Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 EECS Building
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 00-044

Modeling and Processing Directional Relationships in Spatial Databases

Xuan Liu

August 21, 2000

Acknowledgements

I'd like to start with my sincere acknowledgement of the guidance, suggestions, support provided by my academic advisor Prof. Shashi Shekhar, whose advise and encouragement kept the research on the right track and made this thesis possible.

I wish to acknowledge the members in my thesis committee, Prof. Marvin Bauer, Prof. George Karypis, Prof. Nikolaos Papanikolopoulos, Prof. Jaideep Srivastava, Prof. Caroline Hayes. Their advice and comments and the time and effort they invested in me are highly appreciated. Many thanks go to Dr. Sanjay Chawla, who provided significant help and valuable suggestions through the process of thesis writing.

Computer Science Department and Army High Performance Computing Research Center at University of Minnesota have provided ideal working environments. My colleagues in spatial database group have been particularly supportive. I am grateful to Weili Wu, Chang-tien Lu, Ranga Raju Vatsavai, Xinhong Tan, Changqing Zhou, Xiaobin Ma, Hui Xiong, and Pusheng Zhang for making the office such a great place.

All of my friends deserve my heartfelt thanks and gratitude. They helped me in more ways than they can ever imagine and made my graduate student life wonderful. Special thanks to Tong Chen, Bixia Zheng, Difu Su, Lingzhou Pan, Leijun Zheng, Jian Huang, Ping-Yao Chen, and Shu-Ling Chang for their huge help and support.

Lastly, very special thanks go to my mother Ruzhen Liang, my brother Wenjie Liu, and my sister Hui Liu for their love, support and understanding during my whole study.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction to Spatial Databases

A spatial database [28, 35, 75] management system aims at the effective and efficient management of data related to a space such as the physical world (geography, urban planning, astronomy); parts of living organisms (anatomy of the human body); engineering design (very large scale integrated circuits, the design of an automobile or the molecular structure of a pharmaceutical drug); and conceptual information space (a multi-dimensional decision support system, fluid flow, or an electro-magnetic field). The distinguishing features of a spatial database management system are the use of complex data types like points, lines and polygons to represent spatial objects and the existence of many potential relationships between spatial objects.

The field of spatial database research has been an active area of research for over two decades. It's results, e.g. spatial multi-dimensional indexes, are being used in a number of areas. The field of spatial databases can be defined by its accomplishments, and current research is aimed at improving its functionality and its performance. The impetus for improving functionality comes from the needs of existing applications such as Geographic Information Systems (GIS) and Computer Aided Design (CAD), as well as from potential applications such as Multimedia Information System(MMIS), Data Warehousing (DWH) and NASA's Earth Observation System (EOS). The acceptance of GIS as an important tool in government decision-making is also documented [73, 11] and

military planners have embraced GIS technology at all levels of tactical, operational and strategic planning, including battlefield visualization and terrain analysis [46].

Commercial examples of spatial database management system include Informix's spatial data-blades (i.e. 2D, 3D, Geodetic), Oracle's Universal server with either Spatial Data Option or Spatial Data Cartridge and ESRI's Spatial Database Engine (SDE). Research prototype examples of spatial database management systems include spatial datablades with Postgres [66], GeO2, and Paradise[14]. The functionalities provided by these systems include:

- A set of spatial data types such as a point, line-segment and polygon, and a set of spatial operations such as inside, intersection, and distance.
- The spatial types and operations may be made a part of a query language such as SQL, which allows spatial querying when combined with an object-relational database management system [10, 68].
- The performance enhancement provided by these systems includes a multi-dimensional spatial index and algorithms for spatial access methods, spatial range queries and spatial joins. Spatial indexing with concurrency control may be implemented in the object-relational server for performance reasons.

Existing and emerging applications require new functionalities including the modeling of new spatial data types, e.g., oriented objects, open shapes. The performance needs of emerging applications require not only the management of large data-sets, but also new processing strategies for new spatial operations such as directional operations.

The role of the spatial database component is dependent on the type of database management system (DBMS) involved: relational, object-oriented or object-relational. Spatial databases have been one of the most common applications of object-relational databases[10, 68, 67] and have influenced their design a great deal. Object-relational databases allow the inclusion of spatial data-types, spatial operations, and multi-dimensional indexing systems. This three-layer architectural framework is shown in Figure 1.1, and

it consists of an object-relational database management system, a spatial database, and a spatial application such as a GIS or MMIS.

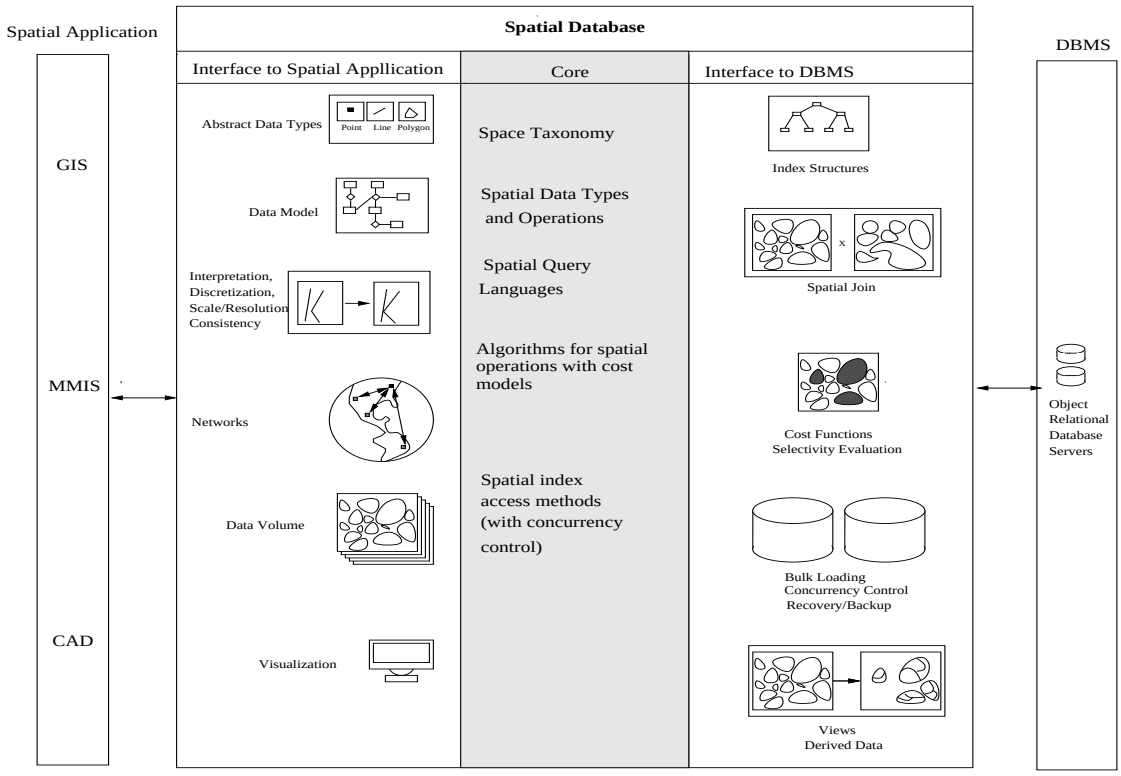


Figure 1.1: 3-layer architecture

The interface between the application and the spatial data system maps application-specific constructs to the spatial database. The spatial database associates the application requirements to the functionality provided by the DBMS. The interface to the DBMS supports specialized query processing, which in turn supports the core database requirements for achieving acceptable performance.

1.2 Spatial Data Model and Query Processing

Research into spatial databases has mainly focused on developing a space taxonomy, data models, query languages, query processing, and spatial access methods. Recent reports[28, 35, 75, 2, 58] have described the accomplishments of spatial database research and have prioritized research needs. A broad survey of spatial database requirements and an overview of research results is provided by [75, 28, 2, 60]. In this section, we will briefly review the accomplishments and research needs of spatial data model and spatial query processing.

A spatial data model [60, 75] is a type of data-abstraction that hides the details of data-storage. There are two common models of spatial information: field-based and object-based. The field-based model treats spatial information such as altitude, rainfall and temperature as a collection of spatial functions transforming a space-partition to an attribute domain. The object-based model treats the information space as if it is populated by discrete, identifiable, spatially-referenced entities. The operations in object model include *distance* and *boundary*. The operations on fields include local, focal, and zonal operations [72, 5], as shown in Table 1.1.

Data model	Operator Group	Operation
Vector Object	Set-Oriented	equals, is a member of, is empty, is a subset of, is disjoint from, intersection, union, difference, cardinality
	Topological	boundary, interior, closure, meets, overlaps, is inside, covers, connected, components, extremes, is within
	Metric	distance, bearing/angle, length, area, perimeter, centroid
	Directional	east, north, northwest, left, front, between
	Network	successors, ancestors, connected, shortest-path
Raster field	Local	Point-wise sums, differences, maximums, means, etc
	Focal	slope, aspect, weighted average of neighborhood
	Zonal	sum or mean or maximum of field values in each zone

Table 1.1: A Sample of Spatial Operations

An implementation of a spatial data model consists of a set of spatial data types and the operations on those types. Much work has been done over the last decade on the design of spatial Abstract Data Types(ADTs) and their embedding in a query language. Consensus is slowly emerging via standardization efforts like OGIS [24]. Figure

1.2 illustrates the spatial data-type hierarchy of OGIS which consists of points, curves and surfaces. The basic operations specified by OGIS [24] operative on *all* data types

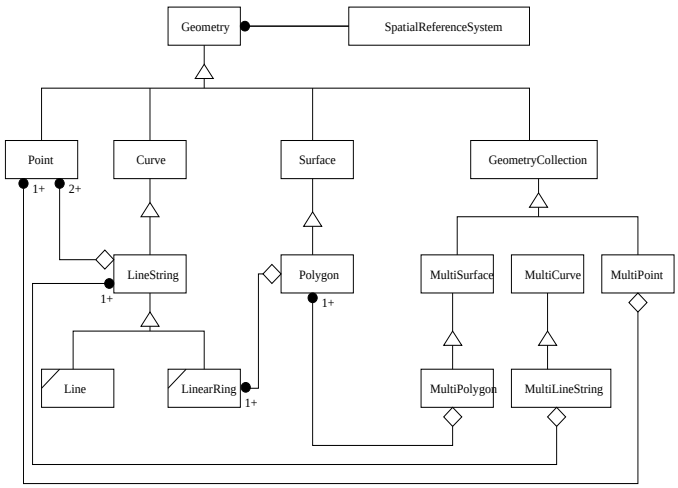


Figure 1.2: Spatial Data Type Hierarchy[21]

are shown in Table 1.2. The topological operations are based on the ubiquitous 9-intersection model [16]. Using the OGIS specification, common spatial queries can be intuitively posed in SQL[16].

Spatial queries are often processed using *filter* and *refine* techniques. Approximate geometry such as the Minimal Bounding Rectangle(MBR) of an extended spatial object is first used to filter out many irrelevant objects quickly. Exact geometry is then used for the remaining spatial objects to complete the processing. Strategies for range-queries include a scan and index-search in conjunction with the plane-sweep algorithm [9]. Strategies for the spatial-join include the nested loop, tree matching [9] when indices are present on all participating relations, and space partitioning [50] in the absence of indices. To speed up computation for large spatial objects (it is common for polygons to have 1000 or more edges), object indices are used in extended filtering.

A large number of spatial indices [57, 39, 63, 36, 61, 41] have been explored for multi-dimensional euclidean space. One of the first access methods created to handle extended objects was Guttman's R-tree structure [29]. The R-tree is a height balanced natural extension of the B+ tree for higher dimensions. Objects are represented in the R-tree

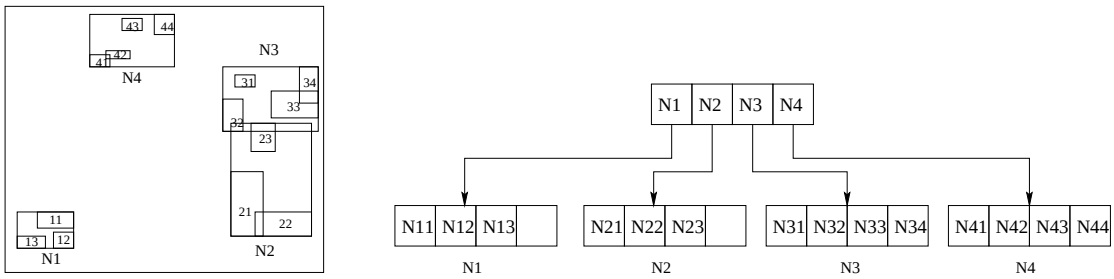
Basic Functions	SpatialReference()	Returns the Reference System of the geometry
	Envelope()	The minimum bounding rectangle of the geometry
	Export()	Convert the geometry into a different representation.
	IsEmpty()	Tests if the geometry is a empty set or not
	IsSimple()	Returns True if the geometry is simple(no self-intersection)
	Boundary()	Returns the boundary of the geometry
Topological/ Set Operators	Equal	Tests if the geometries are spatially equal
	Disjoint	Tests if the geometries are disjoint
	Intersect	Tests if the geometries intersect
	Touch	Tests if the geometries touch each other
	Cross	Tests if the geometries cross each other
	Within	Tests if the given geomtry is within another given geometry
	Contains	Tests if the given geometry contains another given geometry
	Overlap	Tests if the geometry overlaps another geometry
Spatial Analysis	Distance	Returns the shortest distance between two geometries
	Buffer	Returns a geometry that represents all points whose distance from the given is less than or equal to the specified distance
	ConvexHull	Returns the convex hull of the geometry
	Intersection	Returns the intersection of two geometries
	Union	Returns the union of two geometries
	Difference	Returns the difference of two geometries
	SymDiff	Returns the symmetric difference of two geometries

Table 1.2: Representative functions specified by OGIS

by their minimum bounding rectangles(MBRs). Nonleaf nodes are composed of entries of the form $(R, child - pointer)$, where R is the MBR of all entries contained in the *child-pointer*. Leaf nodes contain the MBRs of the data objects. To guarantee good space utilization and height-balance, the parent MBRs are allowed to overlap. Figure 1.3(a) illustrates the spatial objects, while Figure 1.3(b) shows the corresponding R-tree. Many variations of the R-tree structure exist whose main emphasis is on discovering new strategies to maintain the balance of the tree, in case of a split, and to minimize the overlap of the MBRs in order to improve the search time.

