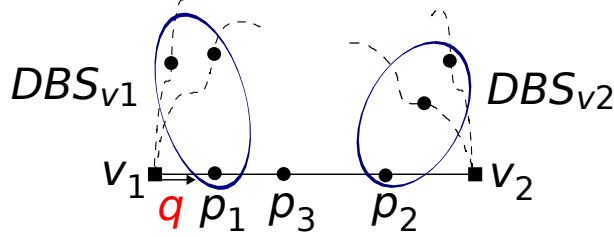
3.5.4 Discussion

Until now, we have discussed the case that the user is moving on an edge where no object exists. However, when the user is moving on an edge where objects exist, Algorithm 8 cannot find the correct answers. The reason is that Property 9 does not hold. Given a user moving on an edge $e(v_1, v_2)$ where an object p' exists, even if an object p , which is a DBS with respect to v_1 and v_2 but not on the edge $e(v_1, v_2)$, can be dominated by the object p' if it has a direction change moment on the edge.

To answer the queries in this special case, we divide the query edge $e(v_1, v_2)$ into three kinds of intervals using the objects $P_e = \{p_1, \dots, p_k\}$ on it. For every kind of intervals, we use different approaches to find the answers.

- Case 1: when the user moves from v_1 to p_1 , we obtain DBS_{v_1} w.r.t. v_1 and $DBS_{p_1} = \{p_1\}$ w.r.t. p_1 by regarding p_1 as a vertex. Next, we use Algorithm 8 to find the answers on the edge (v_1, p_1) .

- Case 2: when the user moves from p_i to p_{i+1} ($i \in [1, k - 1]$), $DBS = \{p_i, p_{i+1}\}$.
- Case 3: when the user moves from p_k to v_2 , we obtain DBS_{v_2} w.r.t. v_2 and $DBS_{p_k} = \{p_k\}$ w.r.t. p_k by regarding p_k as a vertex. Next, we use Algorithm 8 to find the answers on the edge (p_k, v_2) .

FIGURE 3.9: The special case for continuous \mathbb{R} -DBS queries

Example 18. Fig. 3.9 shows an example. The user q is moving on the edge $e(v_1, v_2)$ where objects $P_e = \{p_1, p_2, p_3\}$ exist. By posing two snapshot queries at v_1 and v_2 , we could obtain the results DBS_{v_1} and DBS_{v_2} . The edge e could be divided into four intervals: (v_1, p_1) , (p_1, p_3) , (p_3, p_2) , and (p_2, v_2) . The interval (v_1, p_1) is in case 1, the intervals (p_1, p_3) and (p_3, p_2) are in case 2, and the interval (p_2, v_2) is in case 3. We should find the DBSs according to the respective approaches to the three cases. ■

3.6 Experiments

In this section, we first explain the detailed settings of the experimental study, and then report the experimental results of the \mathbb{R} -DBS queries.

3.6.1 Settings

We evaluate the performance of \mathbb{R} -DBS queries on both synthetic datasets and real datasets. When making the synthetic datasets, we adopt the map of Oldenburg (OL) with 6105 nodes and 7035 edges as the road network which is obtained from [62] and also used in [63]. Six object sets with different cardinalities⁴ are generated via randomly extracting the points from the line segments of the road network.

⁴The six datasets contain 1K, 2K, 3K, 4K, 5K, 6K objects, respectively.

The seven real datasets are extracted from the points of interest on California road network which are obtained from [64] and also used in [65]. The map of California (CA) contains 21048 nodes and 21693 edges. The seven datasets are the 824 hospitals (H), 899 towers (T), 1766 flats (F), 6531 parks (P), 7587 churches (C), 10902 schools (S), and 11055 locales (L).

All the algorithms are implemented in GNU C++ and the experiments are conducted on an Intel(R) Core2 Duo CPU 3.16 GHz PC (4.0 GB RAM) with a Fedora Linux 2.6.32.

3.6.2 Performances of Snapshot Queries

In the experiments of snapshot \mathbb{R} -DBS queries on the synthetic datasets, 200 queries are generated randomly on the road network OL, and the average performance under different object set cardinalities is reported in Fig. 3.10. As Fig. 3.10 (a) shows, the number of DBSs 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, as Fig. 3.10 (b) shows, the cardinality does not have a significant influence on the CPU cost of the naïve algorithm, while the CPU cost of the optimized algorithm decreases with the cardinality. The reason is that the termination condition of the optimized algorithm is more likely to be satisfied if there are more objects. Fig. 3.10 (b) also shows that the optimized algorithm is always faster than the naïve algorithm. The maximum speedup is three orders of magnitude, achieved when the dataset size is 5K.

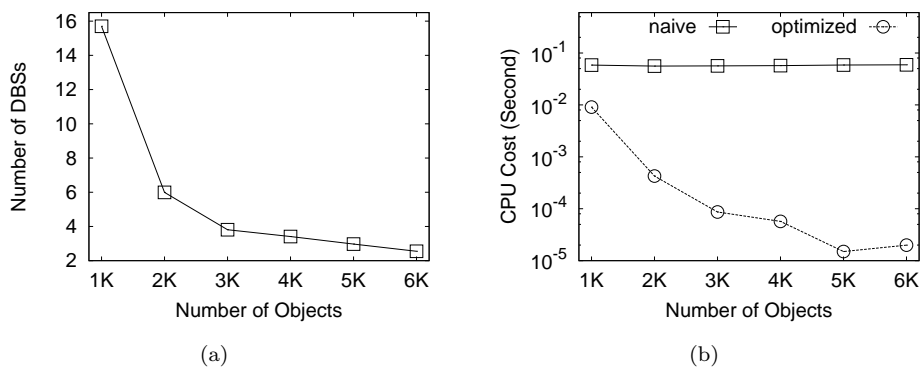


FIGURE 3.10: Performance of snapshot \mathbb{R} -DBS queries w.r.t. the number of objects

In the experiments of snapshot \mathbb{R} -DBS queries on the real datasets, 200 queries are generated randomly on the road network CA, and the average performance under different

object categories is reported in Fig. 3.11. Under the horizontal axis, every label shows the category name and the number of objects in this category. As Fig. 3.11 (a) shows, there are more DBSs in small datasets (i.e., H, T, and F) than in large datasets (i.e., P, C, S, and L). In Fig. 3.11 (b), the vertical axis is the CPU cost which is in the log scale. As observed, the optimized algorithm is always faster than the naïve algorithm. The maximum speedup is almost three orders of magnitude, achieved when the objects are the locales (L). The reasons of the observations are the same with the reasons of the experimental results on the synthetic datasets.

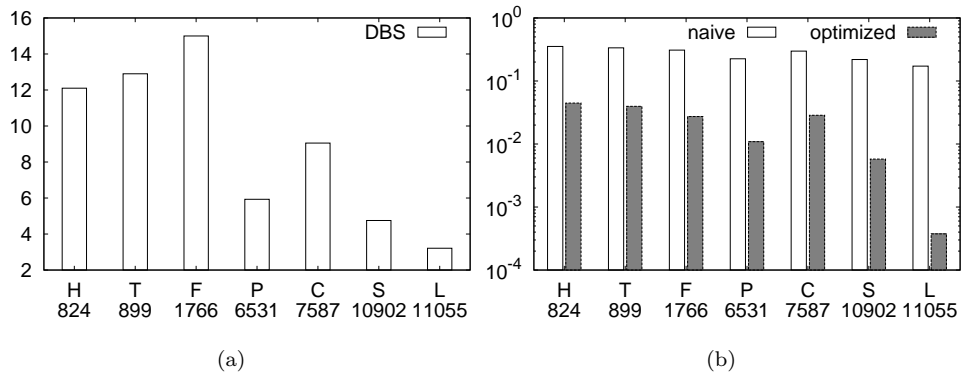


FIGURE 3.11: Performance of snapshot \mathbb{R} -DBS queries w.r.t. the object category

To sum up, we have the following findings after the experiments on the snapshot queries:

- The DBSs are few in number, i.e., less than 16 in average.
- The number of DBSs is influenced by the dataset size, i.e., smaller for large datasets.
- Both the naïve algorithm and the optimized algorithm can answer the queries promptly, i.e., less than one second (naïve algorithm) and less than 0.1 second (optimized algorithm).
- The optimized algorithm performs faster than the naïve algorithm.
- The efficiency of the naïve algorithm is not influenced by the dataset size.
- The efficiency of the optimized algorithm is influenced by the dataset size, i.e., higher for large datasets.

3.6.3 Performances of Continuous Queries

In the experiments of continuous queries on the synthetic datasets, we issue 200 queries that are the edges randomly selected from the road network OL. We report the average number of change moments in Fig. 3.12 (a) and the average CPU cost in Fig. 3.12 (b). The horizontal axis represents the cardinalities of object set varying in the range of $\{1K, 2K, \dots, 6K\}$ and the vertical axis represents the number of change moments and the CPU cost, respectively. As Fig. 3.12 (a) shows, the number of change moments is smaller for small datasets than for big datasets. Fig. 3.12 (b) shows the CPU cost in the log scale. As observed, the CPU cost decreases with the dataset size. The main cost of a continuous query is the sum cost of two snapshot queries that are issued on the endpoints of the query edge. Since the cost of a snapshot query decreases when the dataset size increases, the cost of a continuous query also decreases with the size.

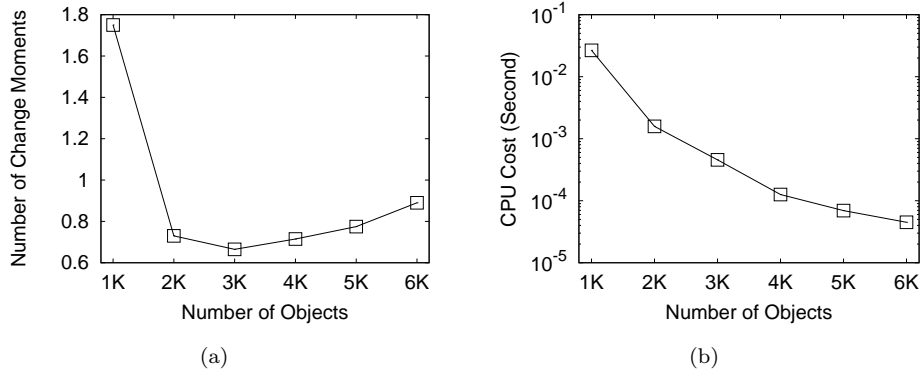


FIGURE 3.12: Performance of continuous \mathbb{R} -DBS queries w.r.t. the number of objects

In the experiments of continuous queries on the real datasets, we also issue 200 queries that are the edges randomly selected from the road network CA. We also report the average number of change moments and the average CPU cost in Fig. 3.13. The horizontal axis represents the different categories of objects and the vertical axis represents the number of change moments and the CPU cost in the log scale, respectively. As Fig. 3.13 (a) shows, the number of change moments is smaller for small datasets (i.e., H, T, and F) than for big datasets (i.e., P, C, S, and L). As Fig. 3.13 (b) shows, the CPU cost is larger for small datasets (i.e., H, T, and F) than for big datasets (i.e., P, C, S, and L). The reason is the same as that on the synthetic datasets.

To sum up, we have the following findings after the experiments on the continuous queries:

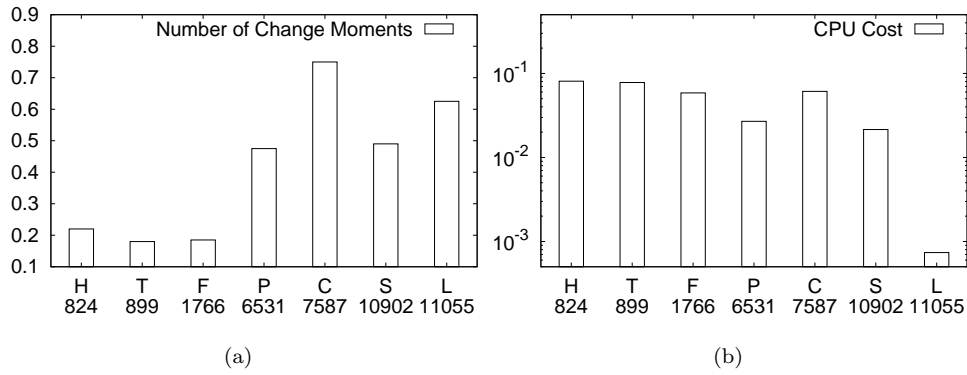


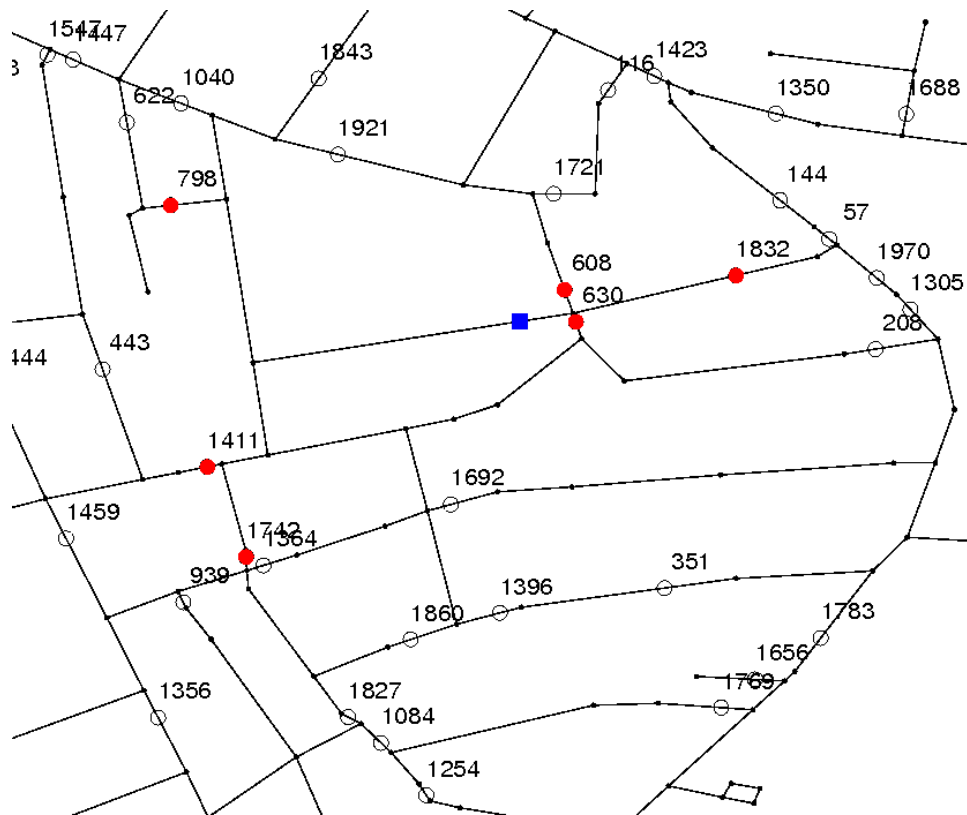
FIGURE 3.13: Performance of continuous \mathbb{R} -DBS queries w.r.t. the object category

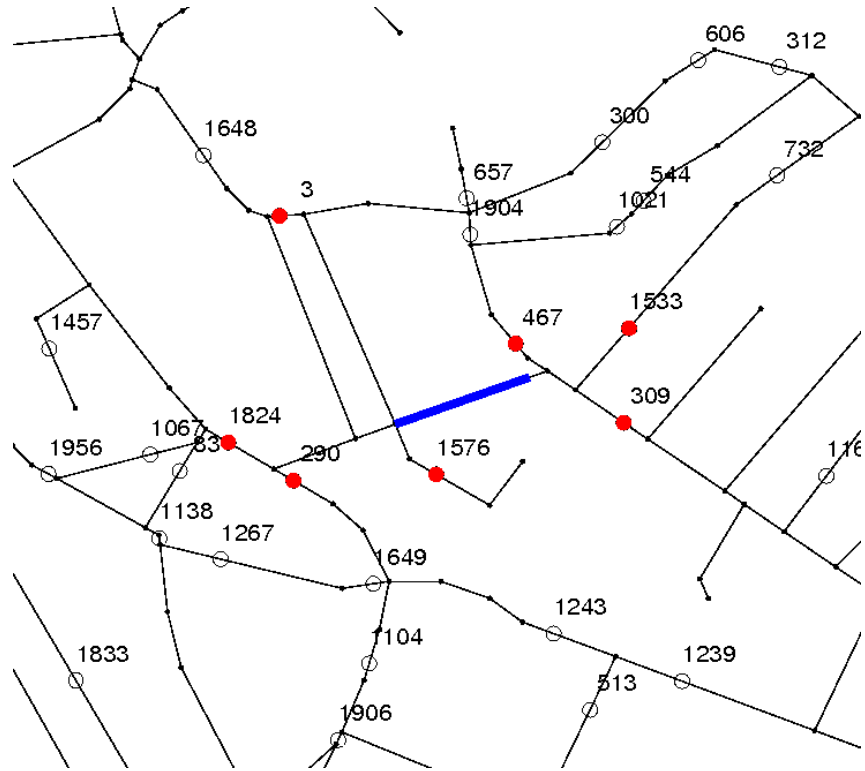
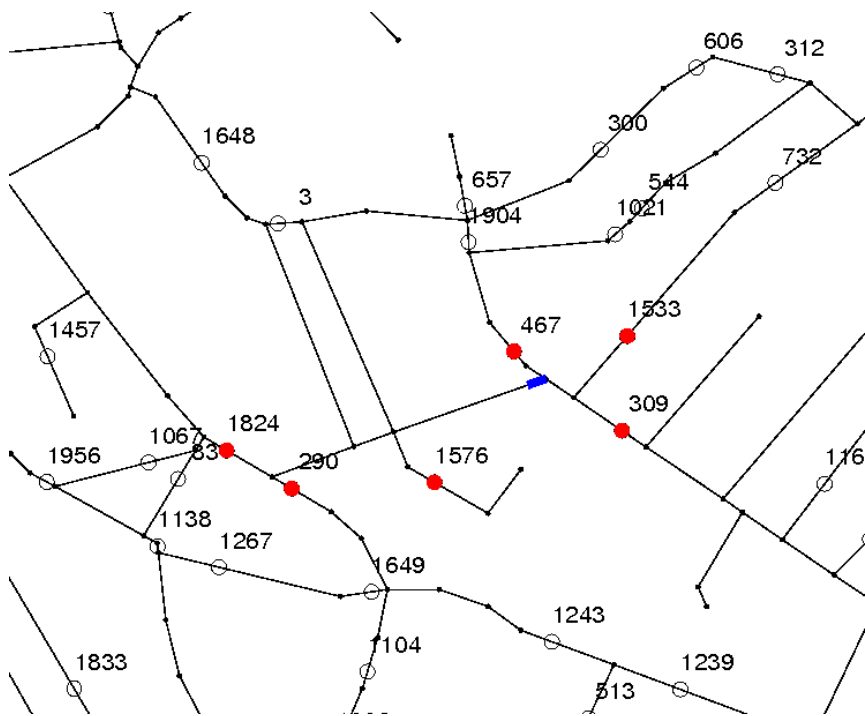
- The change moments are few in number, i.e., less than two in average.
- The number of change moment is influenced by the dataset size, i.e., smaller for small datasets.
- The proposed algorithm can answer the queries promptly, i.e., less than 0.1 second.
- The query cost is influenced by the dataset size, i.e., less for large datasets.

3.6.4 Screenshots

In order to illustrate the DBS objects in road network, Fig. 3.14 shows a snapshot query on the OL road with the object set containing 2000 objects (those represented by hollow points). The blue square point indicates where the snapshot query q is issued, and the red circle points are the DBS objects for q .