Spatial databases are being used for an increasing number of new applications, such as Intelligent Transportation Systems and NASA's Earth Observation System. However, as we notice in Table 1.2, the OGIS specification is confined to topological [12] and metric operations on vector data types. More work is needed to incorporate relationships such as directional [64] operations (see Table 1.1 for examples).

Many processing strategies using the overlap predicate have been developed for range



(a) Examples Objects

(b) Corresponding R-trees

Figure 1.3: Example data objects and the corresponding R-tree

queries and spatial join queries. However, there is a need to develop and evaluate strategies for many other frequent queries. These include queries on objects using predicates other than overlap such as as directional queries, as well as queries on networks such as the shortest path to a set of destinations.

1.3 Modeling and Processing Directional Relationships

Direction is a common spatial concept that is used everywhere in daily life. When people communicate about a geographic space, direction is necessary to convey the information. For instance, direction concepts and direction reasoning are important when people make decision on the layout of new facilities, or on the location of constructing a new building. Directional relationships are also critical in visual and natural language interfaces to Geographic information systems(GIS). For example, GIS should be able to interpret the semantics of the spatial constraints and infer the implicit directional relationship.

Direction is also frequently used as a selection condition in spatial queries[28] or used for similarity accessing in image databases [53]. Examples of direction queries used in army battlefield visualization [27] are "List the enemy targets to the *north* of Building B","Is there anything *over* the ridge?", and "Describe all tanks to the *left* of Landmark L". The first example refers to an absolute directional relationship(*north*) with respect to Building B, the second involves a viewer-orientation-based directional relationship, i.e. the direction is based on the orientation of the viewer, and the third example uses an object-orientation-based direction as its selection criteria.

It is important for a spatial database system to provide a mechanism for modeling and processing direction queries. Figure 1.4 illustrates an example framework. The framework consists of three basic functional components. The first component is direction modeling, which formalizes different direction models and the corresponding ADTs for incorporating direction concepts and directional relationships into spatial databases. The second component deals with the data integrity in databases. The subsystem *consistency checking* checks the consistency among the specified directional constraints among objects in databases, and the subsystem it inferencing provides an inferencing mechanism for qualitative direction reasoning. The last component is the direction query processing. It returns the objects satisfying the search criteria to users.

In this thesis, we aim to explore these components, namely direction modeling, direction inferencing, consistency checking and directional query processing. The issues

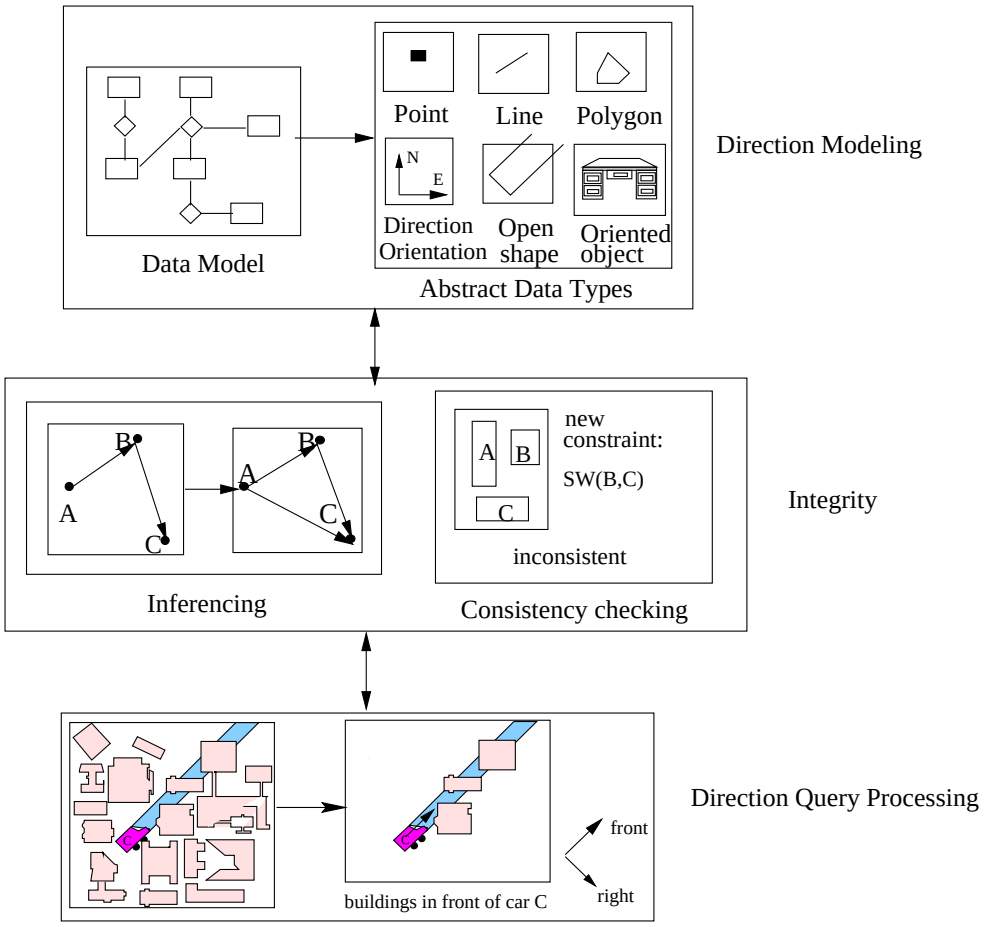


Figure 1.4: Framework for Modeling and Processing directional relationships

involved are addressed in the following chapters. Among these chapters, Chapter 2 deals with the direction modeling in a general 3D Cartesian coordinate system, while chapter 3 and chapter 4 focus on a more specific set of direction predicates in a 2D Cartesian coordinate system. Each chapter is organized to address one of the components in the framework and designed to be self-contained. The related work, motivation, and contributions for each topic are explained in each chapter separately. In order to keep each chapter self-contained, there may be some duplicated introduction to basic concepts. The organization of the rest chapters is as follows:

In chapter 2, we address the problem of direction modeling. As a departure from traditional methods which model the directions as binary boolean functions between

objects, we propose a novel vector based object model, i.e., model direction as a spatial object by using the concepts of vectors, points and angles. This object view of direction has several advantages. Being modeled as a spatial object, a direction object can have its own attributes and operation sets. Secondly, new spatial data types such as oriented spatial objects and open shapes can be defined at the abstract object level. Finally, the object view of direction makes direction reasoning easy and also reduces the need for a large number of inference rules. The applicability of the direction model is demonstrated by geographic query examples.

In chapter 3, we explore the direction query processing strategies. A new open shape based strategy (OSS) is proposed to process direction queries in spatial databases. Previous work on direction query processing has focused on processing absolute directions using range query strategies and R-tree indexing. However, many direction queries depend on the orientation of reference objects(or the viewer) which may change due to motion. Classical methods are inefficient when orientation of the reference object is different from the global reference system[43]. OSS strategy is designed to address this problem. OSS converts the processing of directional queries to the processing of topological operations between open shapes and objects. It eliminates false hits at the earliest opportunity while recursively searching hierarchical indices like R-tree. Since OSS models the direction region as an open shape, it also eliminates the computation related to the embedding world boundary. The algebraic analysis and experimental results demonstrate that OSS consistently outperforms classical strategies (often by an order of magnitude) in terms of both I/O and CPU cost.

In chapter 4, we address the problem of consistency checking for Euclidean spatial constraints. A dimension graph representation is proposed to maintain the Euclidean spatial constraints among objects. The basic idea is to project the spatial constraints on both X and Y dimensions, and the dimension graph is constructed on each dimension. By using the dimension graph representation, the problem of consistency checking is then transformed to a graph cycle detection problem[42]. The consistency checking can be achieved with O(N+E) time as well as space complexity, where N is the number

of spatial objects, and E is the number of spatial predicates in the constraint. The proposed approach to consistency checking for spatial constraints is faster than $O(N^2)$ when the number of predicates is much smaller than N^2 and there are few disjunctions in the spatial constraint. The dimension graph and consistency checking algorithm can be used for points, intervals and polygons in 2 dimensional space. The algorithm also guarantees the global consistency.

In chapter 5, we propose equivalence classes of direction objects to unify the inferencing of different set of directional predicates. By defining an algebra on equivalence classes we construct a framework to model the semantics of direction predicates. The equivalence classes of direction objects together with the algebra defined on it provide an independent interpretation model for qualitative direction reasoning. We also discuss the inferencing for viewer-orientation-based and object-orientation-based reference frames using equivalence classes. Many inferencing rules in previous work can be derived easily using our framework and thus reducing the large number of inference rules needed.

Chapter 2

An Object Model of Direction and Its Implications

Direction is an important spatial concept that is used in many fields such as geographic information systems(GIS)[21] and image interpretation. It is also frequently used as a selection condition in spatial queries. Previous work has modeled direction as a relational predicate between spatial objects. Conversely, in this thesis, we model direction as a new kind of spatial object using the concepts of vectors, points and angles. The basic approach is to model direction as a unit vector. Given an ordered pair (p_1, p_2) of spatial points, one can define a direction, which can take the values of North, Northwest, 3 o' clock, etc. to represent the corresponding qualitative directional predicates on (p_1, p_2) in previous work. This novel view of direction has several obvious advantages: Being modeled as a spatial object, a direction object can have its own attributes and operation set. Secondly, new spatial data types such as oriented spatial objects and open spatial objects can be defined at the abstract object level. Finally, the object view of direction makes direction reasoning easy and also reduces the need for a large number of inference rules. These features are important in spatial query processing and optimization. The applicability of the direction model is demonstrated by geographic query examples.

2.1 Introduction

2.1.1 Modeling Direction

Direction is a common spatial concept that is used everywhere in daily life[40]. When people communicate about a geographic space, for example, giving route descriptions, direction is necessary to convey the information. Direction is also frequently used as a selection condition in spatial queries[28] or used for similarity accessing in image databases[53]. Example queries used in army battlefield visualization[27] are "Is there anything *over* the ridge?," "List the swamps in *front* of the tank." and "Let's move to the *north* of the tree." The first example refers to a viewer-orientation-based direction, the second can be defined on either the intrinsic orientation of the tank (object-orientation-based) or of a viewer, and the third example refers to the absolute direction with respect to the tree. In order to process queries involving direction constraints in the selection criteria, a spatial database system should provide a way for users to formalize directions.

The common means of handling direction is to model direction as a spatial relationship between objects[18, 60, 25, 76, 15, 51, 26, 20, 62, 48]. In this chapter, we formalize direction from a different perspective: as a spatial object. The basic approach is to model direction as a unit vector and orientation as a set of directions. As a spatial object, direction can have its own attributes, and its own operators, and a richer set of predicates and operators on direction and orientation can be defined. Second, new spatial data types such as oriented spatial objects and unbounded spatial objects can be easily defined in the abstract object level. The object view of direction also makes it easy to perform direction reasoning by using only simple vector algebra, which also reduces the large number of inference rules commonly needed. This is useful in the processing and optimization of spatial queries that contain direction constraints. We model the direction for three frames of reference: absolute, object/viewer-orientation-based directions.

We will not discuss the philosophical issue of whether directions are object/entities or not in the strictest sense. Absolute directions such as North, South, East, or West can be defined using a coordinate system w.r.t. the earth, and thus these can be considered

to be objects in their own right. The object view for relative directions (e.g. left, front) needs further thought, because these directions are defined with respect to coordinate systems attached to other objects. The issue needs to be explored further.