We also show screenshots of a continuous query at an edge in the OL road network in Fig. 3.15. When the user moves on the bold segment in Fig 3.15 (a), there are seven DBS objects (solid points). However, when the user moves on the bold segment in Fig 3.15 (b), there are six DBS objects excluding the object 3 in Fig 3.15 (a).

FIGURE 3.14: Screenshot of a snapshot \mathbb{R} -DBS query

(a) $(0, 106.105)$ (b) $(106.105, 120.439)$ FIGURE 3.15: Screenshots of a continuous \mathbb{R} -DBS query

Chapter 4

Combination Skyline Queries

4.1 Motivation

Given a set of objects \mathcal{O} where each $o_i \in \mathcal{O}$ has m -dimensional attributes $\mathcal{A} = \{A_1, \dots, A_m\}$, a *skyline query* [10] returns the objects that are not dominated by any other objects. An object *dominates* another object if it is not worse than the other in every attribute and strictly better than the other in at least one attribute. Skyline problems exist in various practical applications where trade-off decisions are made in order to optimize several important objectives. Consider an example in the financial field: An investor tends to buy the stocks that can minimize the commission costs and predicted risks. Therefore, the goal can be modeled as finding the skyline with minimum costs and minimum risks. Fig. 1.3 shows seven stock records with their costs (A_1 -axis) and risks (A_2 -axis). A , B , and D are the stocks that are not dominated by any others and hence constitute the skyline.

Skyline computation has received considerable attention from the database community [26–28] after the seminal paper [10], yet only a few studies explored the scenario where users are interested in combinations of objects instead of individuals. For the stock market example, assume that each portfolio consists of five stocks and its cost (risk) is the sum of costs (risks) of its components. Users may want to choose the portfolios which are not dominated by any others in order to minimize the total costs and the total risks.

4.2 Problem

In this paper we investigate the *combination skyline query problem*. Its goal is to find combinations that are not dominated by any other combinations. We focus on the combinations consisting of a fixed number of individual objects, and their attribute values are the aggregations of those from its members, as what we have illustrated in Example 2.

Few studies have focused on the combination skyline problem. [13] proposed a solution to find the top- k optimal combinations according to a user-defined preference order of attributes. However, it is difficult to define a user preference beforehand for some complicated decision making tasks. [12] tries to find the skyline combinations that are on the convex hull enclosing all the combinations, yet it will miss other many combinations on the skyline which provide meaningful results. In this paper, we present an efficient solution that constructs the whole combination skyline, within which the user may select a smaller subset of his interest [66–68].

A naïve way to answer constrained combination skyline query is to employ the existing skyline approaches [26–28] by regarding each enumerated combination as a single object. However, the huge number of enumerations renders them inapplicable for large datasets. In addition, some prevalent skyline approaches such as the BBS algorithm [28] uses index structures [29]; it means that we have to create a very large index for the combinations.

In this paper, we propose a *pattern-based pruning* (PBP) algorithm to solve the combination skyline problem by indexing individual objects rather than combinations in an R-tree. The PBP algorithm retrieves skyline combinations with a set of object-selecting patterns organized in a tree that represent the number of objects to be selected in each MBR. We exploit the attribute value ranges in the MBRs as well as search order, and develop two pruning strategies so as to avoid generating a large number of unpromising combinations. We also elaborate how to avoid repeated computations on expanding the same object-selecting patterns to combinations. The efficiency of the PBP algorithm is then evaluated with experiments.

4.3 Related Work

4.3.1 Combination Skyline Queries

To the best of our knowledge, there is no literature directly targeting the combination skyline problem. Two closely related topics are “top- k combinatorial skyline queries” [13] and “convex skyline objectsets” [12].

Top- k Combinatorial Skyline. [13] studied how to find top- k optimal combinations according to a given preference order in the attributes. Their solution is to retrieve non-dominated combinations incrementally with respect to the preference until the best k results have been found. This approach relies on the preference order of attributes and the limited number (top- k) of combinations queried. Both the preference order and the top- k limitation may largely reduce the exponential search space for combinations. However, in our problem there is neither preference order nor the top- k limitation. Consequently, their approach cannot solve our problem easily and efficiently. Additionally, in practice it is difficult for the system or a user to decide a reasonable preference order. This fact will narrow down the applications of [13].

Convex Skyline Objectsets. [12] studied the “convex skyline objectset” problem. It is known that the points on the lower (upper) convex hull, denoted as \mathcal{CH} , is a subset of the points on the skyline, denoted as \mathcal{SKY} . Every point in \mathcal{CH} can minimize (maximize) a corresponding linear scoring function on attributes, while every point in \mathcal{SKY} can minimize (maximize) a corresponding monotonic scoring function [10]. [12] aimed at retrieving the combinations in \mathcal{CH} , however, we focus on retrieving the combinations in $\mathcal{CH} \subseteq \mathcal{SKY}$. Since their approach relies on the properties of the convex hull, it cannot extend easily to solve our problem.

4.3.2 Other Combination Queries

There are some other works [69, 70] focusing on the combination selection problem but related to our work weakly.

Maximal Combination [69] studied how to select “maximal combinations”. A combination is “maximal” if it exceeds the specified constraint by adding any new object.

Finally, the k most representative maximal combinations, which contain objects with high diversities, are presented to the user. In their problem, the objects only have one attribute, in contrast to our multiple attribute problem. The approach for single attribute optimization problem is different from the approach for multiple attributes optimization problem. Thus, our problem cannot be solved by simple extensions of their approach.

Top- k Profitable Products [70] studied the problem to construct k profitable products from a set of new products that are not dominated by the products in the existing market. They constructed non-dominated products by assigning prices to the new products that are not given beforehand like the existing products. Our problem is very different from theirs in two aspects. First, they concern whether a single product is dominated or not, while we concern whether a combination of product is dominated or not. Second, there exist unfixed attribute values (prices) in their problem, while all the attribute values are fixed.

MOCO Problem Outside of the database field, the most studies relevant to our problem is the *multi-objective combinatorial optimization* (MOCO) problem [71]. The goal is to find subsets of objects aiming at optimizing multiple objective functions and complying with a set of constraints. Most approaches for the MOCO problem essentially convert the multiple objectives to one single objective and find one best answer numerically. Such numerical approaches are not good at handling large scale datasets in databases. Furthermore, our problem aims at retrieving optimal combinations without making a trade-off of multiple objectives by some score functions. For these reasons above, we cannot use the existing MOCO approaches to solve our problem in databases.

0-1 MOKP Problem Considering the combinatorial structure, which is the way of aggregating objects, our problem is similar to the *0-1 multi-objective knapsack problem* (0-1 MOKP) [72]. Essentially, the 0-1 MOKP is a special case included in our combination skyline problems. Similar to our problem, 0-1 MOKP has multiple optimization attributes (profits). However, the difference is that 0-1 MOKP has only one constraint attribute (weight), the number of which can be one or more in our problem. Additionally, our constraints limit the ranges of values, but the 0-1 MOKP constraint limits the upper bound of the value. Therefore, 0-1 MOKP is a special case of our problem. Thus, our algorithms can solve 0-1 MOKP, however, the algorithms for 0-1 MOKP cannot fully support our problem. Using the indexes constructed beforehand, our algorithms can handle the 0-1

MOKP on a large amount of objects, which cannot be handled efficiently using MOKP algorithms. The MOKP algorithms are usually dynamic programming approaches for exact answer and greedy algorithm for approximate answer. In addition, when the query changes or there are updates in the dataset, our algorithm does not need to start from scratch to scan the whole database.

0-1 MMKP Problem Another related problem is the *0-1 multi-dimensional multiple choice problem* (0-1 MMKP) [73]. Given several groups of objects, it is to select exactly one object from each group to maximize the total profit subject to several constraints. 0-1 MMKP can be solved approximately by heuristic algorithms [73]. Like 0-1 MOKP, 0-1 MMKP is also a special case included in our combination skyline problems. Similar to our problem, 0-1 MMKP has multiple upper bound constraints. However, 0-1 MMKP has only one optimization objective, the number of which can be two or more in our problem. Consequently, algorithms for 0-1 MMKP cannot fully support our problem. However, our algorithm can efficiently solve 0-1 MMKP on a large amount of objects in databases.

4.3.3 Quoted Work

In our algorithms, we retrieve skyline combinations based on the R-tree structure [29], which is a proven and widely-used index to handle multi-dimensional data. In order to prune the search space for constrained combination skyline queries, which is introduced in Section 4.7.2, we propose the constraint pruning strategy in which the *forward checking* approach is employed. The *forward checking* approach is a common solution for the *constraint satisfaction problems* (CSP) [74] and widely used in database field (e.g., [75]).

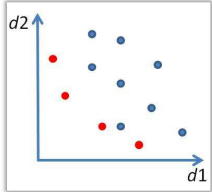
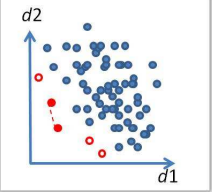
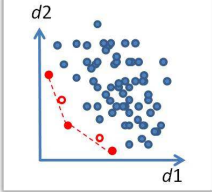
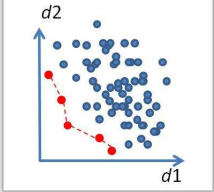
Additionally, our approach of selecting skyline combinations is inspired by the BBS algorithm [28], which traverses the R-tree using a priority queue with the key as the sums of the attribute values of each object.

4.3.4 Summary

Table 4.1 summarizes the related work of combination skyline queries and points out the position of our work ([76] and [77]). There are a lot of existing work focusing on retrieving individual objects that are on the skyline ([10]). There are two papers focusing on the combination skyline problem ([12] and [13]). However, both of them can only find specific

combinations on the skyline. Our work is the first study that can find all the combination on the skyline ([76] and [77]).

TABLE 4.1: Related work of combination skyline queries

individual skyline query	combination skyline query		
	preference	convex	complete
[10], etc.	[13]	[12]	[76] and [77]
			

4.4 Preliminaries

Given a set of objects \mathcal{O} with m attributes in the attribute set \mathcal{A} , a k -item combination c is made up of k objects selected from \mathcal{O} , denoted $c = \{o_1, \dots, o_k\}$. Each attribute value of c is given by the formula below

$$c.A_j = f_j(o_1.A_j, \dots, o_k.A_j), \quad (4.1)$$

where f_j is a monotonic aggregate function that takes k parameters and returns a single value. For the sake of simplicity, in this paper we consider that the monotonic scoring function returns the sum of these values; i.e.,

$$c.A_j = \sum_{i=1}^k o_i.A_j, \quad (4.2)$$

though our algorithms can be applied on any monotonic aggregate function.

Definition 12 (Dominance Relationship). A combination c dominates another combination c' , denoted $c \prec c'$, if c is not larger than c' in all the attributes and is smaller than c' in at least one attribute; formally, $c.A_j \leq c'.A_j$ ($\forall A_j \in \mathcal{A}$) and $c.A_t < c'.A_t$ ($\exists A_t \in \mathcal{A}$).

Problem 1 (Combination Skyline Problem). Given a dataset \mathcal{O} and an item number k , the combination skyline problem $CSKY(\mathcal{O}, k)$ is to find the k -item combinations that are not dominated by any other combinations.

Non-dominated combinations are also called *skyline combinations*. The combination skyline query in Example 2 can be formalized as $CSKY(\{A, \dots, G\}, 3)$ and the result set is $\{ABC, ABD, BCD\}$. We use the term “cardinality” to denote the item number k if there is no ambiguity. In this paper, we consider the case that $k \geq 2$ because the case that $k = 1$ reduces to the original skyline query [10].

In order to solve the combination skyline problem, a naïve approach is to regard the combinations as “objects” and select the optimal ones using existing skyline algorithms. However, these algorithms retrieve optimal objects based on either presorting or indexing objects beforehand. It means that before using such an algorithm we have to enumerate all possible combinations. Due to the explosive number of combinations generated, the naïve approach is inapplicable for large data sets. We choose the BBS algorithm [28] as the baseline algorithm for comparison, and our experiment shows that even for a set of 200 objects and a cardinality of three, it requires an index nearly one gigabyte and spends thousands of seconds on computing the skyline.

4.5 PBP Algorithm

Unlike the baseline approach, we propose a *pattern-based pruning* (PBP) algorithm based on an index on single objects rather than an index on combinations. We choose to index objects with an R-tree [29] as it is proven to be efficient for organizing multi-dimensional data. In order to make combinations, we use a set of *object-selecting patterns* to indicate the number of objects to be selected within each MBR in the R-tree. The object selecting patterns are organized in a *pattern tree*. We retrieve skyline combinations in the order arranged by a *pattern tree* that corresponds to the R-tree.

4.5.1 Object-Selecting Pattern

An R-tree is a data structure that hierarchically groups nearby multi-dimensional objects and encloses them by minimum bounding rectangles (MBRs). Our idea is to create combinations by selecting objects from the MBRs. The way is to select k_i objects from each MBR $r_i \in R$ and to make the total number of selected objects equal to k . Each k_i is limited in the range of $[0, \min(k, |obj(r_i)|)]$, where $obj(r_i)$ denotes the set of objects enclosed by r_i . An *object-selecting pattern* is defined formally below.

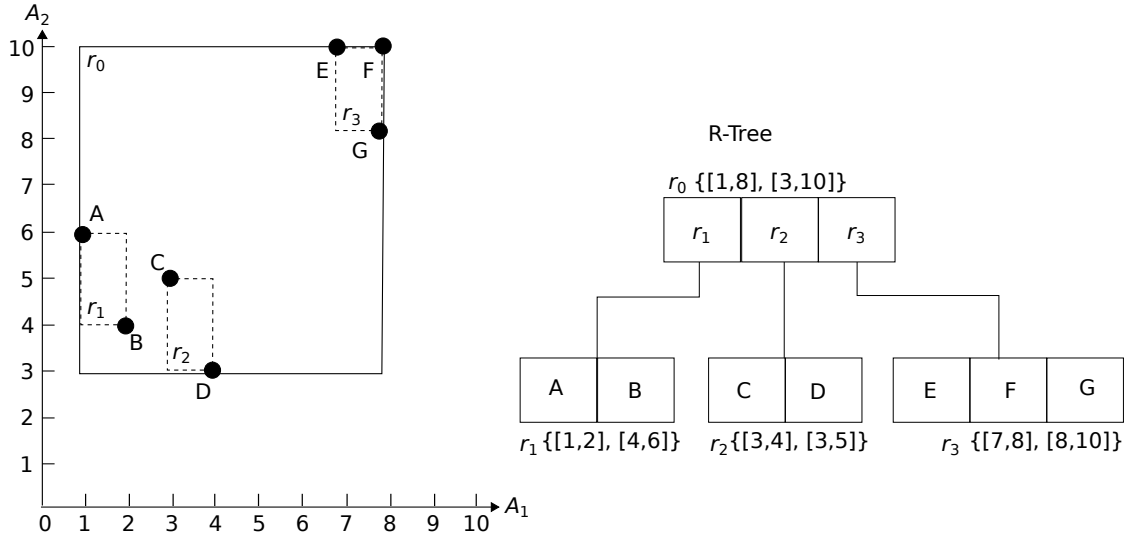


FIGURE 4.1: Object layout and R-tree

Definition 13 (Object-Selecting Pattern). Given a cardinality k and a set of MBRs R , an object-selecting pattern p is $\{(r_i, k_i) \mid r_i \in R, k_i \in [0, \min(k, |obj(r_i)|)]\}$ subject to $\sum_{i=1}^{|R|} k_i = k$. In addition, each MBR in R appears exactly once in the pattern p ; i.e., $r_i \neq r_j$ ($i \neq j$).

We call the pairs (r_i, k_i) *rules* that constitute a pattern. By Definition 13, a *rule* (r_i, k_i) is to select k_i objects from the MBR r_i .

The attribute values of the combinations obtained from a pattern are within $[\sum_{i=1}^{|R|} r_i.A_j^\perp \cdot k_i, \sum_{i=1}^{|R|} r_i.A_j^\top \cdot k_i]$ ($A_j \in \mathcal{A}$), because we can infer attribute value ranges for the combinations formed using the rule (r_i, k_i) as $[r_i.A_j^\perp \cdot k, r_i.A_j^\top \cdot k]$ ($A_j \in \mathcal{A}$), where $r_i.A_j^\perp$ and $r_i.A_j^\top$ are the values of the bottom left and top right corners of r_i .

Example 19. Fig. 4.1 shows the R-tree that indexes the objects in Example 1.3. In order to make 3-item combinations, one of the patterns is $\{(r_1, 2), (r_2, 1), (r_3, 0)\}$, consisting of three rules. Rule $(r_1, 2)$ means to select two objects from MBR r_1 , and rule $(r_2, 1)$ means to select one object from MBR r_2 . Thus, the pattern can generate the set of combinations $\{ABC, ABD\}$ that contains two combinations in total. With the boundaries of the three MBRs, we can limit the attribute values of the generated combinations within $[5, 8]$ for A_1 and $[11, 17]$ for A_2 .

Consider a rule (r, k) . If r is a leaf node of the R-tree, we can scan the objects contained and form combinations of size k . If r is an internal node, we need to expand it to child MBRs, and this will yield a group of patterns that select objects from r 's child MBRs with

the total number of objects summing up to k . We call such patterns the *child patterns* of the rule (r, k) .

Definition 14 (Child Patterns of a Rule). A child pattern of a rule (r, k) is a pattern that selects k objects from all of r 's child MBRs, formally $cp = \{(r_i, k_i) | r_i \in R, k_i \in [0, \min(k, |obj(r_i)|)]\}$ subject to $\sum_{i=1}^{|R|} k_i = k$ where R is the set of the child MBRs of r .

Note that all the child patterns of rule (r, k) share the same set of MBRs, but differ in the quantities of selected objects k_i . In the R-tree shown in Fig. 4.1, the node r_0 has three child MBRs $\{r_1, r_2, r_3\}$. Thus, patterns $\{(r_1, 2), (r_2, 1), (r_3, 0)\}$, $\{(r_1, 2), (r_2, 0), (r_3, 1)\}$, and so on are the child patterns of the rule $(r_0, 3)$, which share the same set of r_i 's but differ in k_i 's.

Algorithm 9: ExpandPattern (p)

Input : A pattern p represented in a set of (r_i, k_i) 's.

Output: The set of child patterns of p .

```

1  $P \leftarrow e;$  // assume  $e$  is the identity element of Cartesian product
2 for each  $(r_i, k_i) \in p$  do
3    $P' \leftarrow$  the child patterns of  $(r_i, k_i);$ 
4    $P \leftarrow P \times P';$ 
5 end for
6 return  $P;$ 

```

Similarly, a pattern can be expanded to a set of child patterns. For each rule in the pattern, we expand the rule to its child patterns, and perform an n -ary Cartesian product on all these child patterns. Algorithm 9 presents the pseudo-code of the procedure.

Starting with the root node r_0 in the R-tree and its corresponding root pattern $p_0 = \{(r_0, k)\}$, if we traverse the R-tree with a breadth-first search, and expand each corresponding pattern using its child patterns, we can obtain all possible combinations at the leaf level. Accordingly, the patterns expanded constitute a *pattern tree*. Example 20 shows the procedure of constructing a pattern tree with respect to the R-tree in Fig. 4.1.