2.1.2 Related work and our contributions

The research work on direction modeling has been carried out in several areas such as geographic information systems and image analysis. Most of the studies is on how to capture the semantics of direction relations, and further, how to do spatial reasoning on the direction [15, 18, 20]. There are two major direction reference frames used to model direction in 2D space: the cone-based model[51], and the projection-based model[18, 20]. Frank[3] compared these two models and found the projection-based reference frame to be better in many aspects. The most common way to model directions between extended objects is through the object's Minimum Bounding Rectangle(MBR), where direction relations are obtained by applying Allen's [4] interval relations along the x and y axis, in which case, 169 different relations[15] can be distinguished. Some work based on MBR has been proposed on picture indexing in pictorial databases[53, 76], and some work aligns each boundary box to the object's major axis[34], which makes it possible to satisfy different reference frames[26]. On the other hand, Freska [19] proposed an alternative method: semi-intervals to formalize the one-dimensional temporal relation based on incomplete knowledge of the object. Goyal and Egenhofer [25] introduced a Direction-Relation Matrix to represent cardinal directions. Based on the projection-based frame, it partitions the space around the reference object and records into which direction tiles a target object falls. But this model still has limitations in the modeling of line objects, and it is limited to 2D space. Little work has been done on directions in 3D space [23].

The previous work modeled direction as a spatial relationship between spatial objects. This seems to be a natural mapping of the directional relationship that is used in geographic space. But this modeling method has some limitations. Operations on direction are limited. Oriented(directed) objects and unbounded objects cannot be represented in

the spatial data model.

This chapter models direction as a spatial object: a unit vector. This novel view of direction has several advantages over the binary boolean relationship view. Being modeled as a spatial object, a direction object can have its own attributes and operation set. The implementation of operators can use vector algebra, making a richer set of predicates and operators on direction feasible. Secondly, new spatial data types such as oriented objects and unbounded objects can be defined at the abstract object level. The object view of direction also makes directional reasoning easy. Basic vector algebra is sufficient for inferencing new directions, and no special or new qualitative rules are needed. This reduces the complexity of directional reasoning, and involving appropriate quantitative information is useful in spatial query processing and optimization in spatial databases.

More discussion on direction objects, formal proofs, and their applications appear in [65], where an algebra on the equivalence classes of direction objects is defined. The algebra unifies different methods for directional qualitative reasoning that have appeared in previous work[18]. These issues are outside the scope of this chapter which focuses on defining the basic concepts and implications of treating directions as objects.

2.1.3 Scope and Outline

The organization of this chapter is as follows: In section 2 , we define a mathematical framework and propose new ADTs for directions and orientations. A new spatial data type hierarchy is also proposed in this section. In section 3, an oriented spatial object is defined using the orientation object, and its application in spatial queries is discussed. Open geometry and its applications are illustrated in section4. Finally, the chapter concludes with discussions and recommendations for future work.

2.2 Basic Concepts

2.2.1 Points, Vectors, Angles

The basic concepts we use here are points, vectors and angles, as illustrated in figure 2.1. A point has a position in space, which can be described using coordinates in a

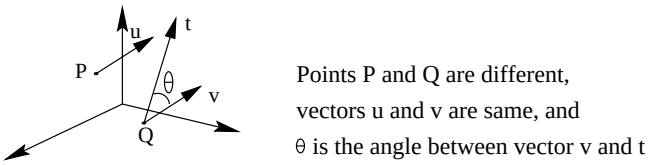


Figure 2.1: Diagram of points, vectors and angles

coordinate system. The only characteristic that distinguishes one point from another is its position. A vector, in contrast, has both magnitude and direction but no fixed position in space. An angle between two vectors represents the direction deviation from one vector to the other. Here, by angle, we mean the smaller one between two vectors, like θ in diagram 2.1. In this chapter, we denote points by capitalized letters, such as P, and denote vectors as lower case letters with an arrow above, such as $\vec{v}$.

For simplicity we will use the Cartesian coordinate system to represent points and vectors in this chapter. The Cartesian coordinate system in 3D is a right-handed rectangular coordinate system, with three axes x, y and z perpendicular to each other and intersecting at the origin. A point P in space is represented as an ordered triple (x, y, z), where x, y and z are rectangular components of P on the x, y and z axes , respectively. Let $\vec{U_x}, \vec{U_y}$ and $\vec{U_z}$ represent three unit vectors which are the basis vectors of the coordinate system. A vector $\vec{v}$ can be defined as a linear combination of these basis vectors in the form of: $\vec{v} = a\vec{U_x} + b\vec{U_y} + c\vec{U_z}$, where a, b and c are some numbers.

Table 2.1 summarizes representative operations on points and vectors. Here $P(x_p, y_p, z_p)$ and $Q(x_q, y_q, z_q)$ are two points, θ is the angle between vectors $\vec{a}$ and $\vec{b}$, where $\vec{a} = x_a\vec{U_x} + y_a\vec{U_y} + z_a\vec{U_z}$ and $\vec{b} = x_b\vec{U_x} + y_b\vec{U_y} + z_b\vec{U_z}$. We can see that the subtraction operation(-) is applied to point pairs, resulting in a vector. The operations of addition(+), subtraction(-), scale multiplication, dot product($\odot$), and cross product($\times$) are

Operands	Operations	Definition
Point(s)	-	$\vec{pq} = Q - P = (x_q - x_p)\vec{U_x} + (y_q - y_p)\vec{U_y} + (z_q - z_p)\vec{U_z}$
Vector(s)	+	$\vec{a} + \vec{b} = (x_a + x_b)\vec{U_x} + (y_a + y_b)\vec{U_y} + (z_a + z_b)\vec{U_z}$
	$\odot$	$\vec{a} \odot \vec{b} = \|a\|\|b\| \ cos\theta = x_a x_b + y_a y_b + z_a z_b$
	$\times$	$\vec{a} \times \vec{b} = \|a\|\|b\| \ sin(\theta) \ \vec{u}$, where $\vec{u}$ is a unit vector, perpendicular to $\vec{a}$ and $\vec{b}$
	scale	$m\vec{a} = mx_a\vec{U_x} + my_a\vec{U_y} + mz_a\vec{U_z}$, where m is a number
Point, Vector	+	$R = P + \vec{a} = R(x_p + x_a, y_p + y_a, z_p + z_a)$, a point away from P with distance $\|\vec{a}\|$ along the direction of $\vec{a}$

Table 2.1: Operations on vectors and points

available between vector operands. The definitions are given in terms of the components in the Cartesian coordinate system. There is only one operation(+) available between a point and a vector which actually produces another point. The associativity, commutativity and distribution properties of these operators will be examined in the future work towards integrating vector objects in query languages.

2.2.2 Direction and Orientation

Direction is defined as a unit vector, i.e., a vector with magnitude equal to 1. Table 2.2 defines the operations on directions.

Operations	Definition
composition	$\vec{d1} + \vec{d2} = \frac{\vec{d1}+\vec{d2}}{\|\vec{d1}+\vec{d2}\|}$
deviation	$cos\theta = \vec{d_1} \odot \vec{d_2}$
reverse	$(-1) \times \vec{d1}$
between[1]	$\vec{d}$ between $\vec{d_1}$ and $\vec{d_2}$ if $\exists c_1 > 0, c_2 > 0$ s.t. $\vec{d} = c_1\vec{d_1} + c_2\vec{d_2}$
among[2]	$\vec{d}$ among $\vec{d_1}$, $\vec{d_2}$ and $\vec{d_3}$ if $\exists c_1 > 0, c_2 > 0, c_3 > 0$ s.t. $\vec{d} = c_1\vec{d_1} + c_2\vec{d_2} + c_3\vec{d_3}$

Table 2.2: Operations on Directions

The operations on directions can be classified into three categories. The first category contains the operations that produce new directions. *Composition* and *reverse* are opera-

[1] This definition works well as long as vector $\vec{d_1}$ is not parallel to vector $\vec{d_2}$. The parallel case can be handled in a user defined manner.

[2] Here $\vec{d_1}$, $\vec{d_2}$, and $\vec{d_3}$ are not in the same plane.

tions in this category. The *composition* operation is actually achieved by *vector-addition*, and the resulting vector is scaled down by its magnitude to be a unit vector, which represents the new direction. The *reverse* operator produces the reverse direction vector. The second category of the operations is to calculate the deviation between two directions. Operator *deviation* calculates the cosine of the angle between two directions, and hence gives the deviation of one direction from the other. A pair of vectors are orthogonal if their dot-product returns zero, i.e., they have 90^0 deviation. The last category of the operations is to test the relationships among directions. The operators *between* and *among* belong to this category. In figure 2.2, $\vec{d}$ is between $\vec{d_1}$ and $\vec{d_2}$; however, $\vec{d_1}$ is not between $\vec{d}$ and $\vec{d_2}$. As we will see in later sections, these three categories of direction

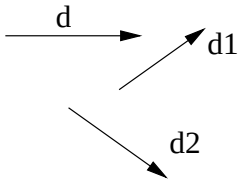


Figure 2.2: between operator

operations make the modeling of direction concise and flexible.

Orientation is modeled as a spatial object which consists of a point of origin and N pair-wise orthogonal directions, where N is the dimension of the embedding space. The origin point and the N directions form a Cartesian coordinate system. In 3D space, the three directions may be labeled the Back-Front, Left-Right, and Below-Above directions of the orientation. Formally, we can define orientation and its operations in 3D space as follows:

Orientation is a quadruple O= $\langle$OP, $\vec{front}$, $\vec{right}$, $\vec{above}\rangle$, where OP is a point, and $\vec{front}$, $\vec{right}$, $\vec{above}$ are three orthogonal directions. It has two operations:

- $translate(O, \vec{v}) = Orientation\ O' = \langle translate(OP, \vec{v}), \vec{front}, \vec{right}, \vec{above}\rangle$;
- $rotate(O, rotationMatrix) = Orientation\ O' = \langle OP, \vec{front}_n, \vec{right}_n, \vec{above}_n\rangle$, where $(\vec{front}_n, \vec{right}_n, \vec{above}_n) = (\vec{front}, \vec{right}, \vec{above}) \odot rotationMatrix$;

An example of the rotation Matrix which rotates the orientation along the $\vec{above}$

axis for an angle θ is[55]:

$$R^{\theta}_{\vec{above}} = \begin{bmatrix} cos(\theta) & -sin(\theta) & 0 \\ sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Table 2.3 gives an illustration of these operations.

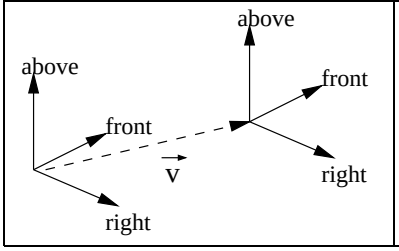
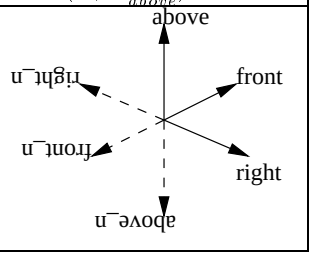
translate(O, $\vec{v}$)	rotate(O, $R^{180^0}_{\vec{above}}$)
	

Table 2.3: Operations on orientation

Abstract Data Types for Modeling Direction

As a summary of the above discussion, the definition of ADTs for vector, direction and orientation is given in Table 2.4. The C++ like syntax is used here. The column labeled

ADT	attributes	representative operations

vector	x-comp: float;
	y-comp: float;
	z-comp: float;
		vector(float, float, float); // constructor
		float magnitude();
		vector operator + (vector);
		vector operator scale();
		float operator dot-product (vector);
		vector operator × (vector);

Direction
subclass of vector
	inherited from vector
	constraint = unit magnitude
		direction(float, float, float); //constructor
		direction(vector); // constructor
		direction operator Composition(direction);
		direction operator reverse();
		float operator deviation(direction);
		boolean between(direction, direction);
		boolean among(direction, direction, direction);

Orientation	OP : *point*;
	$\vec{front}$: *direction*;
	$\vec{right}$: *direction*;
	$\vec{above}$: *direction*;
		orientation(float, direction, direction,direction);
		// constructor
		orientation operator translate(vector);
		orientation operator rotate(rotate-matrix);

Table 2.4: Abstract Data Types for direction and orientation

attributes declares the member variables of each class, and the *representative operations*

column declares the interfaces of operations for each class. The vector class has three member variables of the real type. These variables represent the three coefficients when the vector is written as the linear combination of basis vectors of the coordinate system. The constructor constructs a vector object, given three real arguments. The five vector operators are declared as member operator functions. Direction is defined as a subclass of the vector class. It can be constructed either from a vector or from an ordered triple. Besides inheriting attributes and operators from the vector class, direction also adds new operators such as *between* and *among*. The orientation class has four member variables which form a Cartesian coordinate system. *Translate* and *rotate* are declared as the member operator functions of the orientation class.

2.2.3 Frame of Reference: Absolute, Object/Viewer-orientation-based Directions

In this section, we will deal with point-based objects for simplicity. There are three different perspective systems. We define absolute Directions(e.g. north, south), object-orientation-based directions(e.g. left, above), and viewer-orientation-based directions.

Absolute Direction

Absolute direction in embedding space is defined as a relationship among objects based on their locations in embedding space. For simplicity here, we will ignore the elevation and earth's curvature and use a projection-based model to map the space to a 2D local coordinate system with north up and east right[1].