Example 20. A pattern tree corresponding to the R-tree in Fig. 4.1 is shown in Fig. 4.2. The root pattern is $p_0 = \{(r_0, 3)\}$ where 3 is the required cardinality. Since pattern p_0 only has a single rule $(r_0, 3)$, the eight child patterns of $(r_0, 3)$, $\{p_1, \dots, p_8\}$, are also the child patterns of p_0 . Next, we expand the patterns at the second level of the pattern tree. Consider pattern $p_1 = \{(r_1, 2), (r_2, 1), (r_3, 0)\}$ that contains three rules $(r_1, 2)$, $(r_2, 1)$ and $(r_3, 0)$. Rule $(r_1, 2)$ has one child pattern $\{AB\}$ and rule $(r_2, 1)$ has two child patterns

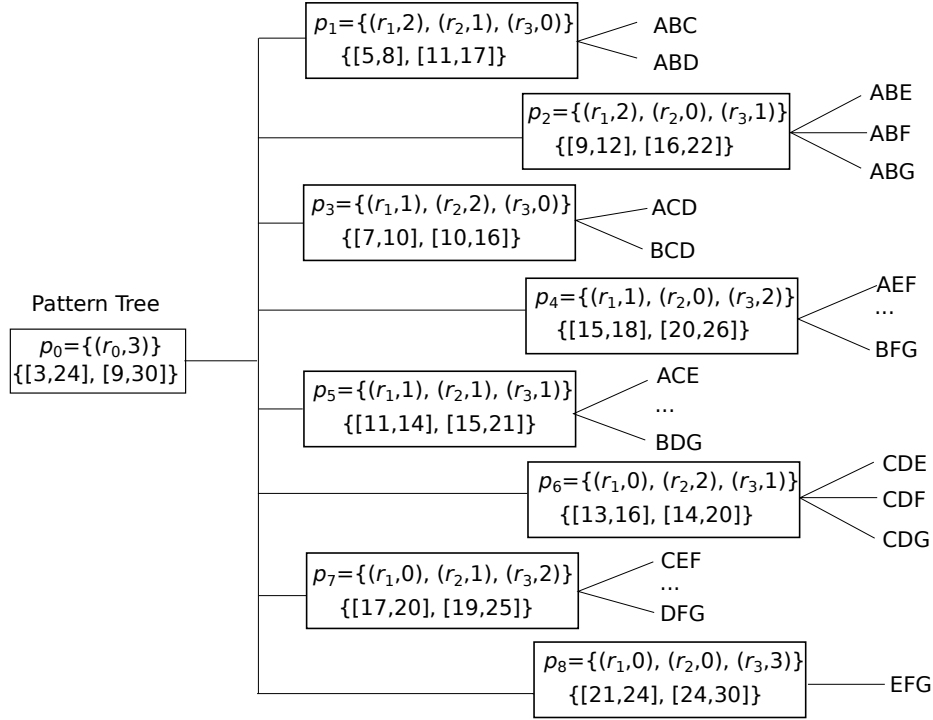


FIGURE 4.2: Pattern tree

$\{C, D\}$ and hence the child patterns of p_1 is $\{AB\} \times \{C, D\} = \{ABC, ABD\}$. Since these child patterns contain objects rather than MBRs, we also call them child combinations.

4.5.2 Basic PBP Algorithm

Following the pattern tree, we design a basic PBP algorithm (Algorithm 10). It takes as an input the set of objects, and first builds an R-tree on the objects. Starting with the root node r_0 and the pattern $\{(r_0, k)\}$, we traverse the R-tree in a top-down fashion. Note that the pattern tree is not materialized in the algorithm. Instead, we use a queue Q to capture the patterns generated while traversing the pattern tree. Each pattern is expanded to its child patterns (line 7) if the nodes in the pattern are internal nodes; otherwise leaf nodes are reached and hence we can make combinations in the MBRs according to the pattern (line 12). The combinations are then checked for dominance relationship with the candidate skyline combination found so far and vice versa (line 13). The candidates not dominated by any combinations are returned as the answer after processing all the expanded patterns.

Compared with the baseline algorithm, the basic PBP algorithm reduces the space consumption by building an R-tree on single objects. However, it suffers from the huge

Algorithm 10: BasicPBP (T, k)

Input : T is the R-tree built on \mathcal{O} ; k is the cardinality.**Output**: The skyline combination set $S = CSKY(\mathcal{O}, k)$.

```

1  $S \leftarrow \emptyset$ ;
2  $r_0 \leftarrow$  the root node of  $T$ ;
3  $Q \leftarrow \{(r_0, k)\}$ ;
4 while  $Q \neq \emptyset$  do
5    $p \leftarrow Q.pop()$ ;
6   if the MBRs in  $p$  are internal nodes then
7      $P \leftarrow \text{ExpandPattern}(p)$ ;
8     for each  $p' \in P$  do
9        $Q.push(p')$ ;
10    end for
11  else
12     $C \leftarrow$  generate combinations with  $p$ ;
13     $S \leftarrow \text{Skyline}(S \cup C)$ ;
14  end if
15 end while
16 return  $S$ ;
```

number of patterns. Even for a rule (r, k) , the number of child patterns is $\binom{h+k-1}{h-1}$ if r has h child MBRs. We will discuss how to reduce this number and consider only promising child patterns in the following section.

4.6 Optimized PBP Algorithm

In a pattern tree, we can decide which patterns should be expanded and which patterns should not be expanded. For example, in the pattern tree shown in Fig. 4.2, the combinations following pattern p_4 must be dominated by the combinations following pattern p_1 . Thus, we can prune pattern p_4 without further expanding. Another intuition is that if the combinations from a pattern are guaranteed to be dominated by the current skyline combinations, the pattern can be pruned as well. We call these two scenarios *pattern-pattern pruning* and *pattern-combination pruning*. We also observe the existence of multiple expansions for same patterns in the pattern tree. In the rest of this section, we will study the two pruning techniques and how to avoid multiple expansions as well.

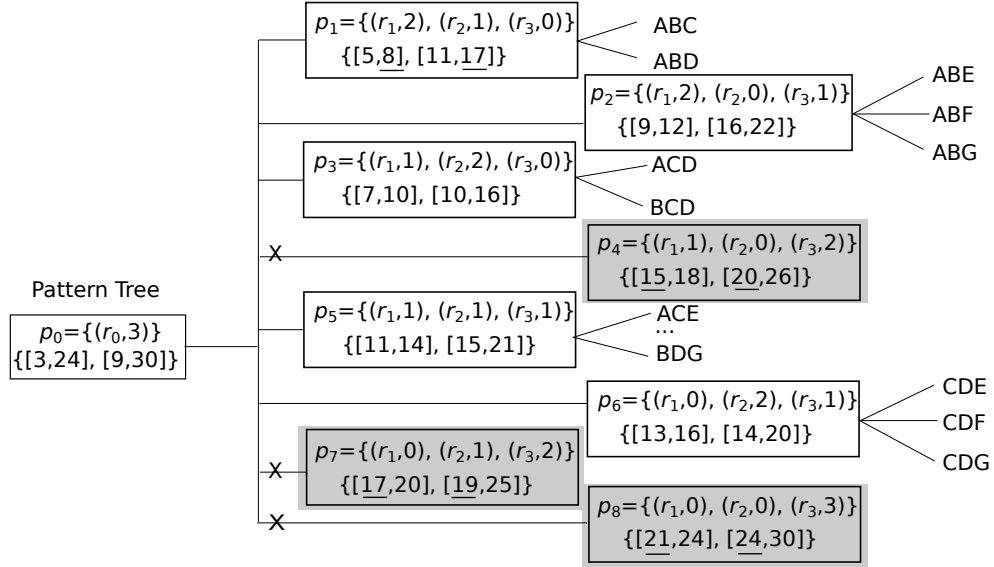


FIGURE 4.3: Pattern-pattern pruning (grey patterns are pruned using Theorem 1)

4.6.1 Pattern-Pattern Pruning

Patterns can be pruned safely without expanding if they will generate combinations that are guaranteed to be dominated by others. We first define the dominance relationship between patterns and capture the idea in Theorem 1.

Definition 15 (Pattern Dominance). A pattern p dominates another pattern p' if $p.A_j^\top \leq p'.A_j^\perp$ ($\forall A_j \in \mathcal{A}$) and $p.A_t^\top < p'.A_t^\perp$ ($\exists A_t \in \mathcal{A}$), and is denoted as $p \prec p'$.

Theorem 1. A pattern p' cannot generate skyline combinations if it is dominated by another pattern p .

Proof 1. Any combination c' following the pattern p' has values $c'.A_j \geq p'.A_j^\perp$ ($\forall A_j \in \mathcal{A}$). Any combination c following the pattern p has values $c.A_j \leq p.A_j^\top$ ($\forall A_j \in \mathcal{A}$). If $p \prec p'$, $c.A_j \leq c'.A_j$ ($\forall A_j \in \mathcal{A}$) and $c.A_t < c'.A_t$ ($\exists A_t \in \mathcal{A}$). Consequently, c' is not a skyline combination because $c \prec c'$.

Example 21. Consider the eight patterns $\{p_1, \dots, p_8\}$ at the second level of the pattern tree shown in Fig. 4.3. Pattern p_1 with upper bounds (8, 17) can dominate pattern p_4 with lower bounds (15, 20), p_7 with lower bounds (17, 19), and pattern p_8 with lower bounds (21, 24). Thus, the three patterns p_4 , p_7 and p_8 can be safely pruned according to Theorem 1.

4.6.2 Pattern-Combination Pruning

Starting with the root pattern, we expand patterns to child patterns until obtaining combinations at the leaf level. Unlike BasicPBP in Algorithm 10 that traverses patterns in a breadth-first way, we can use a priority queue to implement the expansion process in a key-order way. Inspired by the BBS algorithm [28], the keys for the priority queue are the *mindists* of the patterns, and we process the patterns in the priority queue following the increasing order of their keys.

Definition 16 (Mindist of a Pattern). The mindist p , denoted as $p.mindist$, is the sum of its lower bounds in all the attributes \mathcal{A} , namely, $p.mindist = \sum_{j=1}^{|\mathcal{A}|} p.A_j^\perp$ ($A_j \in \mathcal{A}$).

Like BBS, we also insert the generated combinations to a priority queue. In the same way, the mindist of a combination can be defined as the sum of values in \mathcal{A} , namely, $b.mindist = \sum_{j=1}^{|\mathcal{A}|} b.A_j$ ($A_j \in \mathcal{A}$).

Theorem 2. A combination c cannot be dominated by any combinations generated from a pattern p' if $c.mindist < p'.mindist$.

Proof 2. Assume that the combination c can be dominated by c' which is generated from p' . According to Definition 12, $c'.A_j \leq c.A_j$ ($\forall A_j \in \mathcal{A}$) and $c'.A_t < c.A_t$ ($\exists A_t \in \mathcal{A}$). It means that $c'.mindist < c.mindist$ because $c'.mindist = \sum_{j=1}^{|\mathcal{A}|} c'.A_j$ and $c.mindist = \sum_{i=1}^{|\mathcal{A}|} c.A_i$. On the other hand, $p'.mindist \leq c'.mindist$ because $\sum_{j=1}^{|\mathcal{A}|} p'.A_j^\perp \leq \sum_{i=1}^{|\mathcal{A}|} c'.A_i$. Consequently, the inequality $p'.mindist < c.mindist$ contradicts the condition $c.mindist < p'.mindist$, and thus Theorem 2 is proved.

The advantage of expanding patterns using a *mindist*-order priority queue is that when the top element is a combination, according to Theorem 2, it cannot be dominated by the combinations following the patterns behind it in the queue. It just needs comparisons with the skyline combinations already found in the result set $S = CSKY(\mathcal{O}, k)$. If it cannot be dominated by any combinations in S , it is a skyline combination and should be added into S . For the other case where the top element is a pattern, it should be discarded if it is dominated by any combinations in S ; otherwise, it should be expanded and its child patterns are pushed into the queue. The above process begins with the root pattern pushed into the queue and ends when the queue is empty. The final S is returned as the answers. Example 22 illustrates the process.

Priority Queue (Q)	Result S
$\leftarrow \langle p_0, 12 \rangle \langle p_1, 16 \rangle \langle p_3, 17 \rangle \langle p_2, 25 \rangle \langle p_5, 26 \rangle \langle p_6, 27 \rangle \leftarrow \dots$	\emptyset
$\leftarrow \langle p_1, 16 \rangle \langle p_3, 17 \rangle \langle p_2, 25 \rangle \langle p_5, 26 \rangle \langle p_6, 27 \rangle \langle ABD, 20 \rangle \langle ABC, 21 \rangle \leftarrow \dots$	\emptyset
$\leftarrow \langle p_3, 17 \rangle \langle ABD, 20 \rangle \langle ABC, 21 \rangle \langle p_2, 25 \rangle \langle p_5, 26 \rangle \langle p_6, 27 \rangle \langle BCD, 21 \rangle \langle ACD, 21 \rangle \leftarrow \dots$	\emptyset
$\leftarrow \langle ABD, 20 \rangle \langle ABC, 21 \rangle \langle BCD, 21 \rangle \langle ACD, 21 \rangle \langle p_2, 25 \rangle \langle p_5, 26 \rangle \langle p_6, 27 \rangle$	$\{ABD\}$
...	...
$\leftarrow \langle p_6, 27 \rangle$	$\{ABD, ABC, BCD\}$
\emptyset	$\{ABD, ABC, BCD\}$

FIGURE 4.4: Priority queue and query result

Example 22. Fig. 4.4 shows the process of the combination skyline queries. We initialize the priority queue Q as $\{\langle p_0, 12 \rangle\}$ where p_0 is the root pattern and 12 ($p_0.mindist$) is the key. Next, p_0 is popped and its child patterns $\{p_1, p_3, p_2, p_5, p_6\}$ are pushed into Q . Note that other three patterns are pruned according to Theorem 1. We pop the top one p_1 and push its expansions $\{ABD, ABC\}$ into Q . For the next top element pattern p_3 , we pop it and push its expansions $\{BCD, ACD\}$ into Q . Next, the top element is the combination ABD , which is popped and becomes the first result in $S = CSKY(\mathcal{O}, k)$. In the same way, we pop the top element and check whether it is dominated by the skyline combinations in S . If it is dominated, top element is discarded. Otherwise, its child patterns are pushed into Q . For example, when p_6 becomes the top element, it is dominated by $ABD \in S$. Thus, it is discarded. The process continues until the queue Q is empty and we obtain the final result set $\{ABD, ABC, BCD\}$. Fig. 4.5 shows the pattern tree after the pattern-combination pruning.

4.6.3 Pattern Expansion Reduction

Another problem with BasicPBP algorithm is that the same rules may appear in multiple patterns and thus may be expanded multiple times. In Fig. 4.2, among the child patterns

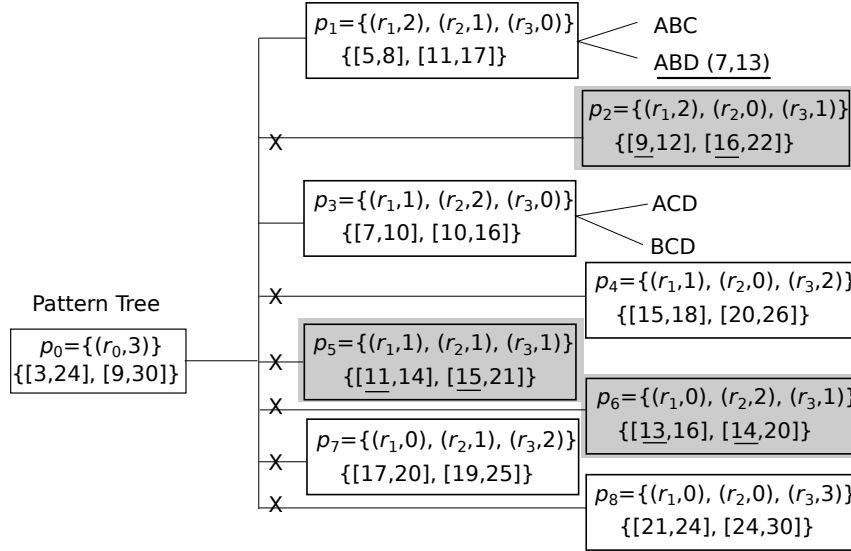


FIGURE 4.5: Pattern-combination pruning (grey patterns are pruned using Theorem 2 and the patterns beginning with \times are pruned using both Theorem 1 and Theorem 2)

expanded from the root pattern p_0 , patterns p_1 and p_2 share the same rule $(r_1, 2)$, and it will be expanded twice into the same set of child patterns.

Duplicate expansion is even worse as the algorithm goes deeper in the R-tree. An immediate solution to address this problem is to perform a lazy expansion if a rule is encountered multiple times. The intuition is that once the descendant patterns of the first occurrence reach the object combination level, the generated combinations are kept, and all the multiple occurrences of the rule can be replaced by the combinations when a dominance check is invoked. In order to keep the combinations for each rule encountered, we use a matrix M with MBRs as rows and cardinalities columns.

As the search order shown in Fig. 4.4, pattern p_1 comes before pattern p_2 in the priority queue. p_2 's child patterns will inherit the rule $(r_1, 2)$ from p_2 , but not expand the rule immediately. After all the descendant patterns of p_1 have been processed to create object combinations, the cells representing $(r_1, 2)$ and its descendants are filled with the combinations. When p_2 is expanded and reaches the object level, its component rule $(r_1, 2)$ is replaced by what we stored in the cell for dominance checking.¹

The above solution ensures no duplicate expansion of a rule in the algorithm. However, it is not space-efficient to record all the combinations for the rules encountered. Thanks

¹We assume the descendants of p_1 come before those of p_2 in the priority queue in this example. For the general case, once a descendant of $(r_1, 2)$ has produced object combinations, other patterns that contain the rule can avoid redundant computations.

	1	2	3	
r_0			①	① is filled when expanding p_0 . ② is filled when expanding p_1 . ③ is filled when expanding p_3 .
r_1	③	②		
r_2	②	③		
r_3				

FIGURE 4.6: Pattern expansion reduction matrix

to the following theorem, we are able to store only the *skyline* combinations instead for each cell in the matrix.

Theorem 3. If a skyline combination $c \in CSKY(\mathcal{O}, k)$ contains k' objects in an MBR r' , the combination consisting of the k' objects is a skyline combination of $obj(r')$ with cardinality k' .

Proof 3. Consider a skyline combination $c \in CSKY(\mathcal{O}, k)$ that contains k' objects in an MBR r' . Assume the k' objects are $o_1, \dots, o_{k'}$, and their combination is dominated by another combination $\{o'_1, \dots, o'_{k'}\}$ whose objects are also enclosed by r' . According to the monotonicity of the aggregate function,

$$c \setminus \{o_1, \dots, o_{k'}\} \cup \{o'_1, \dots, o'_{k'}\} \prec c.$$

It contradicts the assumption that c is a skyline combination of \mathcal{O} with cardinality k , and hence the theorem is proved.

Therefore, we only need to keep $CSKY(obj(r_i), k_i)$ for each $M[r_i][k_i]$. For a rule with a leaf MBR, we compute it with the objects inside. For a rule with an internal MBR, we compute $M[r_i][k_i]$ once all of its child patterns have been expanded to the object level, and store the skyline over the results obtained from the child patterns. Note that this skyline computation is a byproduct of generating combinations of size k and checking dominance, and thus we do not need to compute them separately. Example 23 shows the process of filling in the matrix M .

Example 23. According to the search order shown in Fig. 4.4, when expanding p_0 we fill $M[r_0][3]$ using the child patterns of $(r_0, 3)$. Next, we expand p_1 containing rules $(r_1, 2)$, $(r_2, 1)$, and $(r_3, 0)$. The corresponding cells $M[r_1][2]$ is filled with the combination $\{AB\}$ and $M[r_2][1]$ is filled with combinations $\{C, D\}$ that are not dominated each other. The

Cartesian join products $\{AB\} \times \{C, D\} = \{ABC, ABD\}$ are the child patterns of p_1 . The next pattern expanded is p_3 containing rules $(r_1, 1)$, $(r_2, 2)$, and $(r_3, 0)$. The corresponding cells $M[r_1][1] = \{A, B\}$ and $M[r_2][2] = \{CD\}$. The join products $\{A, B\} \times \{CD\} = \{ACD, BCD\}$ are the child patterns of p_3 .