We model absolute directions in two ways: constant direction objects and directional predicates. Constant direction objects are created for frequently asked directions, e.g: $\vec{east}$, $\vec{west}$, $\vec{north}$, $\vec{south}$, $\vec{NW}$, $\vec{NE}$,$\vec{SE}$,$\vec{SW}$. Table 2.5 illustrates how we can define the constant directions in terms of coordinates in local embedding space. The unit vector is denoted by an ordered pair(a, b) which represents vector $aU_{\vec{east}} + bU_{\vec{north}}$.

unit vector	(1,0)	(0,1)	(-1,0)	(0,-1)	$(\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}})$	$(-\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}})$	$(-\frac{1}{\sqrt{2}}, -\frac{1}{\sqrt{2}})$	$(\frac{1}{\sqrt{2}}, -\frac{1}{\sqrt{2}})$
directions	$\vec{east}$	$\vec{north}$	$\vec{west}$	$\vec{south}$	$\vec{NE}$	$\vec{NW}$	$\vec{SW}$	$\vec{SE}$

Table 2.5: constant absolute direction

Introducing constant direction objects provides much flexibility in describing and deciding any direction. By using the operator *deviation*, we can calculate how much deviation is from one direction to another direction. We can also produce a new direction, given the deviation from a specific constant direction. This is very useful in the layout of facilities.

Second, we also provide corresponding directional predicates to make it convenient for users to specify the direction relation between two objects. They are *East*, *West*,

[1]For a global view, we can extend to the ellipsoidal coordinate system[33]

South, *North*, *Northwest*, *NorthEast*, *SouthWest*, *SouthEast*. For any two objects, assuming their centroids are P_1, P_2, we represent the relationship "P_1 *is to the east of* P_2" by predicate $East(P_1, P_2)$. The predicate is *true* if and only if equation $\frac{\vec{east} \odot \vec{P_2P_1}}{|\vec{P_2P_1}|} = 1$ holds. Other predicates can be defined similarly.

The above constant directions and predicates give precise directional calculation, that is, only one angle matches each case. Sometimes users may be interested only in approximate direction, such as a direction range: *between north and northwest*. The operator *between* makes this approximate directional predicate possible. If we want to test whether P_1 is *between north and northwest* of P_2, we can use the *between* operator in C++ notation as: $\vec{P_2P_1}.between(\vec{north}, \vec{NW})$.

Object-orientation-based direction

Object-orientation-based direction is the direction of the target object with respect to the orientation of the reference object. The reference object is an oriented object, while the target object may or may not have orientation. In figure 2.3, the person and the desk are oriented objects; the person is behind the desk and the desk is behind the person also. We will use O_B to represent the orientation of object B, and $O_B.\vec{front}$, $O_B.\vec{right}$ and $O_B.\vec{above}$ are the three directions of B's orientation. By using the direction operator *reverse* , the directions of B's behind, left and below can be described as $-O_B.\vec{front}$, $-O_B.\vec{right}$ and $-O_B.\vec{above}$, respectively.

Given a target point object A, and a reference point object B, the direction of A relative to B can be calculated by using the deviation operation of the direction objects. First, a direction object $\vec{BA}$ is constructed from the vector $A - B$. Then the direction of point object A with respect to point object B's orientation is decided by the deviation of direction object $\vec{BA}$ from the three directions $O_B.\vec{front}$, $O_B.\vec{right}$ and $O_B.\vec{above}$. The vector dot-product is the meta operation for direction deviation operation.

Table 2.6 illustrates the calculation of the object-orientation-based direction. The first column consists of the directional predicates and the remaining three columns are the conditions needed to satisfy the predicates. The predicate returns *true* if and only if all

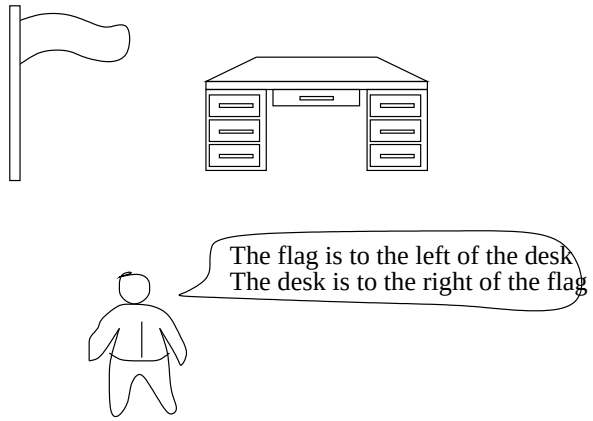


Figure 2.3: Viewer/Object-orientation-based direction

three corresponding conditions hold. The first six rows illustrate the calculation of exact directions such as $front$, $left$, *above*, etc. As in row 1, if the dot-product of vector $\vec{BA}$ and direction $O_B.\vec{front}$ is greater than zero, and the dot-products between vector $\vec{BA}$ and the other two directions of orientation O_B are both zeros , then $front(A, B) = true$, which means that A is in front of B. These directional predicates are for precise direction checking, whereas the other two categories of predicates are for direction range checking. The second category of directional predicates is to test if A is located in the region between two directions. Here the operator *between* is used in C++ notation. For example, the relationship that *A is between front(A) and right(B)* is represented as predicate $\vec{BA}.between(O_B.\vec{front}, O_B.\vec{right})$, which has value *true* if and only if $\vec{BA} \odot O_B.\vec{front}$ and $\vec{BA} \odot O_B.\vec{right}$ are both greater than zero and $\vec{BA} \odot O_B.\vec{above}$ is equal to zero. The predicates in this category are for the directions in 2D space. The third category of predicates consists of eight directional predicates which test if A is among three directions in a 3D space. We represent the predicates using the operator *among*. As can be seen from the table, by using direction object $\vec{BA}$, the checking of directional predicates converts to the calculation of direction deviation which is easily performed by the dot-product operation.

Direction predicates	$\vec{BA} \odot O_B.\vec{front}$	$\vec{BA} \odot O_B.\vec{right}$	$\vec{BA} \odot O_B.\vec{above}$
A in front of B = $front(A, B)$	> 0	0	0
A behind of B = $behind(A, B)$	< 0	0	0
A right to B = $right(A, B)$	0	> 0	0
A left to B = $left(A, B)$	0	< 0	0
A above B = $above(A, B)$	0	0	> 0
A below B = $below(A, B)$	0	0	< 0
A between front(B) and right(B) = $\vec{BA}.between(O_B.\vec{front}, O_B.\vec{right})$	> 0	> 0	0
$\vec{BA}.between(O_B.\vec{front}, -O_B.\vec{right})$	> 0	< 0	0
$\vec{BA}.between(-O_B.\vec{front}, -O_B.\vec{right})$	< 0	< 0	0
$\vec{BA}.between(-O_B.\vec{front}, O_B.\vec{right})$	< 0	> 0	0
$\vec{BA}.between(O_B.\vec{front}, O_B.\vec{above})$	> 0	0	> 0
$\vec{BA}.between(O_B.\vec{front}, -O_B.\vec{above})$	> 0	0	< 0
$\vec{BA}.between(-O_B.\vec{front}, O_B.\vec{above})$	< 0	0	> 0
$\vec{BA}.between(-O_B.\vec{front}, -O_B.\vec{above})$	< 0	0	< 0
$\vec{BA}.between(-O_B.\vec{right}, O_B.\vec{above})$	0	< 0	> 0
$\vec{BA}.between(-O_B.\vec{right}, -O_B.\vec{above})$	0	< 0	< 0
$\vec{BA}.between(O_B.\vec{right}, O_B.\vec{above})$	0	> 0	> 0
$\vec{BA}.between(O_B.\vec{right}, -O_B.\vec{above})$	0	> 0	< 0
A is among front(B),above(B) and left(B)= $\vec{BA}.among(O_B.\vec{front}, O_B.\vec{above}, -O_B.\vec{right})$	> 0	< 0	> 0
$\vec{BA}.among(O_B.\vec{front}, O_B.\vec{above}, O_B.\vec{right})$	> 0	> 0	> 0
$\vec{BA}.among(O_B.\vec{front}, -O_B.\vec{above}, O_B.\vec{right})$	> 0	> 0	< 0
$\vec{BA}.among(O_B.\vec{front}, -O_B.\vec{above}, -O_B.\vec{right})$	> 0	< 0	< 0
$\vec{BA}.among(-O_B.\vec{front}, O_B.\vec{above}, O_B.\vec{right})$	< 0	> 0	> 0
$\vec{BA}.among(-O_B.\vec{front}, O_B.\vec{above}, -O_B.\vec{right})$	< 0	< 0	< 0
$\vec{BA}.among(-O_B.\vec{front}, -O_B.\vec{above}, O_B.\vec{right})$	< 0	> 0	< 0
$\vec{BA}.among(-O_B.\vec{front}, -O_B.\vec{above}, -O_B.\vec{right})$	< 0	< 0	< 0

Table 2.6: Object-orientation-based directions

Viewer-orientation-based direction

Viewer-orientation-based direction refers to the directional relationship which is measured from the viewer's perspective. In figure 2.3, the flag is to the left of the desk from the viewer's perspective.

There are three related components in this system: target object A, reference object B, and the viewer. The viewer has his/her own orientation, whereas objects A and B may or may not be oriented objects. Given these three components, we can estimate the direction of object A relative to object B from the viewer's perspective, in a way similar to that used with the object-orientation-based system. The similar predicates but w.r.t.

the viewer are defined and are calculated by the dot-product between vector $\vec{BA}$ and the three directions of the reference orientation. The reference orientation here is the viewer's orientation O_V, and hence the three dot-products needed to be calculated are $\vec{BA} \odot O_V.\vec{front}$, $\vec{BA} \odot O_V.\vec{right}$, and $\vec{BA} \odot O_V.\vec{above}$. A similar table can be obtained as in Table 2.7.

Direction predicates	$\vec{BA} \odot O_V.\vec{front}$	$\vec{BA} \odot O_V.\vec{right}$	$\vec{BA} \odot O_V.\vec{above}$
$front(A, B)$	> 0	0	0
$behind(A, B)$	< 0	0	0
$right(A, B)$	0	> 0	0
$left(A, B)$	0	< 0	0
$above(A, B)$	0	0	> 0
$below(A, B)$	0	0	< 0
$\vec{BA}.between(O_V.\vec{front}, O_V.\vec{right})$	> 0	> 0	0
$\vec{BA}.among(O_V.\vec{front}, O_V.\vec{above}, O_V.\vec{right})$	> 0	> 0	> 0

Table 2.7: Viewer-orientation-based orientation

In the viewer–orientation-based direction system, the viewer may change his/her orientation or position(unlike objects, which are still). Using the operators *rotate* and *translate* on the orientation, we can easily generate new orientations.

Viewer-orientation-based directions are heavily used in route navigation. The route instructions can be given in terms of absolute direction, i.e., north, south, etc.. But

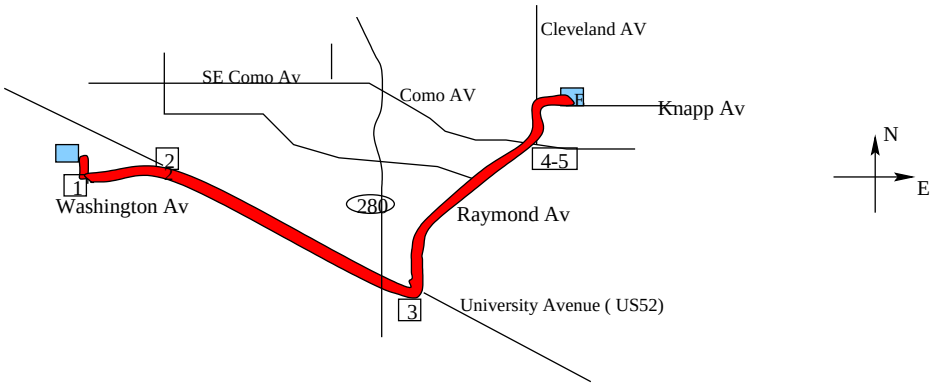


Figure 2.4: Route Planning

in fact, since many people have no idea of the absolute direction, particularly in an area new to them, route instructions in absolute direction have little meaning. The

directions would be much easier to understand if the instructions were given by viewer–orientation-based direction. Many web routing servers such as MapsOnUs use as many viewer-orientation-based directions as possible in their route instructions. Figure 2.4 is an example of a route map that starts from the Computer Science Building of the University of Minnesota to a UM apartment on the St. Paul campus. The turn-by-turn instructions are given in table 2.8. As can be seen from the table, viewer-orientation-based directions are used at every turn to make it easy for users to follow the descriptions intuitively.

	Go	And Then	Total miles
Start		Head South on Union ST SE, From Start point(200 Union ST SE, Mpls, MN)	0.0
1	Less than .1 mi	Turn LEFT onto Washington Ave. SE	0.0
2	0.5 mi	Continue onto US52 E	0.5
3	1.4 mi	Turn LEFT onto RAYMOND Ave.	1.9
4	1.1 mi	Continue onto N Cleveland Ave	3.0
5	0.2 mi	Turn RIGHT onto Knapp street	3.2
End	Less that .1 mi	End Point(2051 Knapp Ave, St.Paul, MN)	3.2

Table 2.8: Turn-by-turn Directions

2.2.4 New Spatial Data Type Hierarchy

Some work has been done on modeling space into a spatial geometry hierarchy. The Open GIS Consortium[32] proposed a spatial object hierarchy for incorporating 2D spatial ADTs in SQL, as illustrated in figure 2.5.