Since pattern-pattern pruning keeps unpromising patterns from the priority queue, not all the cells in the matrix need to be filled. Considering the sparsity of the matrix, we implement it with a hash table with an (MBR, cardinality) pair as the key for each entry, and store the value as

- a set of skyline combinations, if all its child patterns have been expanded to the object level; and
- otherwise, a list of its child patterns.

Algorithm 11: ExpandPatternOpt (p)

Input : A pattern p represented in a set of (r_i, k_i) 's.

Output: The set of child patterns of p .

```

1  $P \leftarrow e$ ;           // assume  $e$  is the identity element of Cartesian product
2 for each  $(r_i, k_i) \in p$  do
3   switch  $M[r_i][k_i]$  do
4     case has not been initialized do
5        $P' \leftarrow$  the child patterns of  $(r_i, k_i)$ ;
6        $M[r_i][k_i] \leftarrow P'$ ;
7     end case
8     case is a list of child patterns do
9        $P' \leftarrow M[r_i][k_i]$ ;
10    end case
11    case is a set of skyline combinations do
12       $P' \leftarrow \{(r_i, k_i)\}$ ;           //  $(r_i, k_i)$  has been explored and replace with
13       $M[r_i][k_i]$  when reaching the object level
14    end case
15  endsw
16   $P \leftarrow P \times P'$ ;
17 return  $P$ ;

```

We design a new pattern expansion algorithm in Algorithm 11. It expands a rule under three different cases. If the rule is encountered for the first time, i.e., the cell in the matrix has not been initialized, we expand it to its child patterns, and fill the cell in the matrix with a list of the child patterns (line 5 and line 6). If the rule is encountered multiple times, but none of the patterns contains it have reached the object level so far, we expand

the rule with the stored list of child patterns (line 9). For the third case, as the object combinations for this rule have been seen before, we keep the rule intact until the patterns containing it reach the object level, and then it is replaced with the skyline combinations stored in the cell (line 12).

4.6.4 Complete Algorithm

Applying the three optimization techniques, we summarize the complete pattern-based pruning (CompletePBP) algorithm in Algorithm 12. The algorithm iteratively pops the top element p in the priority queue Q (line 5). The top element can be either a combination or a pattern. For a combination, we insert them into the final result set S after checking dominance with the current skyline combinations (line 8). For a pattern, if p is dominated by any skyline combinations found so far, we discard it using with pattern-combination pruning (line 11). Otherwise we expand it using the optimized pattern expansion algorithm (line 15). If the MBRs involved in P are internal nodes of the R-tree, we push the non-dominated child patterns to the queue Q (line 18), utilizing pattern-pattern pruning. Otherwise we generate combinations with the patterns in P , and update the matrix M to reduce pattern expansion (line 23–30). The algorithm terminates when the priority queue is empty.

4.7 Variations of PBP Algorithm

In this section, we discuss two variations of the combination skyline problem and extend our PBP algorithm to solve the two variations.

4.7.1 Incremental Combination Skyline

We first discuss the incremental combination skyline problem, as a user may want to increase the cardinality of combinations as he has seen the result of $CSKY(\mathcal{O}, k)$. The problem is defined as follows:

Problem 2 (Incremental Combination Skyline Query). An incremental combination skyline query $CSKY^+(\mathcal{O}, k + \Delta k)$ is to find $(k + \Delta k)$ -item skyline combinations based on an original query $CSKY(\mathcal{O}, k)$ that has already been answered.

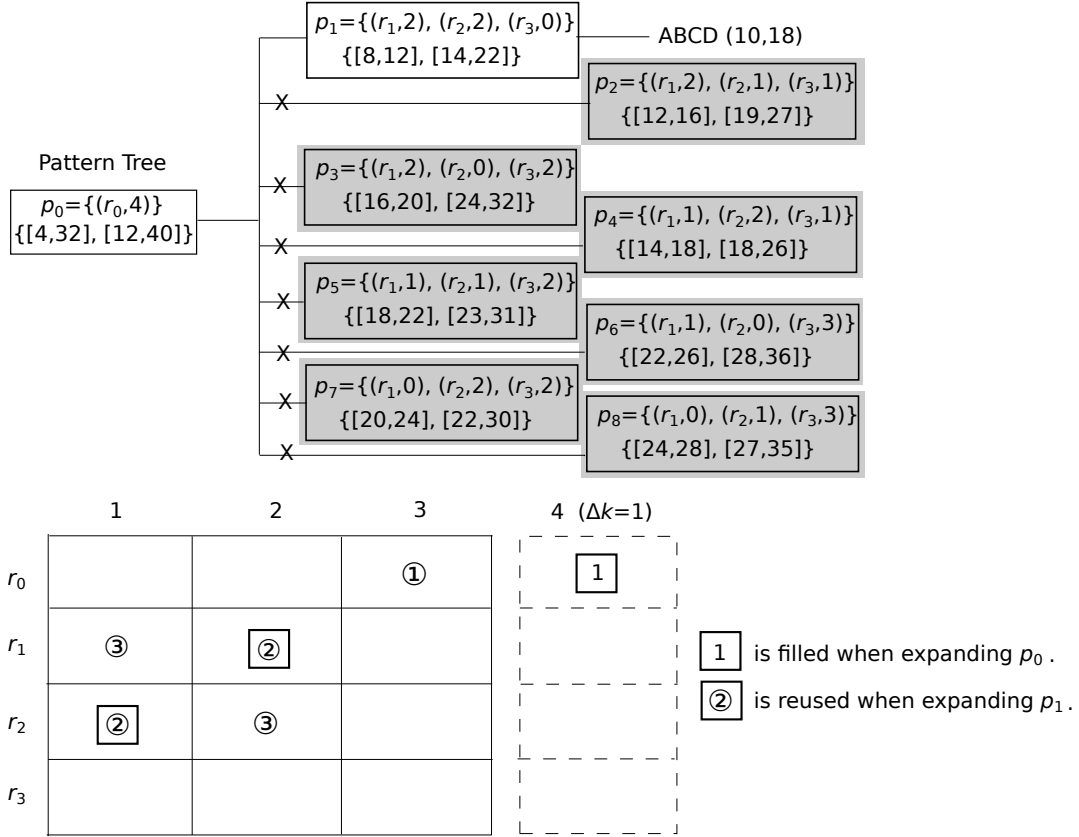


FIGURE 4.7: Incremental combination skyline query

The incremental query $CSKY^+$ searches for skyline combinations from the same dataset \mathcal{O} as the original query $CSKY$, so we can use the same R-tree for the original skyline query. Starting with the root $(r_0, k + \Delta k)$, the patterns are processed using the PBP algorithm. As the matrix M for duplicate expansion reduction has been filled when processing the original query, if not all of its cells, the contents can be utilized. When the child patterns of rule (r_i, k_i) are needed during expansion, we reuse the existing results in $M[r_i][k_i]$ if the cell was already calculated. In this way, the repeated calculations for the same cell can be avoided. Though Δk empty columns are appended to M at first, this is not space consuming as M is implemented in a hash table.

Example 24. Fig. 4.7 shows the pattern tree and the matrix M for the incremental query $CSKY^+(\mathcal{O}, 4)$ based on the original query $CSKY(\mathcal{O}, 3)$ with $\Delta k = 1$. The circled numbers in the matrix indicate what we have processed when processing $CSKY(\mathcal{O}, 3)$, and the quads indicate what we are going to fill for processing the incremental query. We start with expanding the root pattern $p_0 = \{(r_0, 4)\}$, and an empty column is appended to M . Pattern p_0 has three child patterns that survive pattern-pattern pruning: p_1 , p_2 , and p_4 , sorted by the increasing mindist order. Next we expand pattern p_1 consisting of three

rules $(r_1, 2)$, $(r_2, 2)$, and $(r_3, 0)$. Both cells $M[r_1][2]$ and $M[r_2][2]$ were already calculated when answering the original query. By computing the Cartesian product, a combination $ABCD$ is obtained for p_1 . Since $ABCD$ is the first combination found, it is a skyline combination and we put it into the result set. As the next top elements p_2 and p_4 are dominated by the combination $ABCD$, the process terminates when the queue is empty and the final result $CSKY^+ = \{ABCD\}$ is returned.

4.7.2 Constrained Combination Skyline

For a combination skyline query, we retrieve optimal combinations that have values as small as possible with respect to all the attributes $\forall A_j \in \mathcal{A}$. In practice, however, not all the attributes are being concerned and there are even some range constraints on the concerned attributes. We define a *constrained combination skyline query* that retrieves for optimal combinations with respect to a set of concerned attributes $\mathcal{A}^* \subseteq \mathcal{A}$ subject to range constraints V_j on attribute $A_j \in \mathcal{A}^*$.

Problem 3 (Constrained Combination Skyline Query). A constrained combination skyline query $CSKY^*$ is defined as

$$CSKY^* = \{\mathcal{O}, k, \langle A_1, V_1 \rangle, \dots, \langle A_{m^*}, V_{m^*} \rangle\}, \quad (4.3)$$

where $1 \leq m^* \leq m$ and $\{A_1, \dots, A_{m^*}\} \subseteq \mathcal{A}$. We call $\mathcal{A}^* = \{A_1, \dots, A_{m^*}\}$ constraint attributes. $V_j = [v_j^\perp, v_j^\top]$ is a range constraint for attribute A_j ($A_j \in \mathcal{A}^*$). If we do not specify a range constraint on attribute A_j , we set an infinite range $V_j = [-\infty, \infty]$.

The combination skyline query defined in Problem 1 is subsumed in the constrained combination skyline query $CSKY^*$ because $CSKY$ is a special case of $CSKY^*$ when $\mathcal{A}^* = \mathcal{A}$, and subject to $V_j = [-\infty, \infty]$ ($\forall A_j \in \mathcal{A}^*$).

Example 25. Let us consider an example of a constrained combination skyline query, $CSKY^*(\{A, \dots, G\}, 3, \langle A_1, [-\infty, \infty] \rangle, \langle A_2, [5, 13] \rangle)$. As Fig. 4.8 shows, since the combinations $\{ABD, ACD, BCD\}$ are within the range $[5, 13]$ on A_2 , they are candidates for skyline combinations. Among the three candidates, combination ACD is dominated by combination ABD . Thus, the non-dominated combinations $\{ABD, BCD\}$ are the skyline combinations for query $CSKY^*$.

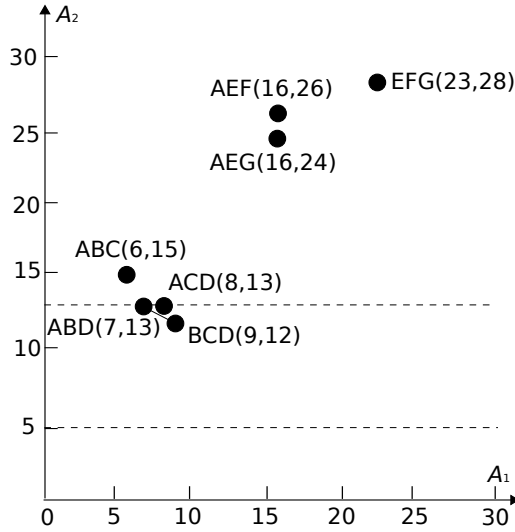


FIGURE 4.8: Constrained combination skyline

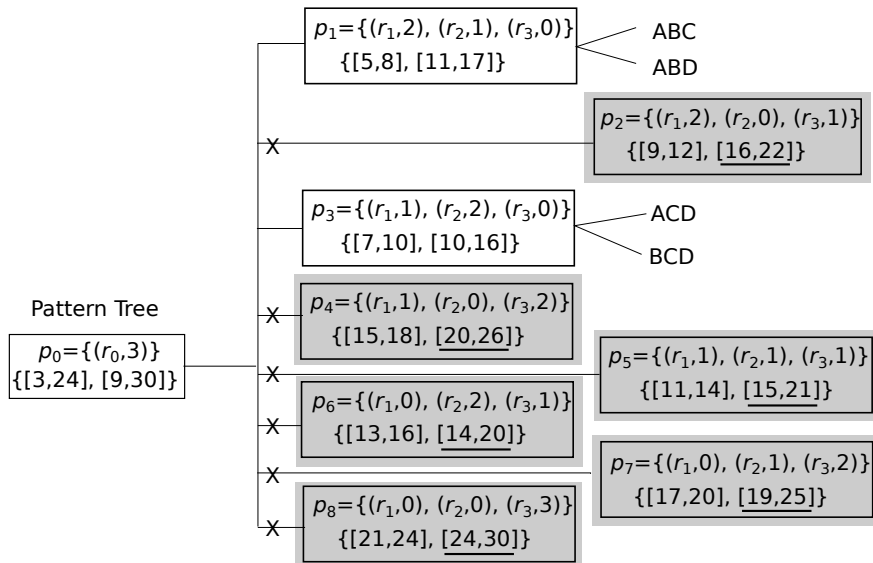


FIGURE 4.9: Constraint-based pruning

Definition 17 (Feasible Combination). A combination c is feasible if it has valid values in all the attributes $\forall A_j \in \mathcal{A}^*$, namely, $c.A_j \in [v_j^\perp, v_j^\top]$, $\forall A_j \in \mathcal{A}^*$.

The patterns can be discarded if they cannot generate feasible combinations.

Theorem 4. A pattern p cannot generate feasible combinations if $[p.A_t^\perp, p.A_t^\top] \cap [v.A_t^\perp, v.A_t^\top] = \emptyset$ ($\exists A_t \in \mathcal{A}^*$), where $[v.A_t^\perp, v.A_t^\top]$ is the valid range of values in attribute A_t .

Proof 4. Any combination c following the pattern p has the value $c.A_t \in [p.A_t^\perp, p.A_t^\top]$ for attribute $A_t \in \mathcal{A}^*$. If $[p.A_t^\perp, p.A_t^\top] \cap [v.A_t^\perp, v.A_t^\top] = \emptyset$, $c.A_t \notin [v.A_t^\perp, v.A_t^\top]$. Consequently, combination c is not a feasible combination.

Example 26. Fig. 4.9 shows the pattern tree for the constrained combination skyline query $CSKY^*$. According to Theorem 4, patterns $\{p_2, p_4, p_5, p_6, p_7, p_8\}$ can be pruned with respect to the constraint $[5, 13]$ on attribute A_2 because their ranges on attribute A_2 are out of the range constraint $[5, 13]$. Thus, we only need to expand $\{p_1, p_3\}$ for query $CSKY^*$ and obtain the final result $S = \{ABD, BCD\}$.

Given a pattern p , the sets of MBRs appearing in its child patterns are the same, and thus only the values of k_i 's need to be assigned. We can avoid enumerating useless ones by employing the forward checking approach, which is a common solution for constraint satisfaction problems [74] and used for answering spatial database queries [75]. At first, the possible value of each variable k_i is in the range of $[0, \min(|obj(r_i)|, k)]$, and then we assign values from k_1 . Once a k_i has been assigned, the ranges of the remaining variables may shrink due to the attribute constraints, and the new ranges can be determined using Theorem 4.

Example 26. (continued) If k_1 is set as 1, we use forward checking to update the value ranges of k_2 and k_3 . The range of k_2 will be $[0, 2]$, and the range of k_3 will be $[0, 1]$. For example, if $k_3 = 2$, then the value of the combination on A_2 is at least $4 + 8 + 8 = 20$, which violates the constraint $[5, 13]$.

4.8 Experiments

In this chapter, we report experimental results and our analyses.

4.8.1 Settings

We used both synthetic and real datasets in our experiment. We generated synthetic dataset using the approach introduced in [10] with various correlation coefficients, and we used the uniform distribution as default unless otherwise stated. For real dataset, we used the NBA dataset² which contains the statistics about 16,739 players from 1991 to 2005. The NBA dataset roughly follows an anti-correlated distribution. The default cardinality and the number of dimensions are both two.

²<http://www.nba.com/>

We compare our complete PBP algorithm with the baseline BBS algorithm. Since BBS cannot handle the explosive number of combinations when the dataset is large, we only compare PBP and BBS on small synthetic dataset. Both PBP and BBS were implemented in C++. The R-tree structure was provided by the spatial index library SaIL [78]. All the experiments were conducted on a Quad-Core AMD Opteron 8378 with 96 GB RAM. The operating system is Ubuntu 4.4.3. All the data structures and the algorithms were loaded into/run in main memory.

4.8.2 Experiments on Synthetic Datasets

Fig. 4.10(a) and 4.10(b) show the distributions of 2-item combinations and 3-item combinations, which are generated from a dataset containing 100 objects with two-dimensional attributes uniformly distributed in the range $[0, 1000] \times [0, 1000]$. In total, there are $\binom{100}{2} = 4950$ combinations and $\binom{100}{3} = 161700$ combinations shown as points in the two figures. The numbers of skyline combinations are much smaller; e.g., 13 from the 4950 2-item combinations and 28 from the 161700 combinations, as shown in the areas close to the horizontal axis and the vertical axis in Fig. 4.10(c) and 4.10(d).

Next, we compare our PBP algorithm with the BBS algorithm, and then study the efficiency of the PBP algorithm with respect to data distribution, cardinality, the number of attributes (dimensionality), and the fanout of R-tree.

Comparison with the BBS Algorithm

Since BBS cannot find skyline combinations from large datasets in acceptable response time, we compare the performances of BBS and PBP on small datasets that contain 50, 100, 150, 200 objects. For every data size, we vary the number of attributes in the range of $[2, 6]$. The experimental query is to find three-item skyline combinations.

Fig. 4.11(a) shows the size of R-trees used by BBS and PBP. For BBS, the R-tree sizes grows dramatically with the data size because R-trees have to index all the combinations that increase in an explosive way. As the figure shows, when the dataset contains 200 objects, the tree size is almost one gigabyte. Even worse, constructing such a huge R-tree consumes a lot of time, which means that BBS cannot work well when handling a huge number of combinations in practice. In contrast, PBP uses the R-tree for indexing single

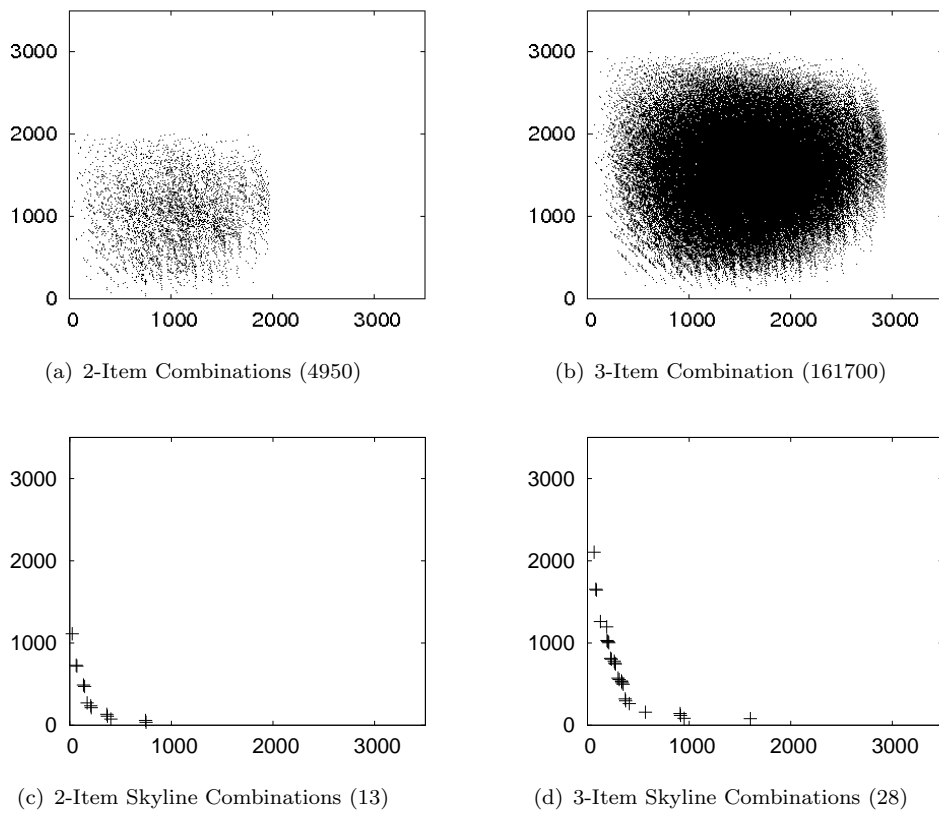


FIGURE 4.10: Distribution of combinations and skyline combinations

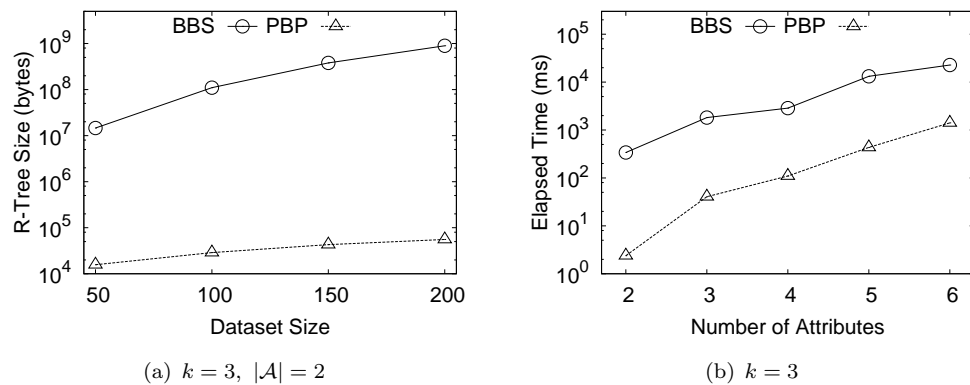


FIGURE 4.11: PBP versus BBS on small datasets

objects rather than combinations. Indexing single objects makes the tree size growing relatively slow. This is also why PBP can handle large datasets. We will show the experimental results on the large datasets in Section 4.8.2 to Section 4.8.3.

Fig. 4.11(b) shows the running time of BBS and PBP on the 100-object datasets, with the number of attributes varying from 2 to 6. For BBS, the time is the sum of the time

for enumerating combinations and the time consumed by retrieving skyline combinations. For PBP, the time is the time consumed by retrieving skyline combinations. The time for constructing R-trees is not included. As the figure shows, PBP outperforms BBS by at least one order of magnitude. One reason is that PBP executes queries on the R-tree that is far smaller than the R-tree used by BBS. Another reason is that the time for enumerating combinations is saved when running PBP.

In conclusion, PBP outperforms BBS in both space and time aspects. PBP consumes much less space and runs much faster than BBS. Additionally, when the number of objects increases, the space consumption speed of PBP is much slower than the speed of BBS.

The Effect of Data Distribution

We evaluate PBP on 4K, 8K, 16K, 32K, 64K datasets with different correlation coefficients -0.9 , -0.6 , -0.3 , 0.0 , 0.3 , 0.6 and 0.9 . The datasets with correlation coefficients -0.9 , -0.6 and -0.3 follow anti-correlated distributions. The datasets with correlation coefficients 0.9 , 0.6 and 0.3 follow correlated distributions. The dataset with correlation coefficient 0.0 follows uniform distributions. Each dataset has objects with two attributes. The queries are to select five-item skyline combinations from these datasets.

Fig. 4.12(a) shows the number of skyline combinations and Fig. 4.12(b) shows the running time. As Fig. 4.12(a) shows, there are more skyline combinations for the anti-correlated datasets and fewer skyline combinations for the correlated datasets. In the anti-correlated datasets, some objects are good in one attribute but are bad in the other attribute. In the correlated datasets, a part of the objects are good in both attributes. It can be seen that there are more results generated from the anti-correlated datasets than the results generated from the correlated datasets. This is because the combinations exhibit distribution features as single objects since their attribute values are the sums of their component objects' attribute values.

Fig. 4.12(b) shows the running time of PBP. It spends more time when running PBP on the anti-correlated datasets than on the correlated datasets. The time depends on the size of the priority queue and the number of dominance checks. Fig. 4.12(c) and 4.12(d) show the maximum size of the priority queue and the number of dominance checks, respectively. Since the patterns also follow the same distributes, there are more patterns which cannot

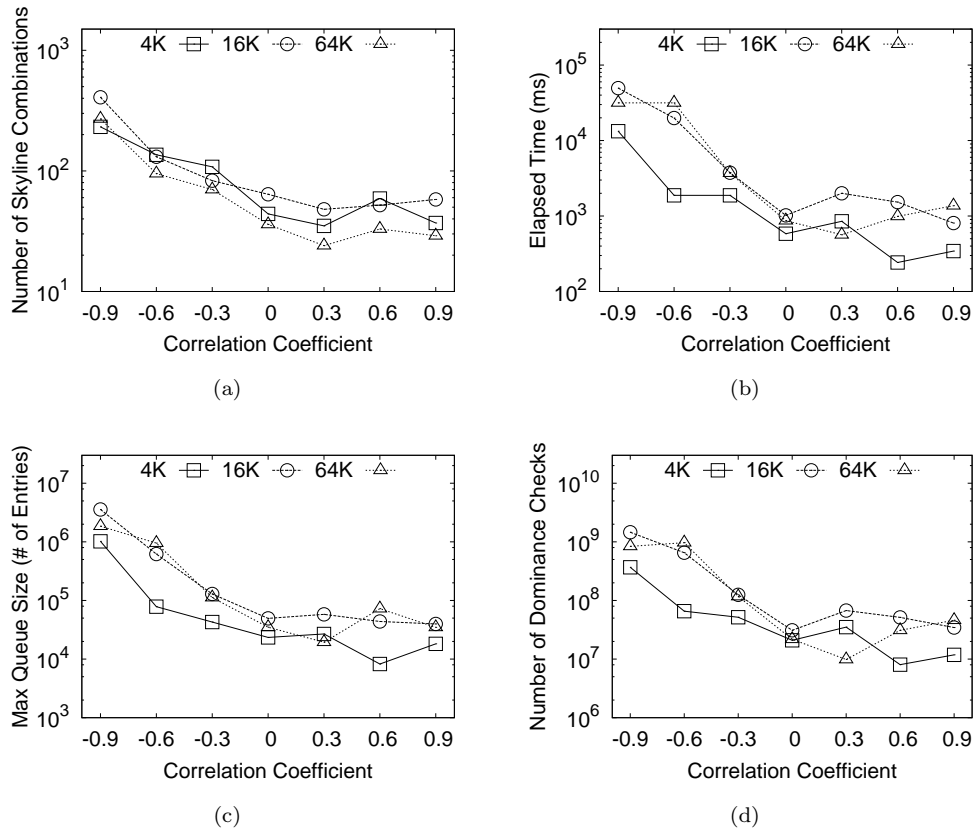


FIGURE 4.12: PBP performance for different distributions

be pruned but have to be pushed into the priority queue for the anti-correlated datasets. Consequently, more dominance checks occur.

Another observation is that running time does not vary significantly with the sizes of datasets. The reason is that the performance of PBP is not sensitive to the data sizes for low dimension cases. Such inference can be seen from the size of the priority queue and the number of dominance checks.

In conclusion, the number of results increases when the correlation coefficient of the dataset increases. The PBP algorithm can answer the combination skyline queries faster in the dataset with a higher correlation coefficient.

The Effect of Cardinality

We run PBP on 8K, 16K, 32K, 64K, and 128K datasets to find skyline combinations of cardinalities $k \in [3, 6]$. The objects in the datasets have two attributes.

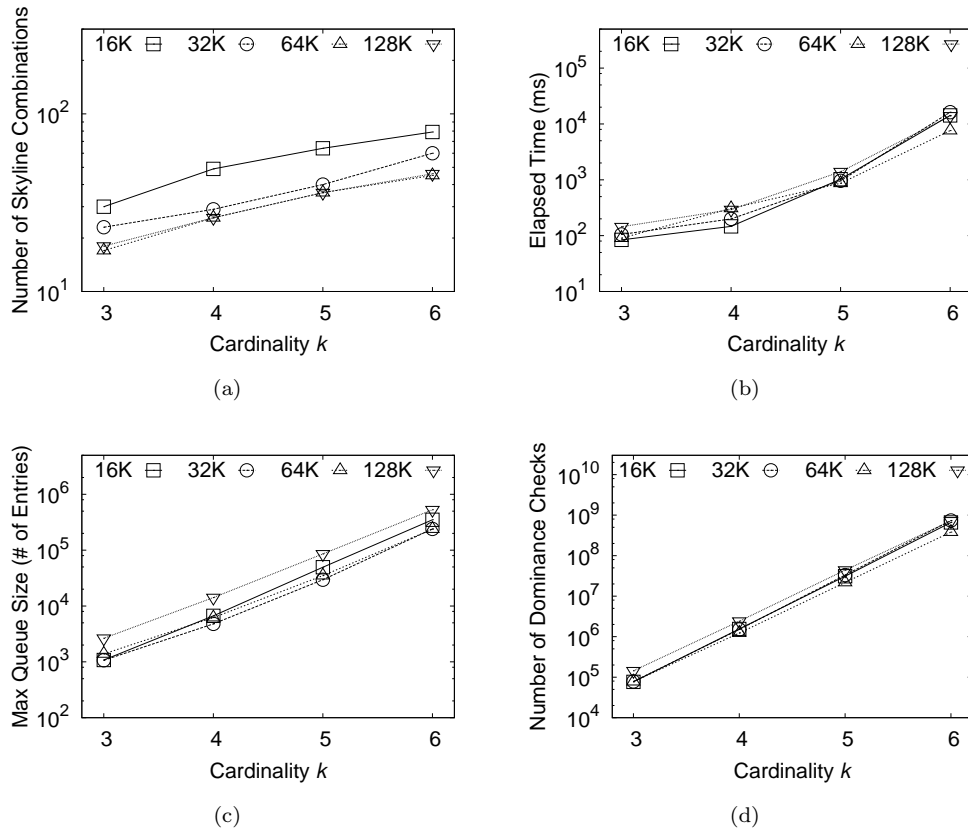


FIGURE 4.13: PBP performance for different cardinalities

Fig. 4.13(a) shows the number of skyline combinations. The number increases with the cardinality but not in an explosive way. The reason is much more combinations can be dominated for larger cardinalities. As Fig. 4.10 shows, more combinations are dominated by the skyline combinations when the cardinality increase from two to three.

Fig. 4.13(b) shows the running time of PBP. The time increases with the cardinality. It depends on the maximum size of the queue and the number of dominance checks, which are shown in Fig. 4.13(c) and 4.13(d), respectively. When the cardinality enlarges, the number of patterns increases. Thus, more patterns are pushed into the queue and more dominance checks are needed. Another general trend is that the running time increases with dataset sizes, but the influence is not as significant as that of cardinality. Considering the number of combination $\binom{|\mathcal{O}|}{k}$, it grows faster with the increase of k than with that of $|\mathcal{O}|$.

In conclusion, the number of results increases when the cardinality increases, but does not increase in an exponential way. The running time of the PBP algorithm also increases with the cardinality.

The Effect of Dimensionality

We evaluate the effect of dimensionality by varying the number of attributes in the range of [2, 6]. For each dimensionality, we run PBP on 1K, 2K, 4K, 8K, 16K datasets, respectively. The query is to find two-item combinations from these datasets.

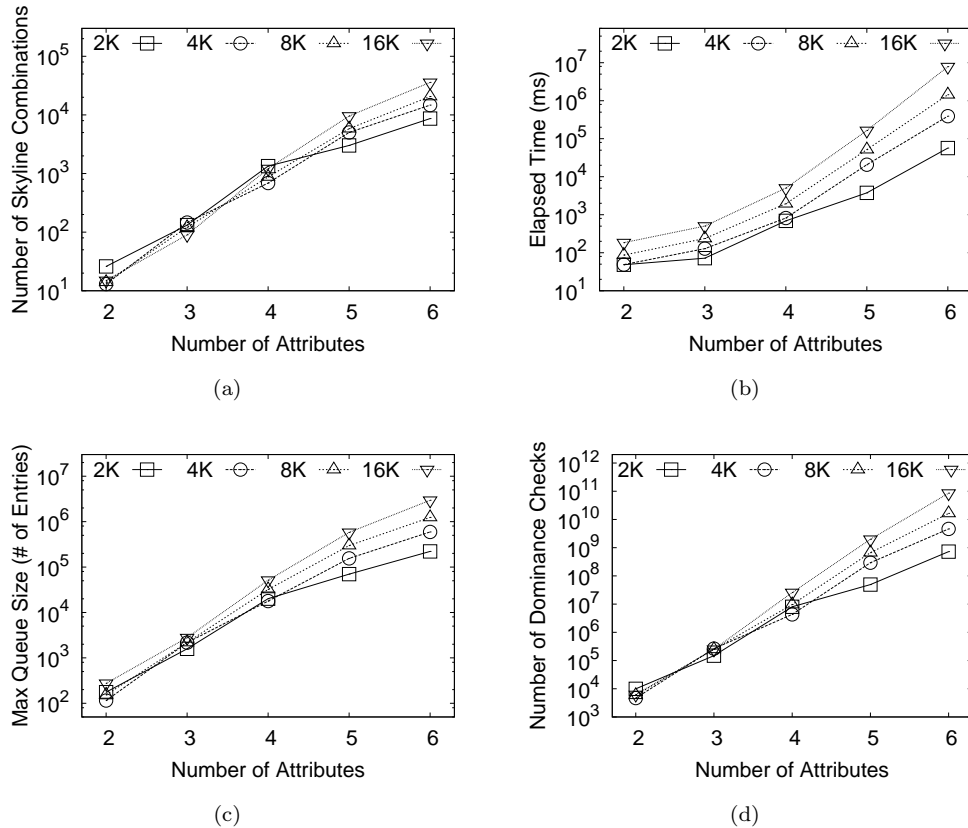


FIGURE 4.14: PBP performance for different number of attributes

Fig. 4.14(a) shows the number of skyline combinations. The number exhibits a rapid growth with the dimension. The reason is that when the dimension increases, it is more likely that two combinations are better than each other in different subsets of the dimensions. Thus, one cannot dominate another and the number of skyline combinations increases. It is also called the curse of dimensionality [79].

Fig. 4.14(b) shows the running time. The time increases with the number of attributes. It depends on the the maximum size of the priority queue and the total number of dominance checks, which are shown in Fig. 4.14(c) and 4.14(d), respectively. In Fig. 4.14(c) and 4.14(d), both the size of the queue and the number of dominance checks increase with the dimension. One reason is that when the dimension increases, the number of nodes in

the R-tree grows and more overlap among MBRs is incurred. More patterns are hence generated, and the pruning power of PBP is reduced as well.

Observing Fig. 4.14(b), the time increases with the size of datasets, and the gap between two datasets with different sizes is more substantial for higher dimensionality. This is also due to the increase of nodes and more overlap in the R-tree.

In conclusion, the number of results increases when the number of attributes (i.e., dimensionality) grows. The PBP algorithm can answer the queries faster in the dataset with a smaller dimensionality. Additionally, for high-dimensional datasets, the query cost grows quickly with the dataset size.

The Effect of R-Tree Fanout

The structure of R-tree may also impact the performance of PBP. Under the in-memory setting, the dominant factor of the algorithm's runtime performance is not I/O but the number of patterns processed. In addition, a large fanout, which is preferred in a disk-resident R-tree, is not necessary a good choice.³ As shown below, a small fanout shows better performance in our problem setting. Consider an R-tree of order (m, M) where each node must have at most M child nodes and at least m child nodes. Note that m decides the fanout of the R-tree. There are at most $\lceil N/m^i \rceil$ nodes at the level i in the R-tree⁴, and thus there are at most $\binom{\lceil N/m^i \rceil + k - 1}{k} \leq \frac{(\lceil N/m^i \rceil + k - 1)^k}{k!}$ patterns at the corresponding level i in the pattern tree. In the worst case, the total number of patterns is

$$\sum_{i=1}^{\lceil \log_m N \rceil} \frac{(\lceil \frac{N}{m^i} \rceil + k - 1)^k}{k!} \quad (4.4)$$

where $\lceil \log_m N \rceil - 1$ is the maximum height of the R-tree. When m increases, the number of patterns decreases according to Equation 4.4; however, the pruning capabilities of Theorems 1 and 2 becomes weaker since the lower and upper bounds of a pattern become looser and less accurate.

We run PBP on the datasets indexed by the different R-tree structures with the fanouts $m \in [4, 8]$. Fig. 4.15(a) shows the running time on three datasets with dimensions $d = 3$, $d = 4$, and $d = 5$. Each dataset has 1K objects and the algorithm searches for skyline

³For example, T-tree, an in-memory index for ordered keys has a binary index structure [80].

⁴Note that level-1 denotes the leaf level and level- $(i + 1)$ denotes the parent level of level- i .

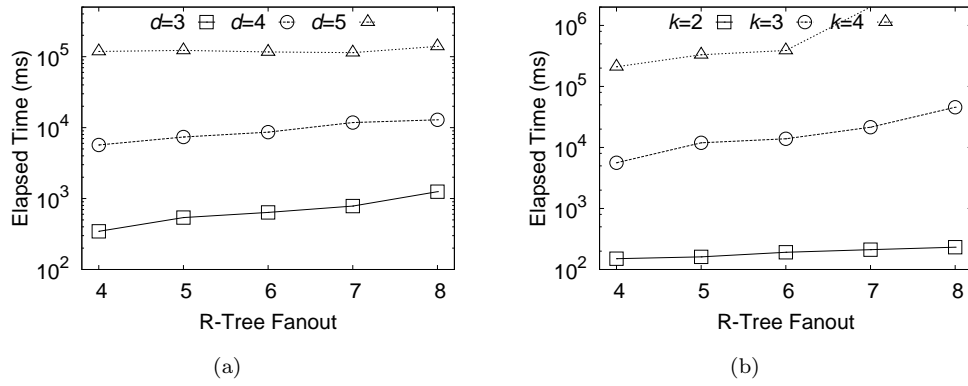


FIGURE 4.15: PBP performance for different fanouts of R-tree

combinations of cardinality $k = 3$. For the datasets with dimensions $d = 3$ and 4, PBP performs best when $m = 4$. In our experiments, when $m = 4$ we enumerate 341.3K patterns, while when $m = 8$ we enumerate 690.5K patterns, which showcases the better pruning power of the proposed algorithm under small fanouts. For the dataset with dimension $d = 5$, PBP performs best when $m = 7$. The reason is that the increase of dimensionality causes more overlaps between MBRs and thus weaken the pruning power. We also found that a large fanout, which is preferred in a disk-resident R-tree, usually results in bad performance. When $k = 3$ and $d = 3$, the running time is 41.9s under a fanout $m = 32$, 121.5 times slower than under a fanout $m = 4$. Fig. 4.15(b) shows the running time on a four-dimensional dataset containing 1K objects. The algorithm retrieves skyline combinations of cardinalities $k = 2$, $k = 3$, and $k = 4$ and performs best when $m = 4$.

In general, we suggest users choose a small fanout, e.g., $m = 4$, for tasks with low dimensionality, and a moderately larger fanout, e.g., $m = 7$, for high-dimensional tasks.