This hierarchy consists of *Points*, *Curves*, *Surface* and the class *GeometryCollection*. The basic operations are confined to topological and metric operations. Since these data types model only bounded shape objects, the hierarchy has limitations in modeling objects; for example, maps, rivers, roads and buildings which have their own directions/orientations. For example, with few exceptions, printed maps are all oriented with north up. This is not only for better visualization, but for more flexibility with spatial queries. It allows queries such as "Which state is to the *left* of Minnesota?" to be asked by non-geographers who may not use geographic concepts like north, west, etc.

Spatial queries such as " Is Minnesota in the north part of the USA?" can be answered

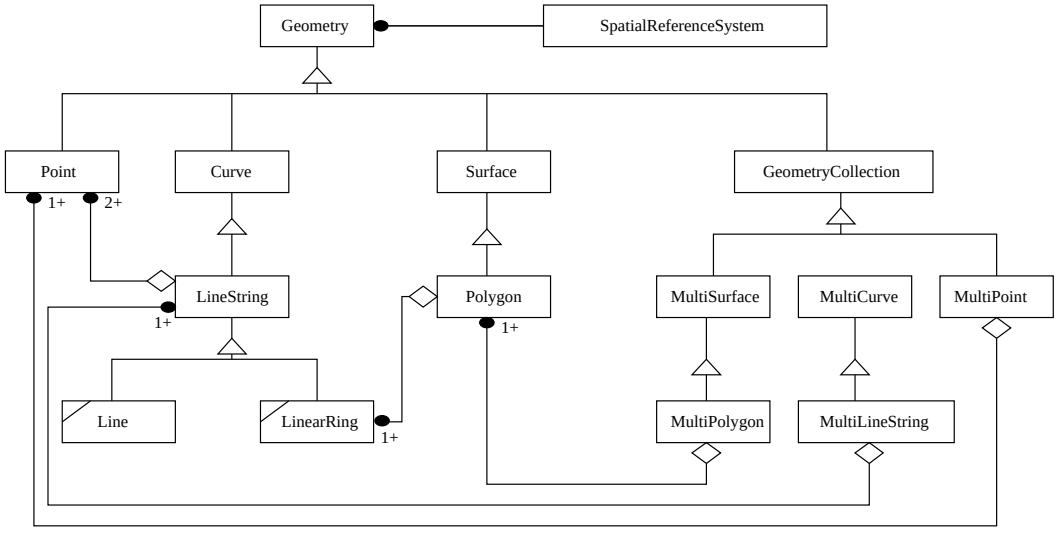


Figure 2.5: Spatial Data Type Hierarchy[12]

by comparing the location of Minnesota with the center of the USA and its direction to the center, without the necessity of getting the information on the boundary of the USA. But in the OGIS hierarchy, spatial objects can only be modeled as bounded objects, so *north part of USA* should be represented in a polygon which needs accurate boundary information that actually has little effect on the query.

A new spatial data type hierarchy is proposed in figure 2.6. This hierarchy includes oriented objects and unbounded(open) objects, which could be defined using the direction and orientation objects discussed above. In the new spatial hierarchy, maps and roads can be defined as oriented objects.

New Spatial Data Type Hierarchy

Figure 2.6 illustrates the extended spatial data type hierarchy which consists of shape objects, vectors and oriented(directed) objects. The spatial data types and their operators can be embedded in an extended query language to implement a spatial query language.

Many new operators are available in the extended spatial object model. Between

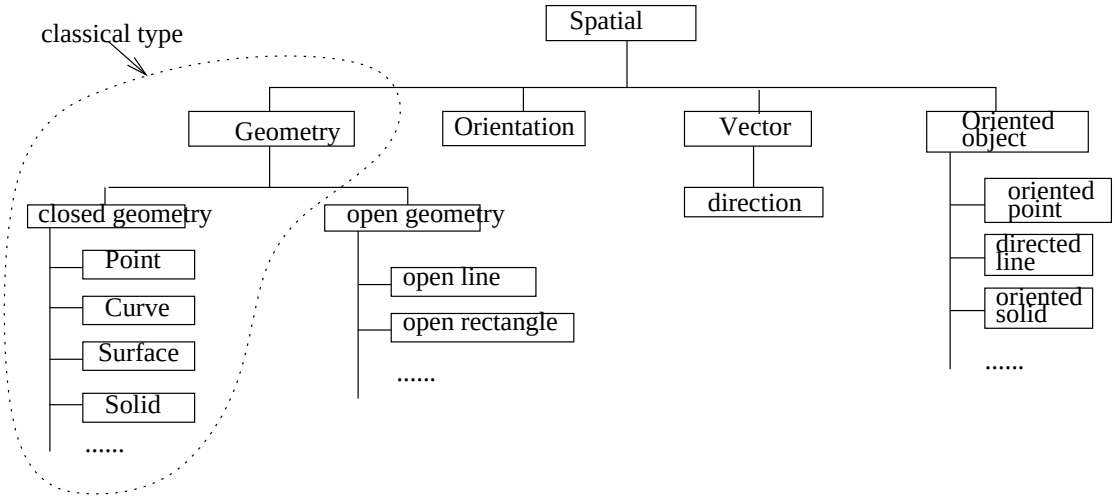


Figure 2.6: Extended Spatial Data Type Hierarchy

vector objects and shape objects, affine transformations can be performed. The addition of these operations makes some GIS applications easy. For example, Urban planning needs interactive placement/orientation of facilities, location-based query exploration can use affine operations, and using these operations can also improve viewer flexibility, so that we do not have to look down on earth from space. Instead, a viewer can view from any direction by performing rotation transformations.

In the later sections, the new spatial data types are discussed in detail. Their applications in geographic space are also described.

2.3 Defining Oriented(directed) Spatial Objects

There are many objects in geographic space that have intrinsic directions/orientations in addition to their locations and shapes, such as buildings, rivers and roads. As we discussed above, modeling maps as oriented objects provides users more flexibility in query specification. It also allows users who have little knowledge of geographic space to query directions intuitively, based on their own orientations. As in the routing example given in figure 2.4, the turn-by-turn directions in table 2.8 mostly use viewer-orientation-based direction to make the information more meaningful to non-geographical users. But since there is no information on the orientation of the starting address, the first directional instruction is based on absolute direction SOUTH. If we know the orientation of the exit of the building, then we can use viewer-orientation-based direction instead, since in this case the viewer's orientation is aligned with the building exit's orientation. The instruction then may be "Turn Right onto Union Street SE", which is very easy for any user to follow.

Table 2.9 shows some queries that can only be answered when the reference objects and/or viewers are modeled as oriented objects.

Queries
List the swamps in front of the tank.
List the farm fields suitable for tank movement, and are left to Lake A as viewed by Viewer 1.
Let's move back a little.
How far is the next intersection down this road?

Table 2.9: Queries Need Orientation Information

Introducing the *Orientation* class into the spatial object hierarchy makes it easy to formalize oriented objects. In addition to the attributes such as *location* and *shape* for the geometric shape objects, an oriented object has an attribute *orientation*, which is an instance object of class orientation. The operators available for oriented objects include both topological operators and direction and orientation operators.

The ADT for oriented objects can be specified in Oracle8.0[74]. We first define the direction object type and orientation object in Oracle 8.0. In the following, we assume

Geometry is the spatial type defined by OGIS[32]. It could be a point, a line, a polygon, and etc.

```
CREATE   TYPE DirectionType AS OBJECT (
         x NUMBER,
         y NUMBER,
         z NUMBER,
         MEMBER FUNCTION composition(aDir IN DirectionType) RETURN DirectionType,
         MEMBER FUNCTION reverse RETURN DirectionType,
         MEMBER FUNCTION deviation(aDir IN DirectionType) RETURN NUMBER,
         MEMBER FUNCTION between(aDir IN DirectionType, bDir IN DirectionType)
            RETURN NUMBER,
         PRAGMA RESTRICT_REFERENCES(default, WNDS)
);

CREATE   TYPE OrientationType AS OBJECT (
         op PointType,
         d1 DirectionType,
         d2 DirectionType,
         d3 DirectionType,
         MEMBER FUNCTION translate(aVector IN VectorType) RETURN OrientationType,
         MEMBER FUNCTION rotate(degree IN NUMBER) RETURN OrientationType,
         PRAGMA RESTRICT_REFERENCES(default, WNDS)
);

CREATE   TYPE OrientedObject AS OBJECT (
         shape Geometry,
         orientation OrientationType,
         MEMBER FUNCTION left(obj IN Geometry ) RETURN NUMBER,
         /* other functions can be defined similarly */
         PRAGMA RESTRICT_REFERENCES(default, WNDS)
);

CREATE   TYPE ViewerType AS OBJECT (
         name VARCHAR2(20),
         orientation OrientationType,
         MEMBER FUNCTION left(obj1 IN Geometry, obj2 IN Geometry) RETURN NUMBER,
         /* other functions can be defined similarly */
         PRAGMA RESTRICT_REFERENCES(default, WNDS)
);
```

We can use these data types to solve spatial queries involving oriented objects. As an example, we will show how to write the first two queries in table 2.9 using these data types. The first query contains object-orientation-based direction and the second one focuses on viewer-orientation-based direction. We first create a database scheme consisting of four tables, $FarmField$, $LandCover$, $Tank$, and $Viewer$. The queries are performed on this database schema.

```
CREATE TABLE FarmField(name VARCHAR2(20), type VARCHAR2(20), extent Geometry);

CREATE TABLE LandCover (name VARCHAR2(20), type VARCHAR2(20), extent Geometry);

CREATE TABLE Tank (tankid VARCHAR2(20), time Time, extent OrientedObject);
```

```
CREATE TABLE Viewer OF ViewerType;
```

Query 1 : List the swamps in front of the tank.

```
SELECT  L.name, L.extent
FROM    Landcover L, Tank T
WHERE   T.extent.front(L.extent) = 1 AND
        T.tankid = 'TANKID' AND L.type = 'swamps'
```

- Explanation:$front$ is an object-orientation-based direction predicate with respect to the oriented object Tank. Landcover is a partition of the space based on landcover type. We search for partitions which lie in front of the tank and check their type. Example of possible types include "swamps", "water", "hill".

Query 2 : List the farm fields suitable for tank movement, and are left to Lake A as viewed by Viewer1.

```
SELECT  F.name, F.extent
FROM    FarmField F, Landcover L, Viewer V
WHERE   V.left(F.extent, L.extent)=1 AND
        L.name = 'Lake A' AND V.name = 'Viewer1' AND F.type = 1
```

- Explanation: A farm field is suitable for tank movement if the crop growing inside is soft and low. The *type* field in the $FarmField$ table identify this property. $left$ is a viewer-orientation-based direction predicate with respect to the orientation of a viewer. The database will search for FarmFields that are left of the lake from Viewer1's perspective, and list all the satisfying fields suitable for tank movement.

2.4 Open Geometry

Open objects mean those geometries whose boundaries are partially defined, or extending beyond the data window, or infinite. To think of a query example first:

Query: Given the map of Minnesota as our data window, what's the length of the Mississippi river?

Questions: What should the GIS system answer?

(a) The length of the Mississippi river inside Minnesota.

(b) At least the length of the Mississippi river inside Minnesota, but I don't know exactly.

Since the Mississippi river extends outside Minnesota, we cannot compute the exact length of the river given only the Minnesota map, so we prefer answer (b). Modeling the river as a closed object can only return answer (a) which is not precise; however, if we model the river as an open object, then answer (b) will be obtained, which is more accurate.

2.4.1 Categories of Open Geometry

Table 2.10 categorizes the open objects based on two factors: the extent of the object and our knowledge of the object. Figure 2.7 gives examples of each category of open objects.

Open Object	Knowledge		Extent
	within window	beyond window	
Open-G	complete	none	finite
Open-I	complete	complete	infinite
Open-P	partial	partial	

Table 2.10: Open Shapes

If the object is finite and its boundary is well defined within the data window, then this is a closed object. Type Open-G refers to the object that has a finite boundary but extends beyond the data window; and the portion of the object within the data window

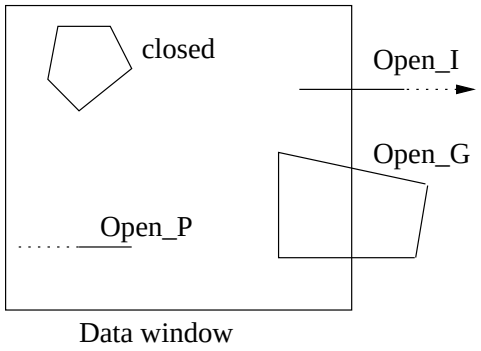


Figure 2.7: Examples of open shapes

is fully known, but the portion outside the data window is only partially known. This type of open object is useful in GIS applications. For example, if we consider the map of Minnesota as our data window, then the Mississippi river falls into the category of Open-G. Open-I refers to the object that has an infinite boundary, but we have complete knowledge of the object either within the data window or outside. Even though it seems of little interest to Geographic science since the earth has a finite surface, the category is useful in modeling direction for extended objects, as we will discuss in a later section. Moreover, Open-I and Open-G have similar topological properties. Type Open-P refers to an object of which we have only incomplete information. Dealing with this kind of object is out of the scope of our discussion. In this chapter, we will focus on the category of Open-G and Open-I, and will use the term *Open object* in the places where difference between the two is unnecessary. to be differentiated.

2.4.2 Geometric and Topological Operations on Open Objects

In this subsection, we will examine the effects of open objects on geometric and topological operations. Table 2.11 and table 2.12 give the results for basic geometric and topological operations, respectively. Here "Closed" represents a closed object and "Open" refers to an open object. The first column of each small table contains the first argument type, and the first row contains the second argument type. For example, $Difference$(Closed, Open) = Closed, and $Difference$(Open, Closed) = Open. If the two arguments of $difference$ are Open objects, the result then could be either a closed object(If the two

open objects extending to the same space) or an open object.

Intersection	Closed	Open
Closed	Closed	Closed
Open	Closed	Open/ Closed

Union	Closed	Open
Closed	closed	open
Open	Open	open

Difference	Closed	Open
Closed	Closed	Closed
Open	open	Open/ Closed

Table 2.11: Domains and Ranges for Geometric Operations on Open Objects

ETDOC	Closed	Open
Closed	boolean	boolean
Open	boolean	boolean/ don't know

Within	Closed	Open
Closed	boolean	boolean
Open	False	boolean/ don't know

Table 2.12: Domain and Ranges for Logical Operations on Open Object. **ETDOC** stands for one of $\{Equals, Touches, Disjoint, Overlap, Crosses\}$

Figure 2.8 illustrates the four cases of operation $Difference$ with an open shape involved. The pattern parts represent the difference between two objects. It is worth to mentioning here that since the open shapes E and F extend to the same region, so $Difference(E, F)$ is a closed shape.

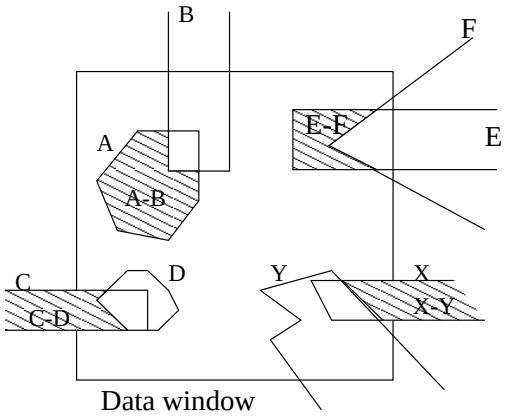


Figure 2.8: Operation $Difference$ on open shapes

In table 2.12, we combine operations that have the same results into one table. Here ETDOC represents one of the operations in the set of {*Equal, Touches, Disjoint, Overlap, Crosses*}. The result of the operations could be either *boolean* or *don't know*. *don't know* is a special value which has properties similar to the null value in a database. A useful

observation is that topological operations involving closed objects always have boolean results.

2.4.3 Defining Open lines, Open Rectangles

Examples of open spatial objects are open lines, open planes, and open spaces. A common way to define these in 3D Cartesian geometry is in terms of equations such as the parametric equations [52]: $x = x_0 + at, y = y_0 + bt, z = z_0 + ct$ for a line through the point (x_0, y_0, z_0), Ax+By+C=0 for a plane, and Ax+C=0 for a 3D space. It would be more convenient if there were corresponding object classes so that users could describe them at an abstract level and use them to describe the space. The *direction* object could be used for this purpose.

We define open lines and open rectangle regions here using directions and points. Table 2.13 shows how to formalize each open object with examples in figure 2.9.

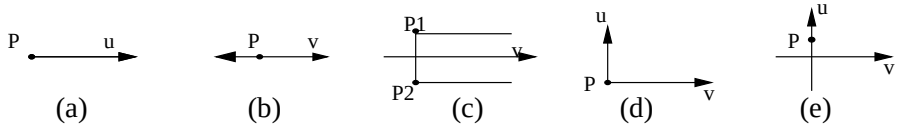


Figure 2.9: Examples of open shapes

Type	Subtype	Descriptions	Examples
open line	openline1(1-end open line)	start-point, direction	(a)
	openline2(2-end open line)	intermediate-point, direction	(b)
open rectangle	openrect1:1-side open rectangle	$vertex_1$,$vertex_2$,direction, where direction perpendicular to segment formed by the two vertices	(c)
	openrect2(2-side open region)	two 1-end open lines from the only vertex	(d)
	openrect3(3-side open rectangle)	2-end open line, direction	(e)

Table 2.13: Defining open objects

There are several subtypes in each type. For an open line, it could be 1-end open(figure 2.9a), which needs a start point and a direction to define it, or it could be 2-end open(figure 2.9b), which needs an intermediate point and a direction to describe it. The open rectangle also has three subtypes: a 1-side open rectangle (2.9c), described by

two vertices of the rectangle and one direction which is perpendicular to the edge connected by the two vertices. A 2-side open rectangle is described using two 1-end open lines which start from the vertex of the rectangle. Also a 3-side open rectangle could be defined by a 2-end open line and a direction. It is obvious that extending to an open cube is easy.

Abstract Data Types for open lines and open rectangles

The ADTs for open objects are given in table 2.14 using C++ notation. The entire open object class is defined as a subclass of geometry. Geometric and topological operations available for open objects include *openUnion, openIntersection, openDifference, openWithin, openEquals, openTouches, openDisjoint, openOverlap, and OverCrosses.* The definition of the operators should be overloaded for open objects, according to table 2.11 and table 2.12. Here, we only show the attributes and constructors of each open object class, and as an illustration, we represent the spatial objects in figure 2.9.

ADT	attributes	constructors	examples in figure 2.9
Openline1	$sPoint : Point$; $dir : Direction$;	openline1(point, direction);	(a): $openline1(P, \vec{v})$
Openline2	$iPoint : Point$; $dir : Direction$;	openline2(point, direction);	(b): $openline2(P, \vec{v})$
Openrect1	$vertex_1 : Point$; $vertex_2 : Point$; $dir : Direction$;	openrect1(point, point, direction);	(c): $openrect1(P_1, P_2, \vec{v})$

Openrect2	$line_1 : Openline1$;
	
	$line_2 : Openline1$;
		openrect2(openline1, openline1);
		
		openrect2(point, direction, direction);
			(c): $openrect2(openline1(P, \vec{u}), openline1(P, \vec{v}))$
			(c): $openrect1(P, \vec{u}, \vec{v})$;

Openrect3	$line : Openline2$; $dir : Direction$;	openrect3(openline2, direction);	(d): $openrect3(openline2(P, \vec{u}), \vec{v})$

Table 2.14: Abstract Data Types for Open Lines and Open Rectangles

2.4.4 Modeling Direction Between Non-point Objects using Open Shapes

We can use open shapes to model the directions between extended objects by converting the calculation of directional relationships to the calculation of topological relationships between objects.

Given two objects TO and RO, we want to decide the direction of target object TO related to reference object RO. Using the approach in [25], first we obtain the MBR of object RO and partition the space around object RO into nine direction tiles based on the MBR of object RO, which is a rectangle ABCD as in figure 2.10. We represent each direction tile as a spatial object, as in table 2.15.

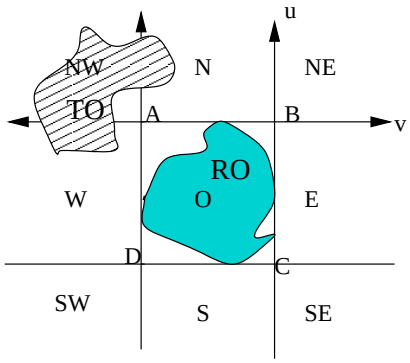


Figure 2.10: Object TO and RO

Rectangle	representation for figure 2.10
NW	$openrect2(openline2(A, \vec{u}), openline2(A, -\vec{v}))$
N	$openrect1(A, B, \vec{u})$
NE	$openrect2(openline2(B, \vec{u}), openline2(B, \vec{v}))$
E	$openrect1(B, C, \vec{v})$
SE	$openrect2(openline2(C, -\vec{u}), openline2(C, \vec{v}))$
S	$openrect1(C, D, -\vec{v})$
SW	$openrect2(openline2(D, -\vec{u}), openline2(D, -\vec{v}))$
W	$openrect1(A, D, -\vec{v})$
O	rectangle(A,B,C, D)

Table 2.15: Direction tile

Eight of the nine direction tiles are open rectangles. NW, NE, SE, and SW are 2-side open rectangles, and N, W, E, and S are 1-side open rectangles. In order to test the direction of object TO related to object RO, we can test into which direction tile TO falls. The calculation is then converted to the overlap relationship between object TO and the direction tiles. For example,

$$North(TO, RO) <==> openOverlap(TO, N)$$

$$NorthWest(TO, RO) <==> openOverlap(TO, NW)$$

The other directional relationship can be tested similarly. As we can see in table 2.12,

the topological operations involving closed objects produce boolean values. The union of all the tiles that overlap with A is the region where the target object TO is located w.r.t. the object RO. For example, in figure 2.10, TO is North and Northwest and West of object RO.

2.5 Conclusions and Future Work

In this chapter, we model direction as a new kind of spatial object using the concepts of vectors, points and angles. The basic approach here is to model direction as a unit vector. This novel object view of direction has several obvious advantages: It allows users to define the orientation of spatial objects; it also gives a richer set of predicates and operators on direction and orientation. New spatial data types such as oriented spatial objects and unbounded spatial objects can be defined as well. Finally, direction reasoning, which is useful in spatial query processing and optimization, can be implemented simply by basic vector algebra.

The benefits of the new viewpoint are obvious and promising. In future work, we would integrate vector abstract data types with an extensible query language to evaluate their efficiency in the context of various GIS applications[61, 63]. We would also like to explore the use of vectors to model values of spatial attributes such as wind-velocity, the magnetic field of the earth, etc.

Chapter 3

Processing Object-orientation-based Direction Queries

Direction based spatial relationships are critical in many domains including geographic information systems(GIS) and image interpretation. They are also frequently used as selection conditions in spatial queries. In this chapter, we explore processing of queries based on object-orientation-based directional relationships. A new Open Shape based strategy(OSS) is proposed. OSS converts the processing of the direction predicates to the processing of topological operations between open shapes and closed geometry objects. Since OSS models the direction region as an OpenShape, it does not need to know the boundary of the embedding world, and also eliminating the computation related to the world boundary. We perform algebraic analysis as well as experimental evaluation for OSS. The experimental result demonstrates that the OSS consistently outperforms classical range query strategy both in I/O and CPU cost.

3.1 Introduction

"Direction" is a common spatial concept frequently used as a selection condition in spatial databases[28, 15] or for similarity accessing in image databases[53]. Examples of direction queries used in army battlefield visualization[27] are "List the enemy targets to the *north*

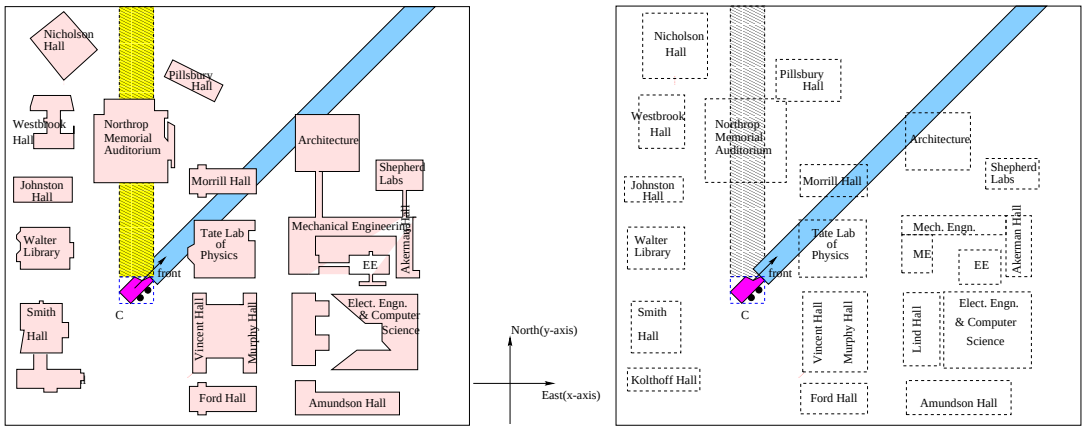
of Building B","Is there anything *over* the ridge?", and "Describe all tanks to the *left* of Landmark L". The first example refers to an absolute directional relationship(*north*) with respect to Building B, the second involves a viewer-orientation-based directional relationship, i.e. the direction is based on the orientation of the viewer, and the third example uses an object-orientation-based direction as its selection criteria.

It is important for a spatial database system to provide a mechanism for modeling and processing direction queries. Most of the previous work [71, 70, 49] on direction query processing has focused on processing predicates of absolute directions(e.g. North, East) using range query strategies. However, object-orientation-based direction queries and predicates(e.g. left) depend on the orientation of reference objects, which may be different from the global reference system. Because of this characteristic, the strategies that are efficient for absolute directions may be inefficient when used for object-orientation-based directions.

In this chapter, we focus on the processing of spatial queries involving object-orientation-based direction predicates in selection conditions. Based on direction object model proposed in our recent work [64], we define a new abstract data type(ADT) for open shapes and propose a new Open Shape based strategy(**OSS**). OSS transforms the computation of the direction predicates into the computation of topological operations between open shapes and objects. Algebraic analysis and experimental evaluation demonstrate that OSS consistently outperforms classical strategies both in I/O and CPU cost.