4.8.3 Experiments on Real Datasets

We run PBP on the real datasets. The sizes of our datasets are 2K, 4K, 8K, 16K, and the number of attributes varies from 2 to 5. We conduct two groups of experiments: one is to verify the effect of cardinality k , and another is to verify the effect of dimensionality $|\mathcal{A}|$.

Fig. 4.16 shows the effect of cardinality on real datasets. Fig. 4.16(a) shows the number of skyline combinations grows with the cardinality. Fig. 4.16(b) shows the running time

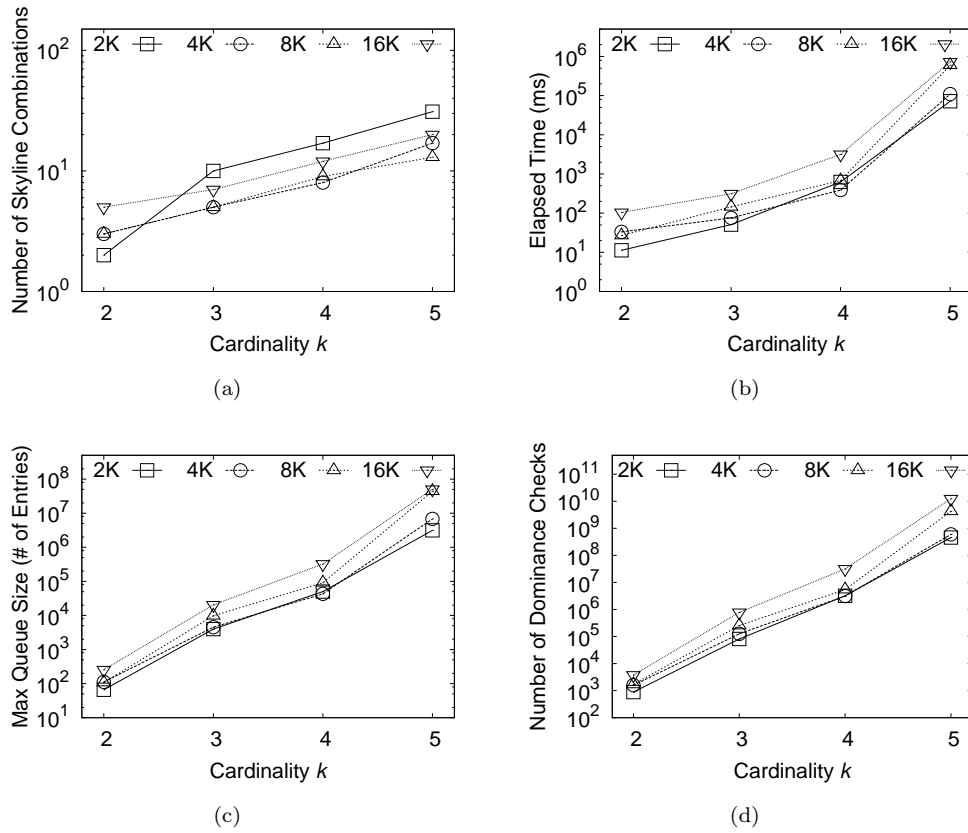


FIGURE 4.16: PBP performance for different cardinalities on real datasets

increases with the cardinality, which is consistent with the increase of the queue size and the increase of the dominance check number shown in Fig. 4.16(c) and 4.16(d), respectively. A similar trend is observed as we have seen for synthetic datasets, but has a more rapid growth of running time with the cardinality. This is because the real dataset follows anti-correlated distribution while the synthetic dataset follows uniform distribution, and hence fewer combinations are dominated for the former.

Fig. 4.17 shows the effect of dimensionality on real datasets. Fig. 4.17(a) shows the number of skyline combinations is larger for higher dimensional datasets. Fig. 4.17(b) shows the running time of PBP on the real datasets with different number of attributes. Since the time depends on the size of the queue and the total number of dominance checks, the shapes and trends of the curves in Fig. 4.17(c) and 4.17(d) are consistent with the appearances of curves in Fig. 4.17(b). When the number of attributes grows, the time increases and the gap between two datasets with different sizes is enlarged, and is more substantial than on synthetic data.

In conclusion, the PBP algorithm can answer the experimental queries in real datasets

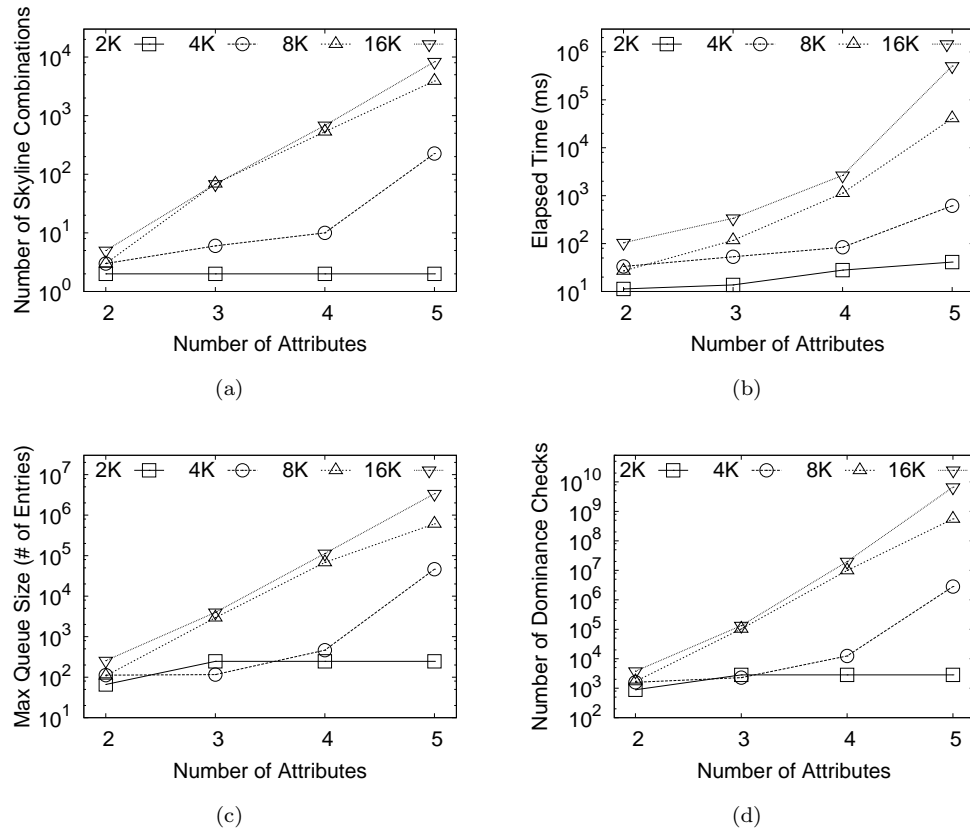


FIGURE 4.17: PBP performance for different number of attributes on real datasets

at acceptable speeds (i.e., within 1000 seconds). When varying the cardinality and the dimensionality, the experimental results are consistent with the results on the synthetic datasets.

4.8.4 Summary

To sum up, we have the following main findings after conducting the experiments on both synthetic and real datasets.

- The number of skyline combinations is far smaller than the total number of combinations.
- BBS cannot efficiently answer the combination skyline queries in large datasets.⁵
- PBP can efficiently answer the queries in large datasets. It outperforms BBS by at least one order of magnitude, i.e., ten times.

⁵The word “large” means that the number of objects varies from 1,000 to 128,000.

- PBP is influenced by the data distribution, i.e., the query cost is higher on the anti-correlated dataset than on the correlated dataset.
- PBP is also influenced by the cardinality, the dimensionality, and the data size, i.e., the query cost gets higher when the cardinality or/and the dimensionality or/and the data size gets larger, .

Algorithm 12: CompletePBP (T, k)**Input** : T is the R-tree built on \mathcal{O} ; k is the cardinality.**Output:** The skyline combination set $S = CSKY(\mathcal{O}, k)$.

```

1  $S \leftarrow \emptyset$ ;
2  $r_0 \leftarrow$  the root node of  $T$ ;
3  $Q \leftarrow \{(r_0, k)\}$ ;  $M \leftarrow \emptyset$ ;
4 while  $Q \neq \emptyset$  do
5    $p \leftarrow Q.pop()$ ;
6   if  $p$  is a combination then
7     if  $\nexists c \in S, c \prec p$  then
8        $S \leftarrow S \cup \{p\}$ ;
9     end if
10  else
11    if  $\exists c \in S, c \prec p$  then
12      continue;
13    end if
14    if the MBRs in  $p$  are internal nodes then
15       $P \leftarrow$  ExpandPatternOpt ( $p$ );
16      for each  $p' \in P$  do
17        if  $\nexists p'' \in P, p'' \prec p'$  then
18           $Q.push(\langle p', p'.mindist \rangle)$ ;
19        end if
20      end for
21    else
22       $C \leftarrow e$ ; //  $e$  is the identity element of Cartesian product
23      for each  $(r_i, k_i) \in p$  do
24        if  $(r_i, k_i)$  is a set of skyline combinations then
25           $C' \leftarrow M[r_i][k_i]$ ;
26        else
27           $C' \leftarrow CSKY(obj(r_i), k_i)$ ;
28           $C \leftarrow C \times C'$ ;
29           $M[r_i][k_i] \leftarrow C'$ ;
30          update  $(r_i, k_i)$ 's ancestor rules in  $M$ ;
31        end if
32      end for
33      for each  $c \in C$  do
34         $Q.push(c, c.mindist)$ ;
35      end for
36    end if
37  end if
38 end while
39 return  $S$ 

```

Chapter 5

Conclusions and Future Work

In this Chapter, we first summarize this thesis in Section 5.1. Next, we present several interesting extensions that serve as our future work in Section 5.2.

5.1 Conclusions

We have studied two new decision-support queries, *DBS queries* for geographical data and *combination skyline queries* for multi-attribute data. Decision-support queries search for a handful of desired objects by considering the multiple preferences of a user. The DBS queries can search for spatial objects considering the user's preference on distance and direction. Combination skyline queries can search for object combinations considering the user's preferences on different attributes.

There are usually two ways to select appropriate objects to answer decision support queries. One way is top- k queries, which retrieve objects according to their scores. Another way is skyline queries, which retrieve the objects with respect to domination relationships. Generally speaking, both DBS queries and combination skyline queries fall into the skyline query category, because they select objects according to domination relationships.

In DBS queries, if two spatial objects are in the same direction, the nearer object dominates the farther object. In combination skyline queries, the combinations consist of a fixed number of objects and the attribute values are aggregations of their components'

values. From this perspective, the combinations are also objects with multi-attributes. One combination dominates another combination if it is better than another one in at least one attribute and is not worse in any attribute.

The basic ideas of processing DBS queries and combination skyline queries are the same. The main observation is that the selected objects (i.e., spatial objects, or combinations) are far fewer than the objects in the whole set. Due to this observation, the design goal of the algorithms is to find the desired objects as fast as possible by checking an adequate number of objects rather than checking all the objects. To reach this goal, the algorithms for processing DBS queries aim at comparing fewer objects and terminating the process as quickly as possible. The PBP algorithm for processing combination skyline queries aims at enumerating a small number of combinations that show promise for being among the final results.

We have conducted experiments to evaluate the performance of the proposed algorithms. The results of experiments using both synthetic and real data sets demonstrate that every algorithm proposed can find answers efficiently.

5.1.1 DBS Queries

In particular, we have studied DBS queries in both Euclidean spaces (i.e., \mathbb{E} -DBS queries) and road networks (i.e., \mathbb{R} -DBS queries). For the two types of DBS queries, the measures for distances and directions are different. In \mathbb{E} -DBS queries, the distance of an object is the Euclidean distance between the object and the user, while its direction is the vector from the user to the object. If the included angle between the directions of two objects is smaller than a user-defined threshold θ , the two objects are in the same direction. In \mathbb{R} -DBS queries, the distance of an object is the length of the shortest path from the user to the object, while the direction is set to be the shortest path itself. If the shortest paths of two objects overlap, the two objects are in the same direction.

Definitions of the domination relationship are the same in both Euclidean spaces and road networks. For the two types of DBS queries, the nearer object can dominate the farther object if they are in the same direction. The DBSs are the objects that cannot be dominated. According to this definition, we studied two types of queries, snapshot queries

and continuous queries. A snapshot query finds DBSs with respect to the user's current position. The continuous query is to update the DBSs when the user keeps moving.

We developed efficient algorithms for processing snapshot queries and continuous queries, respectively. The basic idea of algorithms for snapshot queries is to reduce the number of dominance checks. To answer continuous queries, the idea is to predict the change moments of the results and to update the results at those moments.

Based on extensive experiments with both real and synthetic data sets, we demonstrated the performance of our proposed algorithms. Investigating the experimental results, we found that the number of results and the query cost are influenced by the threshold θ in \mathbb{E} -DBS queries. The number of results and the query cost are influenced by the object density in \mathbb{R} -DBS queries. Experimental results also confirm that the proposed algorithms can answer both the snapshot case and the continuous case promptly.

5.1.2 Combination Skyline Queries

We have studied the combination skyline problem, a new variation of the skyline problem. The combination skyline problem is to find combinations consisting of k objects that are not dominated by others. Due to the exponential number of k -item combinations, the traditional approaches cannot find the answers efficiently.

Motivated by the inefficiency of the traditional algorithms, we have proposed the PBP algorithm to answer combination skyline queries efficiently. With an R-tree index, the algorithm generates combinations with object-selecting patterns organized into a tree. In order to prune the search space and improve efficiency, we have presented two pruning strategies and a technique to avoid duplicate pattern expansion.

In addition, we have discussed two variations of the combination skyline queries, i.e., incremental combination skyline queries and constrained combination skyline queries. It is easy to modify the PBP algorithm to answer the two types of queries.

The efficiency of the proposed algorithm was evaluated by extensive experiments using synthetic and real data sets. The query cost increases when the correlation coefficient decreases, the cardinality increases, or the dimensionality increases. Investigating the experimental results on real data sets, we found that the proposed algorithms can answer queries efficiently.

5.2 Future Work

5.2.1 DBS Queries

In the future, we would like to extend DBS queries in the following five directions.

First, we intend to explore other approaches (e.g., using the MBR-trimming method of [81]) to tackle \mathbb{E} -DBS queries. Our current approach is to reduce the search space by considering the partition angle sizes. Another possible approach is to trim the MBRs incrementally when the objects are indexed by an R-tree.

Second, we plan to extend our work to consider non-spatial attributes, for example, queries such as “find inexpensive hotels near me.” In this case, we should consider the non-spatial attribute “price” as well as the distance and direction.

Third, we can consider the situation of a moving user along with several moving objects. Suppose a football game is going on and a player wants to pass the ball to a teammate. The teammates and opponents have different directions and distances according to the players current position. In this situation all the objects are moving, including the player. We can help this football player to make a good decision about passing the ball by considering distance and direction.

Fourth, we can weigh objects according to distance and direction, and recommend objects with higher weights. Assuming that we can infer the user direction from movement, nearby objects in the same direction should be assigned higher weights. Thus, we can sort the objects by weight and return the top ones in the ranking list.

Fifth, we can construct a prototype system to provide DBS query services for mobile users based on the proposed ideas. When road network information is available, users can submit \mathbb{R} -DBS queries. When the road network information is not detailed enough, or even when it is not available, the users can submit \mathbb{E} -DBS queries.

5.2.2 Combination Skyline Queries

In the future, we would like to extend combination skyline queries towards the following four directions.

First, we intend to propose approximation algorithms for answering combination skyline queries. Experimental results have shown that the query cost increases steeply when the cardinality k or the dimensionality m increases. An approximation algorithm would be able to answer the queries with large k or/and large m values. The approximation algorithm should run fast and the results returned should be within an acceptable approximation ratio.

Second, we plan to extend the k -item combination skyline problem to a general version where the cardinality k varies. It is easy to infer that a combination is dominated by its sub-combinations. For example, the combination $\{ABCD\}$ is dominated by its sub-combination $\{ABC\}$. Except in situations where the sub-combination relationships exist, it is not easy to infer the dominance relationship between a k_1 -item combination and a k_2 -item combination, where $k_1 \neq k_2$. This creates difficulties and challenges in the problem.

Third, we plan to solve the problem when the aggregation function is not monotonic. In the current problem definition, we form a combination by using a monotonic aggregation function. However, when the monotonic property does not hold, we cannot estimate the value ranges of the combinations from the patterns. Thus, the two pruning strategies do not work. To solve the problem, other effective methods are required.

Fourth, we will implement a prototype system to support combination skyline queries based on the proposed ideas. To help users understand the query results easily, we would like to explain the superiority of each skyline combination. One possible measurement of the superiority is the number of combinations dominated by the skyline combination. Further, by ranking the results in their superiority order, we can present an adequate number of high-quality results to users when too many skyline combinations have been found.

Appendix A

Appendix for DBS Queries

A.1 Processing of Continuous k NN Queries

We show how to retrieve k -nearest objects for a linearly moving query point \vec{q} during a time interval \mathcal{I} by extending the idea of [7]. The function CNNQUERY in Algorithm 2 is implemented based on the method. Consider the following example, which is continued from Example 8.

Example 27. We explain the idea using Fig. A.1. We want to find the nearest neighbor objects for the time interval $\mathcal{I} = (0, 100)$. Initially, we find the nearest neighbor object of the query point when $t = 0$. It is object a . Then we try to find out when the nearest object changes to other objects during $(0, 100]$. The algorithm proposed in [7] finds the first change for the current nearest object. The candidates set is made up of objects whose distances are smaller than a when the query point is at the end position ($t = 100$). In other words, the objects falling into the circle in Fig. A.1 are candidates. There is only one object g in the circle. We calculate the change moment, which is the time when $d_g = d_a$. As described below, the time is $t = 75$. If we have multiple candidates, the one with the smallest change moment is selected. ■

In the above example, we have shown how to find the first nearest objects for an interval \mathcal{I} . We can extend this method to find the k -th nearest neighbor objects for an interval \mathcal{I} . For example, if we want to find the second nearest neighbor objects for $(0, 75)$ where the first nearest one is a , we remove a from the target and reuse the above method. In this scheme, we can find out k -nearest neighbor objects incrementally.

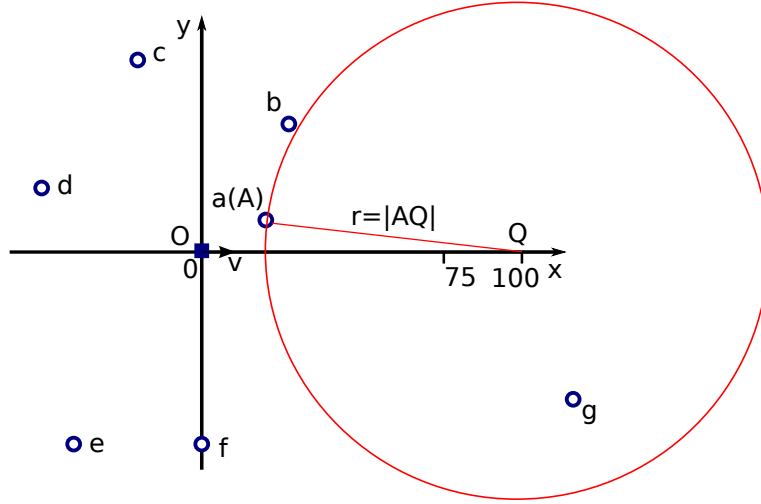


FIGURE A.1: Candidate area for the next nearest object

Algorithm 13 shows function `CNNQUERY` that implements the idea. The function receives \vec{q} , the location of the time-parameterized query point as shown in Eq. (2.3), and the target time interval $\mathcal{I} = (I_s, I_e)$. The function returns k -th nearest neighbor objects with their corresponding time intervals.