3.1.1 Problem Definition

Absolute direction of a target object relative to a reference object is defined by their locations in the embedding space. Usually, the absolute direction is described in context of a global 2-D coordinate system e.g. (north up, east right) as shown in figure 3.1. Some examples of absolute directional relationships are $North$, $East$, and $NorthEast$. Figure 3.1(a) shows a small part of the campus map of University of Minnesota with a car C. A query involving an absolute direction predicate is "List all buildings north of C". The dashed shaded rectangle in the figure shows the region satisfying the query



(a) Campus map of University of Minnesota (b) Approximate buildings by MBRs

Figure 3.1: Absolute direction vs. Object-orientation-based direction

selection condition $North$. This shaded rectangle is called *direction region*, which consists exactly of all the points satisfying the direction predicate. The query identifies "Northrop Memorial Auditorium" as the only building satisfying the query.

Object-orientation-based direction of a target object is described with respect to the orientation of a reference object. The orientation of the target object is not relevant. In figure 3.2, the desk is an oriented object, but the flag is not. The flag is to the right of the desk based on the orientation of the desk. In other words, imagine the reference object *desk* were a person, then object-orientation-based direction is described with respect to the person's orientation and location.

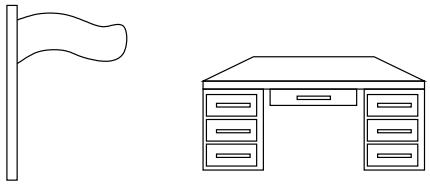


Figure 3.2: Object-orientation-based direction: The flag is to the right of the desk, assuming front of desk is facing out of the chapter

In Figure 3.1(a), the car C has an intrinsic orientation described by the direction $front$. Consider a query based on the orientation of C, i.e., object-orientation-based

direction query, "List all the buildings in front of the car C". The solid shaded region shows the direction region for "in front of the car C", and hence, the buildings in front of the car include "Tate lab", "Morrill Hall", and "Architecture".

It is common to approximate 2-D regions by minimum bounding rectangles(MBRs) which are orthogonal with respect to the global coordinate system. We refer to this type of MBR as MBR_g. Figure 3.1(b) replaces all the buildings in Figure 3.1(a) by their corresponding $MBR_g s$. However, for a reference object, the MBR may need to be taken based on the orientation of the object. We represent this type of MBR as MBR_o. Figure 3.3 shows examples of these two MBR types for a given object. The dashed orthogonal rectangle is the MBR according to the global coordinate system, and the solid rectangle is the MBR according to the orientation of the object.

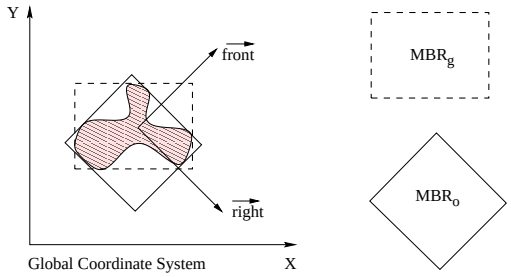


Figure 3.3: Different MBRs according to different coordinate systems

Collections of $MBR_g s$ can be organized by a R-tree for efficient search. Readers not familiar with R-trees may refer to related literature [74, 29, 22]. In the rest of the chapter, we assume data objects are approximated by MBR_g and reference objects are approximated by MBR_o. For simplicity, if not explicitly clarified, MBR represents MBR_g.

In this chapter, we intend to explore processing strategies for spatial queries which involve object-orientation-based direction predicates as selection criteria. The problem is denoted as Object-Orientation-based Direction Queries(OODQ) and can be formally defined as follows:

Object-Orientation-based Direction Queries(OODQ) problem:

Given: (a) A query involving an object-orientation-based direction predicate in selection conditions

(b) a R-tree index of the dataset

Find: All objects satisfying the selection condition

Objective: Minimize the number of page access.

Constraint: (a) Objects are approximated as MBR_g.

(b) There is only one R-tree index based on global coordinate system

3.1.2 Related work and our contributions

Direction is often modeled as a binary relationship between objects [18, 60, 25, 76, 15, 51, 26, 20, 62, 48]. Different direction predicate sets [49, 25, 31, 18] have been proposed using binary relationship view of directions. Research on processing direction-based queries has explored indexing methods and processing strategies. A previous study[70] evaluated different indices for processing absolute direction queries. Its results showed that spatial index (e.g. R-tree) is efficient for retrieving queries with constraints on both dimensions. Range query strategy(RQS) was used to implement queries involving absolute directions[71, 70, 49]. For example, consider the query based on the objects in Figure 3.4: *Find all objects that are exactly east of object B.* In the context of absolute directions, the x-axis and y-axis of the coordinate system may be aligned with the *East* and *North* directions. The direction queries based on absolute directions(B's East) can be viewed as a special case of range query as shown in figure 3.4. Here, all the objects

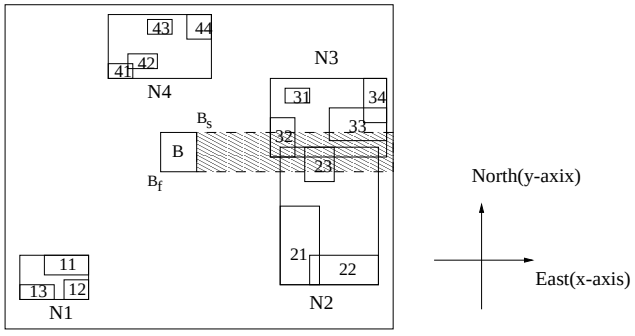


Figure 3.4: Processing absolute direction query using range query strategy

whose interiors intersect with the interior of the shaded *direction region* belong to the query result. Since an absolute direction is based on the global reference system, its direction region is a rectangle with sides parallel to the axes of the global coordinate system. It is natural and efficient to use range query strategy[70], which is specified by Algorithm 1 named RQS_AD. RQS_AD finds all object MBRs satisfying a given direction predicate using a R-tree. The input of RQS_AD is a set of data objects indexed by a R-tree and a direction region(*dirRectangle*). This algorithm starts from the root node of R-tree, excludes the intermediate nodes whose MBRs do not intersect with *dirRectangle*, and recursively searches the children of the remaining intermediate nodes. A refinement using the exact geometry of these objects may occur. But for simplicity, we will omit that in the rest of the chapter.

Algorithm 1 RQS_AD: Processing Absolute Direction Query using Range Query Strategy

```
Input: rtree is the R-tree index for the dataset;
       dirRectangle is the direction region
Output: resultSet contains all object MBRs whose interiors intersect with dirRectangle

RQS_AD(R-tree *rtree, Rectangle dirRectangle) {
    currentNode = rtree;
    if overlaps(currentNode.Mbr, dirRectangle) {
        if (currentNode is a leaf node)
            Add currentNode to resultSet;
        else
            for each childNode ∈ child node of currentNode
                RQS_AD(childNode, dirRectangle);
    }
}
```

Let us now consider extension of RQS_AD for object-orientation-based direction queries. If the orientation of the reference object is not aligned with the global coordinate system, then the corresponding direction region will not have sides parallel to the axes of the global coordinate system. One way to use range query strategy for object-orientation-based directional relationship is first to calculate MBR_g of the direction region, and then use this MBR as the range query constraint. The algorithm *RQS* shows the steps for object-orientation-based direction query processing. Since the MBR of the direction region is a coarse approximation of the actual direction region,

the intermediate result set produced by *RQS_AD* may contain objects whose MBRs do

Algorithm 2 RQS: Processing object-orientation-based direction query using range query strategy

```
Input: rtree points to the R-tree of data objects;
       referenceObject is the reference object with intrinsic orientation ;
       dirPredicate is the predicate involved in the selection condition of the query;
Output: resultSet = All object MBRs which satisfy the query

RQS(R-tree *rtree, Geometry referenceObject, Predicate dirPredicate) {
    directionRegion = computeDirRegion(referenceObject, dirPredicate);
    directionMBR = getMBR(directionRegion);
    intermediateResult = RQS_AD(rtree, directionMBR);      /* see algorithm 1*/
    resultSet = postFilter(intermediateResult);
}
```

not intersect with the interior of the direction region. A *postFilter* step is needed to eliminate false hits.

Figure 3.5 illustrates the query processing using range query strategy(RQS). There is a reference object B with orientation as shown in figure 3.5. The query to be processed is

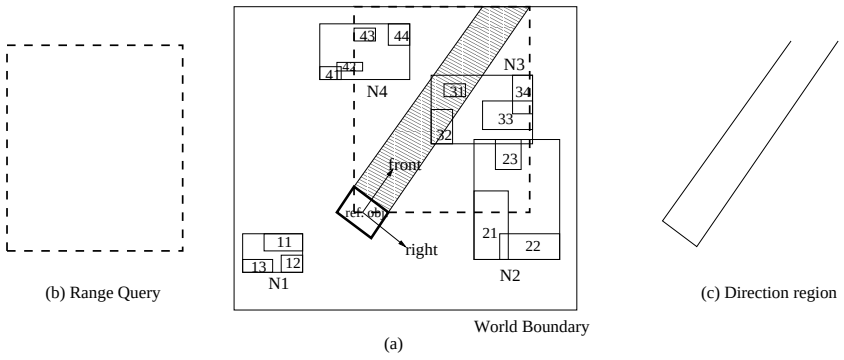


Figure 3.5: Processing object-orientation-based query using range query strategy

List all objects which are exactly in front of B. The shaded region is the direction region, and the dashed rectangle is *directionMBR* which is calculated via subroutine *getMBR*. By applying algorithm *RQS_AD*, we obtain all the objects whose MBRs intersect with the interior of directionMBR. The object selected are N21, N23, N33, N42, N43, N44 as shown in Table 3.1. A post-filtering step is needed to exclude from intermediateResult those objects whose MBRs do not intersect with the interior of directionRegion. The data MBRs selected are N31 and N32. Table 3.1 lists the trace of the query processing.

only node N1 is excluded in the first level retrieval of R-tree, although node N2 and

Steps	MBRs excluded	MBRs remain
level 1	N1	N2, N3, N4
level 2	N41, N22	N21, N23, N31, N32, N33, N42, N43, N44
postFilter	N21, N23, N33, N42, N43, N44	N31, N32

Table 3.1: Trace of the query processing

N4 are not satisfied with the direction predicate. There are eight object MBRs to be checked in the postFilter step and only two of them satisfy the query.

As seen from the above example, range query strategy can be used as a filter step to process object-orientation-based direction queries. However, in many instances the directionMBR is much larger than the direction region, which incurs unnecessary I/O and CPU costs. The intermediate result contains a large number of false hits. A postFilter step is required to check for intersection with the direction region, leading to additional CPU overhead.

In this chapter, we propose a new strategy, namely, Open Shape based Strategy(OSS), for processing object-orientation-based direction queries. The basic idea is to use the actual direction region as the filter during traversal of spatial indices to exclude potential false hits as early as possible. This is accomplished by modeling direction regions as a new ADT, namely OpenShape, defined in terms of direction objects and operators. Open shapes refer to the geometries whose boundaries are partially defined. Open shape objects conceptually extend beyond the the boundary of embedding world. Figure 3.5(c) shows the open direction region. Since OSS models the direction region as an OpenShape, it does not need to know the boundary of the embedding world, eliminating the computation related to the world boundary. OSS converts the calculation related to the direction predicates into the computation of topological operations between OpenShapes and closed geometric objects. OSS filters out irrelevant intermediate nodes during recursive search of R-tree. It eliminates the postFilter step of RQS for MBR objects. Algebraic analysis and experiment results demonstrate that the OSS outperforms RQS(range query strategy) both in terms of I/O cost and CPU cost. Relative performance of OSS in comparison to RQS is affected by several parameters including the orientation and size of the

reference object, the overlap degree of the data sets, and the type of direction predicate. On average, the best relative performance improvement is achieved when angle between $front$ axis and x-axis is close to $\frac{\pi}{4}+k\times\frac{\pi}{2}$(k is an integer). Relative performance improves with the decrease in the size of the reference objects.

3.1.3 Scope and Outline

In this chapter, we propose a new strategy to process object-orientation-based direction queries. We focus on queries that involve object-orientation-based directional relationships between simple region objects. We approximate the region objects by MBR_gs. A set of object-orientation-based direction predicates between region objects is defined, and our discussion about query processing is based on the predicates within this set. We assume that there is only one R-tree index with fixed orientation. The situation with multiple R-tree indices with different orientation is out of the scope of this thesis

The organization of this chapter is as follows: In section 2, we define a set of direction predicates for region objects. In section 3, a new strategy OSS is proposed to process the queries involving direction predicates. A new ADT for modeling direction region is defined in section 3 as well. In section 4, an algebraic analysis for OSS is performed. The experimental evaluation is discussed in section 5. The chapter ends with conclusions and recommendations for future work.

3.2 Object-Orientation-based Direction Predicates

In this section, we will give a brief review of the direction model and define an example set of direction predicates between region objects. More details on direction objects are available in [64].

3.2.1 Basic Concepts: Direction and Orientation

Direction is defined as a unit vector, i.e., a vector with magnitude equal to 1. Table 3.2 defines the operations on directions using vector algebra operations.