In lines 5 and 6, q_s and q_e denote the start and end points of the query. In line 7, we construct the *exclusion list* E , which consists of the current top $(k - 1)$ -objects for the time interval \mathcal{I} . The construction is easy because we maintain a tree structure as shown in Fig. 2.11 while the process of a continual query. For example, when we issue `CNNQUERY`(\vec{q} , (0, 71), 3) in Fig. 2.11, we already know that $n_1 = a$ and $n_2 = b$.

In line 10, we use function `get_NN_object()` to retrieve the k -th nearest object n_k for the start position q_s . The exclusion list E is used for finding the k -th object. In line 13, we get the candidates for the next change by using function `dist_query()`. We assume that two functions `get_NN_object()` and `dist_query()` are efficiently supported by the spatial index.

In the while loop (lines 16-26), we incrementally add a k -th nearest neighbor object to the result R . In line 17, we calculate the *change moment* between n_k , the current k -th nearest neighbor object, and p , one of the candidates. In line 19, we select the candidate p_c with the minimum change moment t_c as the next k -th nearest neighbor object. In line 21, we update R by inserting the information of the current k -th nearest neighbor n_k and

Algorithm 13: CNNQuery(\vec{q}, \mathcal{I}, k)

```

//  $\mathcal{I} = [I_s, I_e]$  is a time interval
// Initialization
1  $R \leftarrow \emptyset;$  // Set of  $k$ -th NN objects
2  $q_s \leftarrow \vec{q} \big|_{t=I_s};$  // Start point of  $\vec{q}$ 
3  $q_e \leftarrow \vec{q} \big|_{t=I_e};$  // End point of  $\vec{q}$ 
4  $E \leftarrow \{n_1, n_2, \dots, n_{k-1}\};$  // Exclusion set:  $(k-1)$ -NN objects for  $\mathcal{I}$ 
// Initialization for loop
5  $n_k \leftarrow \text{get\_NN\_object}(q_s, E);$  // Get initial  $k$ -th NN object (by excluding  $E$ )
6  $t_s \leftarrow I_s;$  // Initialize the start time
7  $C \leftarrow \text{dist\_query}(q_e, \text{dist}(n_k, q_e));$  // Get candidates of the next  $k$ -th NN object
// Add next  $k$ -th NN object to  $R$  in each iteration
8 while  $C \neq \emptyset$  do
9    $T \leftarrow \{\langle p, \text{change\_moment}(n_k, p) \rangle \mid p \in C\};$  // Compute change moment for each candidate
10   $\langle p_c, t_c \rangle \leftarrow \langle p, t \rangle$  such that  $t$  is the smallest in  $T;$  //  $p_c$  causes the first change at  $t_c$ 
11   $R \leftarrow R \cup \{\langle n_k, (t_s, t_c) \rangle\};$  // Add the new  $k$ -th NN object to  $R$ 
12   $n_k \leftarrow p_c;$ 
13   $t_s \leftarrow t_c;$ 
14   $C \leftarrow \text{dist\_query}(q_e, \text{dist}(n_k, q_e));$ 
15 end while
16  $R \leftarrow R \cup \{\langle n_k, (t_s, I_e) \rangle\};$  // Add the last one to  $R$ 
17 return  $R;$ 

```

its valid time interval (t_s, t_e) . In lines 24 and 25, we update the status, and then continue the while loop. Finally, in line 27, we add the information for the last time interval.

A remaining problem is how to implement `change_moment()` function. For two points $p_i = (x_{p_i}, y_{p_i})$ and $p_j = (x_{p_j}, y_{p_j})$, their change moment is the solution of the equation $d_i = d_j$, where d_i (d_j) is the time-parameterized distance between q and q_i (q_j). Since the equation is quadratic in terms of t , it is easily solvable.

A.2 Details of Dominance Checking

Given a query point \vec{q} and two points $a = (x_a, y_a)'$, $b = (x_b, y_b)'$. The vectors of the two points are:

$$\vec{a} = \vec{p}\vec{a} = \begin{pmatrix} x_a \\ y_a \end{pmatrix} - \begin{pmatrix} x_v t + x_{\vec{q}} \\ y_v t + y_{\vec{q}} \end{pmatrix} \quad (\text{A.1})$$

$$\vec{b} = \vec{pb} = \begin{pmatrix} x_b \\ y_b \end{pmatrix} - \begin{pmatrix} x_v t + x_{\bar{q}} \\ y_v t + y_{\bar{q}} \end{pmatrix}. \quad (\text{A.2})$$

We consider the problem whether (and when) $\lambda_{ab} \geq \theta$ (i.e., $\cos \lambda_{ab} \leq \cos \theta$) is satisfied during a certain time interval $[t_s, t_e]$. For the problem, we consider the equality formula $\cos \lambda_{ab} = \cos \theta$. The formula can be represented as

$$\frac{Gt^2 + Ht + I}{\sqrt{Gt^2 + Jt + K}\sqrt{Gt^2 + Lt + M}} = \cos \theta, \quad (\text{A.3})$$

where

$$G = x_v^2 + y_v^2 \quad (\text{A.4})$$

$$H = -[(C + E)x_v + (D + F)y_v] \quad (\text{A.5})$$

$$I = CE + DF \quad (\text{A.6})$$

$$J = -2(Cx_v + Dy_v) \quad (\text{A.7})$$

$$K = C^2 + D^2 \quad (\text{A.8})$$

$$L = -2(Ex_v + Fy_v) \quad (\text{A.9})$$

$$M = E^2 + F^2 \quad (\text{A.10})$$

$$C = x_a - x_{\bar{q}} \quad (\text{A.11})$$

$$D = y_a - y_{\bar{q}} \quad (\text{A.12})$$

$$E = x_b - x_{\bar{q}} \quad (\text{A.13})$$

$$F = y_b - y_{\bar{q}}. \quad (\text{A.14})$$

Eq. (A.3) can be transformed to a quartic formula

$$c_4 t^4 + c_3 t^3 + c_2 t^2 + c_1 t + c_0 = 0, \quad (\text{A.15})$$

where

$$\begin{aligned} c_4 &= G^2(1 - T) \\ c_3 &= 2GH - (GL + JG)T \\ c_2 &= 2GI + H^2 - (GK + JL + GM)T \\ c_1 &= 2HI - (KL + JM)T \\ c_0 &= I^2 - KMT \\ T &= \cos^2 \theta. \end{aligned} \quad (\text{A.16})$$

We can solve this quartic formula by using the GNU Scientific Library [46] and get four solutions t_i ($i = 1, \dots, 4$). Note that a solution of Eq. (A.15) may not be a valid solution

of Eq. (A.3) because it gives a negative value $\cos \theta$. Thus, we have to verify whether each solution t_i is a real solution for Eq. (A.3). If $(Gt_i^2 + Ht_i + I)$ and $\cos \theta$ has the same sign, namely,

$$(Gt_i^2 + Ht_i + I) \cos \theta \geq 0, \quad (\text{A.17})$$

t_i is a real solution for Eq. (A.3).

If the real solutions fall into $\mathcal{I} = [t_s, t_e]$, they separate the whole time interval into several sub-intervals. We check $\cos \lambda_{ab}(t)$ for every sub-interval $\mathcal{I}_j = [t_s^j, t_e^j]$ to judge whether $\lambda_{ab}(t)$ is greater than θ or not. Because $\cos \lambda_{ab}(t)$ is a continuous function, we can easily decide as follows:

$$\lambda_{ab}(t)|_{t \in \mathcal{I}_j} \begin{cases} \geq \theta, & \text{if } \cos \lambda_{ab}(\frac{t_s^j + t_e^j}{2}) < \cos \theta \\ < \theta, & \text{if } \cos \lambda_{ab}(\frac{t_s^j + t_e^j}{2}) > \cos \theta. \end{cases} \quad (\text{A.18})$$

A.3 Details of Termination Checking

The function of the partition angle φ_{ab} is $\cos \varphi_{ab} = A/B$, where

$$A = (x_v t - x_a)(x_v t - x_b) + (y_v t - y_a)(y_v t - y_b) \quad (\text{A.19})$$

$$B = \sqrt{(x_v t - x_a)^2 + (y_v t - y_a)^2} \sqrt{(x_v t - x_b)^2 + (y_v t - y_b)^2}. \quad (\text{A.20})$$

If we consider the semantics of $\cos \varphi_{ab}$, it is clear that $\cos \varphi_{ab} \rightarrow 1$ when $t \rightarrow \pm\infty$. We get the derivative of the cosine function

$$\frac{d \cos \varphi_{ab}}{dt} = \frac{A'B - AB'}{B^2}, \quad (\text{A.21})$$

where A' and B' are abbreviations for the derivatives. Note that $B^2 > 0$ is always hold (we do not assume that a and b are on the same locations). Since our concern is the behavior of $\cos \varphi_{ab}$, we can omit B^2 . Thus, we focus on the function

$$\begin{aligned} C &= A'B - AB' \\ &= \frac{2A'DE - A(D' \cdot E + D \cdot E')}{2D^{\frac{1}{2}}E^{\frac{1}{2}}}, \end{aligned} \quad (\text{A.22})$$

where

$$D = (x_v t - x_a)^2 + (y_v t - y_a)^2 \quad (\text{A.23})$$

$$E = (x_v t - x_b)^2 + (y_v t - y_b)^2. \quad (\text{A.24})$$

Since $D^{\frac{1}{2}} E^{\frac{1}{2}} > 0$, we focus on

$$F = 2A'DE - AD'E - ADE' \quad (\text{A.25})$$

$$= 2GH, \quad (\text{A.26})$$

where

$$G = (x_b y_v - x_a y_v - y_b x_v + y_a x_v)t + (x_a y_b - y_a x_b) \quad (\text{A.27})$$

$$H = It^2 + Jt + K \quad (\text{A.28})$$

$$I = (x_v^2 + y_v^2)(x_b y_v - x_a y_v - y_b x_v + y_a x_v) \quad (\text{A.29})$$

$$J = 2(x_v^2 + y_v^2)(x_a y_b - y_a x_b) \quad (\text{A.30})$$

$$K = (x_a^2 + y_a^2)(x_b y_v - y_b x_v) + (x_b^2 + y_b^2)(y_a x_v - x_a y_v). \quad (\text{A.31})$$

A.3.1 Function G

We consider function G (Eq. (A.27)). By taking $G = 0$, we get

$$t = \frac{x_a y_b - y_a x_b}{(x_a - x_b)y_v - (y_a - y_b)x_v}. \quad (\text{A.32})$$

If we use the notation $\vec{v} \times \vec{w}$ for the *outer product* of vectors \vec{v} and \vec{w} , we get

$$t = \frac{\vec{a} \times \vec{b}}{\vec{ba} \times \vec{q}}, \quad (\text{A.33})$$

where $\vec{ba} = (x_a - x_b, y_a - y_b)'$.

Recall that three cases A, B, and C presented in Section 2.6.4. We observe that this t -value means that for each case. For case A, an example was shown in Fig. 2.14. Eq. (A.33) takes the value $t = 2.5$, and it corresponds to the local maximum. For case B, Eq. (A.33) corresponds to the minimum value. In Fig. 2.15, the value is $t = 2.2$. For case C, when \vec{ab} is parallel to \vec{q} ($\vec{ab} \times \vec{q} = 0$), $t = \infty$ because the denominator is zero.

In summary, we obtained the following properties:

1. In case A (a, b are on the same side and \vec{ab} is not parallel to \vec{q}), Eq. (A.33) gives the t -value at which the cosine function takes the local maximum 1.
2. In case B (a, b are on the different sides), Eq. (A.33) gives the t -value at which the cosine function takes the local minimum -1 .
3. In case C (a, b are on the same side and \vec{ab} is parallel to \vec{q}), Eq. (A.33) does not correspond to the local minimum.

A.3.2 Function H

We want to know the condition when $H = 0$ holds. Thus, we consider the *discriminant* of H in Eq. (A.28):

$$\begin{aligned} J^2 - 4IK &= 4(x_v^2 + y_v^2)[(x_v^2 + y_v^2)(x_a y_b - y_a x_b)^2 - \\ &\quad - (x_b y_v - x_a y_v - y_b x_v + y_a x_v)K]. \end{aligned} \quad (\text{A.34})$$

Since $x_v^2 + y_v^2 > 0$, it is suffice to analyze

$$\begin{aligned} &(x_v^2 + y_v^2)(x_a y_b - y_a x_b)^2 - (x_b y_v - x_a y_v - y_b x_v + y_a x_v)K \\ &= |\vec{ba}|^2 (\vec{a} \times \vec{q})(\vec{b} \times \vec{q}), \end{aligned} \quad (\text{A.35})$$

and since $|\vec{ab}|^2 > 0$, we consider

$$M = (\vec{a} \times \vec{q})(\vec{b} \times \vec{q}) = (x_a y_v - y_a x_v)(x_b y_v - y_b x_v). \quad (\text{A.36})$$

There are four cases.

1. If $M = 0$, this situation corresponds when $x_a y_v - y_a x_v = 0$ or $x_b y_v - y_b x_v = 0$, namely,

$$\frac{x_a}{y_a} = \frac{x_v}{y_v} \quad \text{or} \quad \frac{x_b}{y_b} = \frac{x_v}{y_v}. \quad (\text{A.37})$$

Each one corresponds to the case that a (or b) is located on the trajectory of the query object. We can exclude this situation from the consideration.

2. If $M > 0$, firstly consider the case when $(x_a y_v - y_a x_v > 0) \wedge (x_b y_v - y_b x_v > 0)$. This situation corresponds to the case $x_a/y_a > x_v/y_v$ and $x_b/y_b > x_v/y_v$. It means that two points a and b are located on the upper side of the trajectory. Since $M > 0$, function H takes zero's for two different t values. Given values for \vec{a} , \vec{b} , \vec{q} , we can easily obtain the two t values using the *quadratic formula*:

$$\begin{aligned} t &= \frac{-J \pm \sqrt{J^2 - 4IK}}{2I} \\ &= \frac{-|\vec{q}|(\vec{a} \times \vec{b}) \pm |\vec{a}\vec{b}|\sqrt{(\vec{a} \times \vec{q})(\vec{b} \times \vec{q})}}{|\vec{q}|(\vec{a}\vec{b} \times \vec{q})}. \end{aligned} \quad (\text{A.38})$$

Second, when $(x_a y_v - y_a x_v < 0) \wedge (x_b y_v - y_b x_v < 0)$, the situation is the same except that a and b are located on the lower side of the trajectory. This case corresponds to case A in Section 2.6.4.

3. If $M < 0$, the situation corresponds to the case $(x_a y_v - y_a x_v > 0) \wedge (x_b y_v - y_b x_v < 0)$ or $(x_a y_v - y_a x_v < 0) \wedge (x_b y_v - y_b x_v > 0)$. It means that a and b are located in the different sides of the trajectory. In this case, since $M < 0$, function H does not have solutions. This situation is case B in Section 2.6.4.

Note that if $I = (x_v^2 + y_v^2)(x_b y_v - x_a y_v - y_b x_v + y_a x_v) > 0$, namely, if $x_b y_v - x_a y_v - y_b x_v + y_a x_v > 0$, $H > 0$ always holds. On the other hand, if $x_b y_v - x_a y_v - y_b x_v + y_a x_v < 0$, $H < 0$ always holds. Now remember that function F is defined as $F = 2GH$ and G is defined as $G = (x_b y_v - x_a y_v - y_b x_v + y_a x_v)t + (x_a y_b - y_a x_b)$. If $x_b y_v - x_a y_v - y_b x_v + y_a x_v > 0$, G is a monotonically increasing function and if $x_b y_v - x_a y_v - y_b x_v + y_a x_v < 0$, G is a monotonically decreasing function. By combining these results, we can say that F is a monotonically increasing function. Thus, $\cos \varphi_{ab}$ has one local minimum.

4. If \vec{ab} is parallel to \vec{q} , we need to consider an exceptional case. This corresponds to case C in Section 2.6.4:

$$x_b y_v - x_a y_v - y_b x_v + y_a x_v = 0. \quad (\text{A.39})$$

It means that $I = 0$ (Eq. (A.29)) for function $H = It^2 + Jt + K$ (Eq. (A.28)). Therefore, $H = 2|\vec{q}|^2(\vec{a} \times \vec{b})t + K$ is a linear function. In this case, note that G is a constant: $G = \vec{a} \times \vec{b}$. Therefore, $F = 2GH = 2|\vec{q}|^2(\vec{a} \times \vec{b})^2 t + (\vec{a} \times \vec{b})K$

and it is a monotonically increasing function. It is concluded that $\cos \varphi_{ab}$ has one local minimum. Thus, F takes a local minimum when $H = 0$. By taking $H = 0$, we get

$$t = \frac{|\vec{a}|^2(\vec{b} \times \vec{q}) - |\vec{b}|^2(\vec{a} \times \vec{q})}{2|\vec{q}|^2(\vec{b} \times \vec{a})}. \quad (\text{A.40})$$

At this t -value, $\cos \varphi_{ab}$ takes a local minimum.

A.4 Proofs of Property 8 and Property 9

Proof 5 (Proof of Property 8). Assume object p is dominated by object p_i when q is located at v_i , and object p is dominated by object p_j when q is located at v_j . Based on Property 5, object p_i locates on the shortest path from v_i to p , and object p_j locates on the shortest path from v_j to p . If Property 8 is not valid, there must be at least one point q' of $e(v_i, v_j)$ such that p is not dominated when q is located at q' . As $q' \in e(v_i, v_j)$, the shortest path from q' to p must pass either v_i or v_j . If $SP(q', p)$ passes v_i , $SP(q', p) = SP(q', v_i) \cup SP(v_i, p)$ which means object p_i still locates on the shortest path from q' to p , and hence p_i dominates p . The same analysis holds for v_j . Consequently, the assumption that when q locates at q' , object p is not dominated is invalid, and our proof completes. ■

Proof 6 (Proof of Property 9). Assume Property 9 is invalid, there must be at least one point $q' \in e(v_i, v_j)$ such that when q locates at q' , object p is not a DBS point (i.e., $\exists p'$ such that $p' \prec p$). Based on Property 5, object p' locates on the shortest path from q' to p . If $SP(q', p)$ passes v_i , object p' should locate on the shortest path from v_i to p . Consequently, q' dominates object p when $q = v_i$ which contradicts the fact that p is a DBS point when $q = v_i$. The same analysis holds for v_j . Thus, our assumption is invalid and the proof completes. ■

Bibliography

- [1] Adrian C. Ott. *The 24-Hour Customer: New Rules for Winning in a Time-Starved, Always-Connected Economy*. HarperBusiness, 2010. ISBN 0061798614.
- [2] Bing Liu. *Web Data Mining: Exploring Hyperlinks, Contents, and Usage Data (Data-Centric Systems and Applications)*. Springer-Verlag, 2006. ISBN 3540378812.
- [3] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schtze. *Introduction to Information Retrieval*. Cambridge University Press, 2008. ISBN 0521865719, 9780521865715.
- [4] A. Rajaraman and J.D. Ullman. *Mining of Massive Datasets*. Cambridge University Press, 2011. ISBN 9781107015357.
- [5] Apostolos N. Papadopoulos. *Nearest Neighbor Search: A Database Perspective*. Springer-Verlag, 2004. ISBN 0387229639.
- [6] Zhexuan Song and Nick Roussopoulos. K-nearest neighbor search for moving query point. In *Proc. of the Symp. on Spatial and Temporal Databases (SSTD)*, pages 79–96, 2001.
- [7] Yufei Tao, Dimitris Papadias, and Qiongmao Shen. Continuous nearest neighbor search. In *Proc. of the Int’l Conf. on Very Large Data Bases (VLDB)*, pages 287–298, 2002.
- [8] Ken C. K. Lee, Wang-Chien Lee, and Hong Va Leong. Nearest surrounder queries. *IEEE Trans. Knowl. Data Eng.*, 22(10):1444–1458, 2010. DOI: 10.1109/TKDE.2009.172.
- [9] Ken C. K. Lee, Josh Schiffman, Baihua Zheng, Wang-Chien Lee, and Hong Va Leong. Round-eye: A system for tracking nearest surrounders in moving object

- environments. *Journal of Systems and Software*, 80(12):2063–2076, 2007. DOI: 10.1016/j.jss.2007.03.007.
- [10] Stephan Börzsönyi, Donald Kossmann, and Konrad Stocker. The skyline operator. In *Proc. of the IEEE Int’l Conf. on Data Engineering (ICDE)*, pages 421–430, 2001. DOI: 10.1109/ICDE.2001.914855.
- [11] Ihab F. Ilyas, George Beskales, and Mohamed A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4): 11:1–11:58, October 2008. DOI: 10.1145/1391729.1391730.
- [12] Md. Anisuzzaman Siddique and Yasuhiko Morimoto. Algorithm for computing convex skyline objectsets on numerical databases. *IEEE Trans. Inf. & Syst.*, 93-D(10):2709–2716, 2010. DOI: 10.1587/transinf.E93.D.2709.
- [13] I-Fang Su, Yu-Chi Chung, and Chiang Lee. Top- k combinatorial skyline queries. In *Proc. of the Int’l Conf. on Database Systems for Advanced Applications (DASFAA)*, pages 79–93, 2010. DOI: 10.1007/978-3-642-12098-5_6.
- [14] Yunjun Gao, Baihua Zheng, Gang Chen, Chun Chen, and Qing Li. Continuous nearest-neighbor search in the presence of obstacles. *ACM Trans. Database Syst.*, 36(2):9:1–9:43, 2011. DOI: 10.1145/1966385.1966387.
- [15] Yunjun Gao, Baihua Zheng, Gencai Chen, Qing Li, and Xiaofa Guo. Continuous visible nearest neighbor query processing in spatial databases. *The VLDB Journal*, 20(3):371–396, 2011. DOI: 10.1007/s00778-010-0200-z.
- [16] Sarana Nutanong, Egemen Tanin, and Rui Zhang. Visible nearest neighbor queries. In *Proc. of the Int’l Conf. on Database Syst. for Advanced Applications (DASFAA)*, pages 876–883, 2007. DOI: 10.1007/978-3-540-71703-4_73.
- [17] Muhammad Aamir Cheema, Wenjie Zhang, Xuemin Lin, Ying Zhang, and Xuefei Li. Continuous reverse k nearest neighbors queries in euclidean space and in spatial networks. *The VLDB Journal*, 21(1):69–95, 2012. DOI: 10.1007/s00778-011-0235-9.
- [18] Dimitris Papadias and Yufei Tao. Reverse nearest neighbor query. In *Encyclopedia of Database Systems*, pages 2434–2438. 2009. DOI: 10.1007/978-0-387-39940-9_318.

- [19] Wei Wu, Fei Yang, Chee Yong Chan, and Kian-Lee Tan. Continuous reverse k-nearest-neighbor monitoring. In *Proc. of the Int'l Conf. on Mobile Data Management (MDM)*, pages 132–139, 2008. DOI: 10.1109/MDM.2008.31.
- [20] J.H. Schiller and A. Voisard. *Location-Based Services*. Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann, 2004. ISBN 9781558609297.
- [21] R.H. Güting and M. Schneider. *Moving Objects Databases*. Morgan Kaufmann Series in Data Management Systems. Elsevier, 2005. ISBN 9780120887996.
- [22] Yifan Li, Jiong Yang, and Jiawei Han. Continuous k-nearest neighbor search for moving objects. In *Proc. of the Int'l Conf. on Scientific and Statistical Database Management (SSDBM)*, pages 123–126, 2004. DOI: 10.1109/SSDBM.2004.24.
- [23] Hyung-Ju Cho and Chin-Wan Chung. An efficient and scalable approach to CNN queries in a road network. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB)*, pages 865–876, 2005.
- [24] Zaiben Chen, Heng Tao Shen, Xiaofang Zhou, and Jeffrey Xu Yu. Monitoring path nearest neighbor in road networks. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data (SIGMOD)*, pages 591–602, 2009. DOI: 10.1145/1559845.1559907.
- [25] Haibo Hu, Dik Lun Lee, and Jianliang Xu. Fast nearest neighbor search on road networks. In *Proc. of the Int'l Conf. on Extending Database Technology (EDBT)*, pages 186–203, 2006. DOI: 10.1007/11687238_14.
- [26] Jan Chomicki, Parke Godfrey, Jarek Gryz, and Dongming Liang. Skyline with pre-sorting. In *Proc. of the IEEE Int'l Conf. on Data Engineering (ICDE)*, pages 717–719, 2003. DOI: 10.1109/ICDE.2003.1260846.
- [27] Parke Godfrey, Ryan Shipley, and Jarek Gryz. Maximal vector computation in large data sets. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB)*, pages 229–240, 2005. DOI: 10.1.1.60.4954.
- [28] Dimitris Papadias, Yufei Tao, Greg Fu, and Bernhard Seeger. Progressive skyline computation in database systems. *ACM Trans. Database Syst.*, 30(1):41–82, 2005. DOI: 10.1145/1061318.1061320.

- [29] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data (SIGMOD)*, pages 47–57, 1984. DOI: 10.1145/602259.602266.
- [30] Y. Collette and P. Siarry. *Multiobjective Optimization: Principles and Case Studies*. Decision Engineering. Springer, 2004. ISBN 9783540401827.
- [31] Kalyanmoy Deb and Deb Kalyanmoy. *Multi-Objective Optimization Using Evolutionary Algorithms*. Wiley, 2001. ISBN 047187339X.
- [32] Zhiyong Huang, Christian S. Jensen, Hua Lu, and Beng Chin Ooi. Skyline queries against mobile lightweight devices in manets. In *Proc. of the IEEE Int'l Conf. on Data Engineering (ICDE)*, page 66, 2006. DOI: 10.1109/ICDE.2006.142.
- [33] Akrivi Vlachou, Christos Doukeridis, and Yannis Kotidis. Angle-based space partitioning for efficient parallel skyline computation. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data (SIGMOD)*, pages 227–238, 2008. DOI: 10.1145/1376616.1376642.
- [34] Zhiyong Huang, Hua Lu, Beng Chin Ooi, and Anthony K. H. Tung. Continuous skyline queries for moving objects. *IEEE Trans. Knowl. Data Eng.*, 18(12):1645–1658, 2006. DOI: 10.1109/TKDE.2006.185.
- [35] Baihua Zheng, Ken C. K. Lee, and Wang-Chien Lee. Location-dependent skyline query. In *Proc. of the Int'l Conf. on Mobile Data Management (MDM)*, pages 148–155, 2008. DOI: 10.1109/MDM.2008.14.
- [36] Kostas Patroumpas and Timos K. Sellis. Monitoring orientation of moving objects around focal points. In *Proc. of the Symp. on Spatial and Temporal Databases (SSTD)*, pages 228–246, 2009. DOI: 10.1007/978-3-642-02982-0_16.
- [37] Nan Chen, Lidan Shou, Gang Chen, Yunjun Gao, and Jinxiang Dong. Predictive skyline queries for moving objects. In *Proc. of the Int'l Conf. on Database Systems for Advanced Applications (DASFAA)*, pages 278–282, 2009. DOI: 10.1007/978-3-642-00887-0_23.
- [38] Mu-Woong Lee and Seung-won Hwang. Continuous skylining on volatile moving data. In *Proc. of the IEEE Int'l Conf. on Data Engineering (ICDE)*, pages 1568–1575, 2009. DOI: 10.1109/ICDE.2009.162.

- [39] Xuegang Huang and Christian S. Jensen. In-route skyline querying for location-based services. In *Proc. of the Int'l Symp. on Web and Wireless Geographical Information Systems (W2GIS)*, pages 120–135, 2004. DOI: 10.1007/11427865_10.
- [40] Simonas Saltenis, Christian S. Jensen, Scott T. Leutenegger, and Mario A. Lopez. Indexing the positions of continuously moving objects. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data (SIGMOD)*, pages 331–342, 2000. DOI: 10.1145/342009.335427.
- [41] Mehdi Sharifzadeh and Cyrus Shahabi. The spatial skyline queries. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB)*, pages 751–762, 2006.
- [42] Wanbin Son, Mu-Woong Lee, Hee-Kap Ahn, and Seung-Won Hwang. Spatial skyline queries: An efficient geometric algorithm. In *Proc. of the Symp. on Spatial and Temporal Databases (SSTD)*, pages 247–264, 2009. DOI: 10.1007/978-3-642-02982-0_17.
- [43] Katerina Raptopoulou, Apostolos Papadopoulos, and Yannis Manolopoulos. Fast nearest-neighbor query processing in moving-object databases. *GeoInformatica*, 7(2):113–137, 2003. DOI: 10.1023/A:1023403908170.
- [44] Xi Guo, Yoshiharu Ishikawa, and Yunjun Gao. Direction-based spatial skylines. In *Proc. of the Int'l ACM Workshop on Data Engineering for Wireless and Mobile Access (MobiDE)*, pages 73–80, 2010. DOI: 10.1145/1850822.1850835.
- [45] Xi Guo, Baihua Zheng, Yoshiharu Ishikawa, and Yunjun Gao. Direction-based surrounder queries for mobile recommendations. *The VLDB Journal*, 20(5):743–766, October 2011. DOI: 10.1007/s00778-011-0241-y.
- [46] GNU. Gnu scientific library. Website, 2011. <http://www.gnu.org/software/gsl/>.
- [47] U.S. Census Bureau. Tiger, u.s. census bureau. Website, 1990. <http://tiger.census.gov/>.
- [48] Norio Katayama. R*-tree library. Website, 1997. <http://research.nii.ac.jp/~katayama/homepage/research/srtree/English.html>.
- [49] Dimitris Papadias, Jun Zhang, Nikos Mamoulis, and Yufei Tao. Query processing in spatial network databases. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB)*, pages 802–813, 2003.

- [50] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.
- [51] Mohammad R. Kolahdouzan and Cyrus Shahabi. Voronoi-based k nearest neighbor search for spatial network databases. In *Proc. of the Int’l Conf. on Very Large Data Bases (VLDB)*, pages 840–851, 2004.
- [52] Haibo Hu, Dik Lun Lee, and Victor C. S. Lee. Distance indexing on road networks. In *Proc. of the Int’l Conf. on Very Large Data Bases (VLDB)*, pages 894–905, 2006.
- [53] Ken C. K. Lee, Wang-Chien Lee, and Baihua Zheng. Fast object search on road networks. In *Proc. of the Int’l Conf. on Extending Database Technology (EDBT)*, pages 1018–1029, 2009. DOI: 10.1145/1516360.1516476.
- [54] Mohammad R. Kolahdouzan and Cyrus Shahabi. Alternative solutions for continuous k nearest neighbor queries in spatial network databases. *GeoInformatica*, 9(4):321–341, 2005. DOI: 10.1007/s10707-005-4575-8.
- [55] Cyrus Shahabi, Mohammad R. Kolahdouzan, and Mehdi Sharifzadeh. A road network embedding technique for k-nearest neighbor search in moving object databases. *GeoInformatica*, 7(3):255–273, 2003. DOI: 10.1023/A:1025153016110.
- [56] Christian S. Jensen, Jan Kol’arvr, Torben Bach Pedersen, and Igor Timko. Nearest neighbor queries in road networks. In *Proc. of the ACM SIGSPATIAL Int’l Conf. on Advances in Geographic Information Systems (ACM SIGSPATIAL GIS)*, pages 1–8, 2003. DOI: 10.1145/956676.956677.
- [57] Kyriakos Mouratidis, Man Lung Yiu, Dimitris Papadias, and Nikos Mamoulis. Continuous nearest neighbor monitoring in road networks. In *Proc. of the Int’l Conf. on Very Large Data Bases (VLDB)*, pages 43–54, 2006.
- [58] Ugur Demiryurek, Farnoush Banaei-Kashani, and Cyrus Shahabi. Efficient continuous nearest neighbor query in spatial networks using euclidean restriction. In *Proc. of the Symp. on Spatial and Temporal Databases (SSTD)*, pages 25–43, 2009. DOI: 10.1007/978-3-642-02982-0_5.
- [59] Hanan Samet, Jagan Sankaranarayanan, and Houman Alborzi. Scalable network distance browsing in spatial databases. In *Proc. of the ACM SIGMOD Int’l Conf. on Management of Data (SIGMOD)*, pages 43–54, 2008. DOI: 10.1145/1376616.1376623.

- [60] Ning Jing, Yun-Wu Huang, and Elke A. Rundensteiner. Hierarchical encoded path views for path query processing: An optimal model and its performance evaluation. *IEEE Trans. on Knowl. and Data Eng.*, 10(3):409–432, May 1998. DOI: 10.1109/69.687976.
- [61] Fang Wei. Tedi: efficient shortest path query answering on graphs. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data (SIGMOD)*, pages 99–110, 2010. DOI: 10.1145/1807167.1807181.
- [62] Thomas Brinkhoff. A framework for generating network-based moving objects. *GeoInformatica*, 6(2):153–180, 2002. DOI: 10.1023/A:1015231126594.
- [63] Man Lung Yiu and Nikos Mamoulis. Clustering objects on a spatial network. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data (SIGMOD)*, pages 443–454, 2004. DOI: 10.1145/1007568.1007619.
- [64] Feifei Li, Dihan Cheng, Marios Hadjieleftheriou, George Kollios, and Shang-Hua Teng. On trip planning queries in spatial databases. In *Proc. of the Symp. on Spatial and Temporal Databases (SSTD)*, pages 273–290, 2005. DOI: 10.1007/11535331_16.
- [65] Lingkun Wu, Xiaokui Xiao, Dingxiong Deng, Gao Cong, Andy Diwen Zhu, and Shuigeng Zhou. Shortest path and distance queries on road networks: An experimental evaluation. *Proceedings of the VLDB Endowment (PVLDB)*, 5(5):406–417, 2012.
- [66] Xuemin Lin, Yidong Yuan, Qing Zhang, and Ying Zhang. Selecting stars: The k most representative skyline operator. In *Proc. of The IEEE Int'l Conf. on Data Engineering (ICDE)*, pages 86–95, 2007. DOI: 10.1109/ICDE.2007.367854.
- [67] Atish Das Sarma, Ashwin Lall, Danupon Nanongkai, Richard J. Lipton, and Jun (Jim) Xu. Representative skylines using threshold-based preference distributions. In *Proc. of the IEEE Int'l Conf. on Data Engineering (ICDE)*, pages 387–398, 2011. DOI: 10.1109/ICDE.2011.5767873.
- [68] Yufei Tao, Ling Ding, Xuemin Lin, and Jian Pei. Distance-based representative skyline. In *Proc. of the IEEE Int'l Conf. on Data Engineering (ICDE)*, pages 892–903, 2009. DOI: 10.1109/ICDE.2009.84.

- [69] Senjuti Basu Roy, Sihem Amer-Yahia, Ashish Chawla, Gautam Das, and Cong Yu. Constructing and exploring composite items. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data (SIGMOD)*, pages 843–854, 2010. DOI: 10.1145/1807167.1807258.
- [70] Qian Wan, Raymond Chi-Wing Wong, and Yu Peng. Finding top-k profitable products. In *Proc. of the IEEE Int'l Conf. on Data Engineering (ICDE)*, pages 1055–1066, 2011. DOI: 10.1109/ICDE.2011.5767895.
- [71] Matthias Ehrgott and Xavier Gandibleux. A survey and annotated bibliography of multiobjective combinatorial optimization. *OR Spectrum*, 22(4):425–460, 2000.
- [72] Hans Kellerer, Ulrich Pferschy, and David Pisinger. *Knapsack Problems*. Springer, 2004. ISBN 3540402861.
- [73] Md. Mostofa Akbar, Mohammad Sohel Rahman, Mohammad Kaykobad, Eric G. Manning, and Gholamali C. Shoja. Solving the multidimensional multiple-choice knapsack problem by constructing convex hulls. *Computers & OR*, 33:1259–1273, 2006. DOI: 10.1016/j.cor.2004.09.016.
- [74] Krzysztof Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003. ISBN 0521825830.
- [75] Dimitris Papadias, Nikos Mamoulis, and Vasilis Delis. Algorithms for querying by spatial structure. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB)*, pages 546–557, 1998.
- [76] Xi Guo and Yoshiharu Ishikawa. Multi-objective optimal combination queries. In *Proc. of the Int'l Conf. on Database and Expert Systems Applications (DEXA)*, pages 47–61, 2011. DOI: 10.1007/978-3-642-23088-2_4.
- [77] Xi Guo, Chuan Xiao, and Yoshiharu Ishikawa. Combination skyline queries. *Transactions on Large-Scale Data- and Knowledge Centered Systems VI*, 7600:1–30, 2012. DOI: 10.1007/978-3-642-34179-3_1.
- [78] Marios Hadjieleftheriou, Erik G. Hoel, and Vassilis J. Tsotras. Sail: A spatial index library for efficient application integration. *GeoInformatica*, 9(4):367–389, 2005. DOI: 10.1007/s10707-005-4577-6.

-
- [79] Chee Yong Chan, H. V. Jagadish, Kian-Lee Tan, Anthony K. H. Tung, and Zhenjie Zhang. On high dimensional skylines. In *Proc. of the Int'l Conf. on Extending Database Technology (EDBT)*, pages 478–495, 2006. DOI: 10.1007/11687238_30.
- [80] Tobin J. Lehman and Michael J. Carey. A study of index structures for main memory database management systems. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB)*, pages 294–303, 1986.
- [81] Yufei Tao, Dimitris Papadias, and Xiang Lian. Reverse k NN search in arbitrary dimensionality. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB)*, pages 744–755, 2004.

List of Publications

Journal Papers

- Xi Guo, Baihua Zheng, Yoshiharu Ishikawa, and Yunjun Gao, “Direction-Based Surrounding Queries for Mobile Recommendations”, *The VLDB Journal*, Vol. 20, No. 5, pp. 743–766, 2011.
- Xi Guo, Chuan Xiao, and Yoshiharu Ishikawa, “Combination Skyline Queries”, *Transactions on Large-Scale Data- and Knowledge Centered Systems*, VI, LNCS 7600, pp. 1–30, 2012.

International Conference/Workshop Papers

- Kazuki Kodama, Yuichi Iijima, Xi Guo, and Yoshiharu Ishikawa, “Skyline Queries Based on User Locations and Preferences for Location-Based Recommendations”, *Proceedings of the 2009 International Workshop on Location Based Social Networks (LBSN 2009)*, pp.9–16, Seattle, Washington, USA, November, 2009.
- Xi Guo, Yoshiharu Ishikawa, and Yunjun Gao, “Direction-Based Spatial Skylines”, *Proceedings of the 11th International ACM Workshop on Data Engineering for Wireless and Mobile Access (MobiDE 2010)*, pp.73–80, Indianapolis, Indiana, USA, June, 2010.
- Xi Guo and Yoshiharu Ishikawa, “Multi-Objective Optimal Combination Queries”, *Proceedings of the 22nd International Conference on Database and Expert Systems Applications (DEXA 2011)*, pp.47–61, Toulouse, France, August, 2011.
- Masanori Mano, Xi Guo, Tingting Dong, and Yoshiharu Ishikawa, “Privacy Preservation for Location-Based Services Based on Attribute Visibility”, *Proceedings of the*

2nd International Workshop on Information Management for Mobile Applications (IMMoA 2012), pp. 33–41, Istanbul, Turkey, August, 2012.