Operations	Definition
compose	$\vec{d1} + \vec{d2} = \frac{\vec{d1}+\vec{d2}}{\lvert\vec{d1}+\vec{d2}\rvert}$
deviation	$cos\theta = \vec{d_1} \odot \vec{d_2}$
reverse	$(-1) \times \vec{d1}$
isBetween[1]	$\vec{d}$ is between $\vec{d_1}$ and $\vec{d_2}$ if $\exists c_1 > 0, c_2 > 0$ s.t. $\vec{d} = c_1\vec{d_1} + c_2\vec{d_2}$

Table 3.2: Operations on Directions $\vec{d_1}, \vec{d_2}$

Operations on directions can be classified into three categories. The first category contains the operations that produce new directions: *compose* and *reverse* are operations in this category. The *compose* operation is actually achieved by *vector-addition*, and the resulting vector is scaled down by its magnitude to be a unit vector, which represents the new direction. The *reverse* operator produces the reverse direction vector. The second category of operations is to calculate the deviation between two directions. Operator *deviation* calculates the cosine of the angle between two directions, and hence gives the deviation of one direction from the other. A pair of vectors are orthogonal if their dot-product equals zero, i.e., they have 90^0 deviation. The last category of operations test the relationships among directions. The operator *isBetween* belongs to this category. In figure 3.6, $\vec{d}$ is between $\vec{d_1}$ and $\vec{d_2}$; however, $\vec{d_1}$ is not between $\vec{d}$ and $\vec{d_2}$.

Orientation is modeled as a spatial object which consists of a point of origin and N

[1] This definition works well as long as vector $\vec{d_1}$ is not parallel to vector $\vec{d_2}$. The parallel case can be handled in a user defined manner.

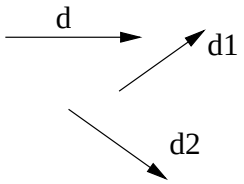


Figure 3.6: direction $\vec{d}$ is between $(\vec{d1}, \vec{d2})$

pair-wise orthogonal directions, where N is the dimension of the embedding space. The origin point and the N directions form a Cartesian Coordinate System. In 2D space, the two directions may be labeled as the $\vec{right}$ and $\vec{front}$ directions of the orientation. Formally, we can define orientation and its operations in 2D space as follows:

Orientation is a triple O= $\langle$OP, $\vec{right}$, $\vec{front}\rangle$, where OP is a point, and $\vec{right}$, $\vec{front}$ are two orthogonal directions. It has two operations:

- $translate(O, \vec{v}) = \langle translate(OP, \vec{v}), \vec{right}, \vec{front}\rangle$;
- $rotate(O, rotationMatrix) = \langle OP, \vec{right}_n, \vec{front}_n\rangle$,
  where $(\vec{right}_n, \vec{front}_n) = (\vec{right}, \vec{front}) \odot rotationMatrix$;

An example of the rotation Matrix which rotates the orientation along the origin point for an angle θ is[55]:

$$R^{\theta}_{\vec{OP}} = \begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix}$$

Table 3.3 gives an illustration of these operations.

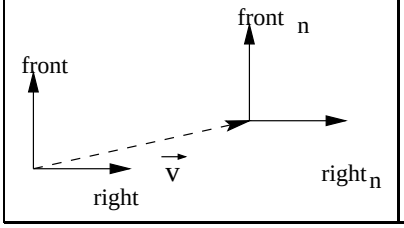
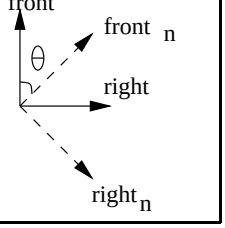
translate(O, $\vec{v}$)	rotate(O, $R^{-\theta}_{\vec{OP}}$)
	

Table 3.3: Operations on orientation

ADTs for direction and orientation objects are described in [64]. We will use the C++ dot

notation in the rest of the chapter to refer to the attributes and operations of direction and orientation objects.

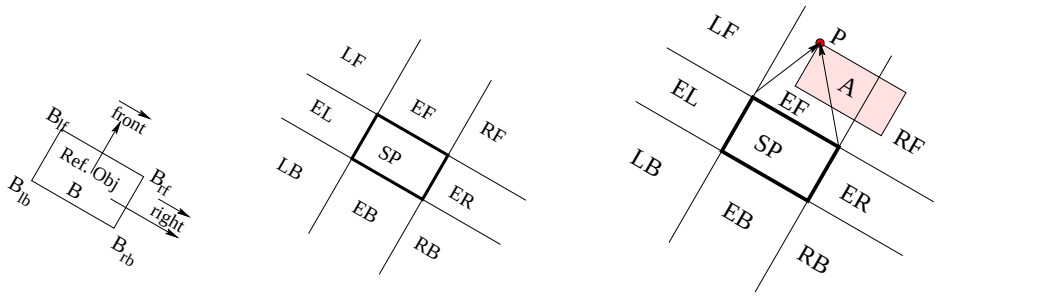
3.2.2 Direction Predicates Between Region Objects

After reviewing direction and orientation objects and their operators, we now model object-orientation-based directions between region objects. We begin by defining an example set of direction predicates. Table 3.4 gives the definition of our predicates in terms of the *isBetween* operator. Left column in Table 3.4 lists the object-orientation-based

Predicates	Definitions
$SP(A,B)$	$\exists$ point P $\in$ A, s.t. P $\in$ interior(B)
$EF(A,B)$	$\exists$ point P $\in$ A, s.t. $\vec{B_{lf}}P.isBetween(\vec{front},\vec{right}) \wedge$ $\vec{B_{rf}}P.isBetween(\vec{front},\vec{right}.reverse())$
$EB(A,B)$	$\exists$ point P $\in$ A, s.t. $\vec{B_{lb}}P.isBetween(\vec{front}.reverse(),\vec{right}) \wedge$ $\vec{B_{rb}}P.isBetween(\vec{front}.reverse(),\vec{right}.reverse())$
$ER(A,B)$	$\exists$ point P $\in$ A, s.t. $\vec{B_{rf}}P.isBetween(\vec{front}.reverse(),\vec{right}) \wedge$ $\vec{B_{rb}}P.isBetween(\vec{front},\vec{right})$
$EL(A,B)$	$\exists$ point P $\in$ A, s.t. $\vec{B_{lb}}P.isBetween(\vec{front},\vec{right}.reverse()) \wedge$ $\vec{B_{lf}}P.isBetween(\vec{front},\vec{right})$
$RF(A,B)$	$\exists$ point P $\in$ A, s.t. $\vec{B_{rf}}P.isBetween(\vec{front},\vec{right})$
$RB(A,B)$	$\exists$ point P $\in$ A, s.t. $\vec{B_{rb}}P.isBetween(\vec{front}.reverse(),\vec{right})$
$LF(A,B)$	$\exists$ point P $\in$ A, s.t. $\vec{B_{lf}}P.isBetween(\vec{front},\vec{right}reverse())$
$LB(A,B)$	$\exists$ point P $\in$ A, s.t. $\vec{B_{lb}}P.isBetween(\vec{front}.reverse(),\vec{right}reverse())$

Table 3.4: Direction Predicates for MBR Objects

direction predicates between regions. Right column provides a definition for the corresponding predicate using basic concepts from section 3.2.1. A predicate between target region A and reference region B can be defined in terms of the directional relationship between a point inside A and the four corner points of B. It's easily explained via Figure 3.7. The orientation O_B of the reference object B illustrated in figure 3.7(a) contains two orthogonal direction objects $\vec{front}$ and $\vec{right}$ representing B's *front* and *right* directions. Figure 3.7(b) illustrates the corresponding direction regions for direction predicates. The directions $left$ and *behind* can be represented as $\vec{right}.reverse()$ and $\vec{front}.reverse()$ by using the *reverse* operator of the direction object. The discussions can be extended to



(a) Reference object

(b) Direction regions

(c) EF(A,B) iff ∃ a point P ∈ A, $B_{lf}\vec{P}.isBetween(\vec{front}, \vec{right}) \wedge B_{rf}\vec{P}.isBetween(\vec{front}, \vec{right}.reverse())$

Figure 3.7: Direction Regions for MBR Objects, B_{lf}, B_{lb}, B_{rf} and B_{rb} represent B's left-front, left-behind, right-front and right-behind corners respectively

three dimensions by adding an orthogonal axis $\vec{above}$.

The predicate set defined in Table 3.4 consists of nine predicates, each of which checks whether the target object A is overlapped with a specific direction region of object B with respect to B's orientation. $EF(A,B)$, $EB(A,B)$, $EL(A,B)$, and $ER(A,B)$ return TRUE if and only if object A is overlapped with the EF, EB, ER, and EL regions of B respectively as shown in figure 3.7. This can be formally defined using $isBetween$ operator. For instance, $EF(A,B)$ is TRUE if and only if there exists a point P in A, such that, point P is within the direction region of EF. In other words, the direction $B_{lf}\vec{P}$ is between $\vec{front}$ and $\vec{right}$ and the direction $B_{rf}\vec{P}$ is between $\vec{front}$ and $\vec{right}.reverse()$ as listed in Table 3.4. Similarly, the predicates $RF(A,B)$, $LF(A,B)$, $RB(A,B)$, and $LB(A,B)$ will be satisfied if and only if there exists any point Q in A that Q is in the corresponding direction region. For example, $RF(A,B) = TRUE$ iff there is a point Q satisfying that $B_{rf}\vec{Q}$ is between $\vec{front}$ and $\vec{right}$.

Lemma 1 *Predicate set defined in Table 3.4 is complete, i.e., For any pair of simple region objects A and B with no holes, The direction of target object A with respect to the reference B can be represented in terms of a single predicate or a disjunction of several*

predicates in the predicate set.

Proof: As a simple convex objects without holes, the target region object A must contain at least three points that do not form a line segment. It is impossible that all the points are located on the boundary between two direction regions. In other words, there must exist at least one point located in one of the direction region. And hence, the directional relationship can be represented by the predicate that corresponding to that direction region. ■

3.3 Proposed Approach for Object-Orientation-based Direction Queries

We propose an open shape based strategy(OSS) using the actual direction regions modeled as open shapes. The processing of the direction predicates is then converted to the processing of the topological operations between target object and the corresponding open regions. In this section, we will first formalize the open shape based strategy and then describe new ADT of open shapes to explain the approach. This top-down description is used to simplify the presentation.

3.3.1 Proposed Open Shape Based Strategy(OSS)

The goal of the open shape based strategy(OSS) is to improve filtering efficiency by eliminating false hits at the earliest opportunity while recursively searching R-tree. OSS uses the actual direction region as the filter to exclude those intermediate nodes that do not satisfy the direction predicate. No refinement step is needed if objects are MBRs. Algorithm 3 shows the steps of OSS. First, OSS constructs a new ADT **OpenShape** to

Algorithm 3 OSS: Processing object-orientation-based query using OpenShape-based Strategy

```
Input: rtree points to the R-tree of data objects;
       referenceObject is the reference object with intrinsic orientation;
       dirPredicate is the direction predicate involved in the selection condition of the query;
Output: resultSet contains all object MBRs which satisfy the query

OSS(R-tree *rtree, Geometry referenceObject, Predicate dirPredicate) {
    Object *resultSet = ∅ ;
    OpenShape dpRegion = ConstructOpenShape (referenceObject, dirPredicate);
    osQuery(rtree, dpRegion, resultSet); //find all objects interiorIntersect dpRegion
}

osQuery(R-tree *rtree, OpenShape dpRegion, Object *resultSet) {
    R-tree *currentNode = rtree;
    if dpRegion.interiorIntersects(currentNode.Mbr) {
        if (currentNode is a leaf node)
            Add currentNode to resultSet;
        else
            for each childNode ∈ child nodes of currentNode
                osQuery(childNode, dpRegion, resultSet);
    }
}
```

represent the actual direction region. After obtaining *dpRegion*, OSS calls the subroutine **osQuery** with parameters *rtree* and *dpRegion* to find all objects whose interiors intersect with the interior of *dpRegion*. The subroutine **osQuery** recursively checks if the MBR of the current node intersects with *dpRegion*. The function *interiorIntersects* is designed as a member function of **OpenShape**, and returns TRUE if and only if the interior of the input object intersects with the interior of OpenShape. Since OSS models the direction region as an OpenShape, it does not need to know the boundary of the embedding world, and thus eliminating the computation related to the world boundary.

Revisit the example in figure 3.5, given reference object B and its orientation, we want to find all objects that are *EF* of B. First OSS constructs the OpenShape(*dpRegion*) of the direction region represented by shadow in figure 3.5. After *dpRegion* is created, the second step is to check *interiorIntersects* relationships for the entries in the root of the R-tree. Only N3 remains after retrieving the first level of the R-tree, and N31 and N32 are in resultSet. Table 3.5 lists the steps of query processing using OSS strategy. For MBR objects, no refinement step needs to be performed.

Steps	MBRs excluded	MBRs remain
level 1	N1, N2, N4	N3
level 2	N33, N34	N31, N32

Table 3.5: Trace of the query processing using OSS for the example in Figure 3.5

Theorem 1 *Proposed strategy OSS is complete and correct, i.e., OSS can find all objects satisfying the direction predicate and all the objects in the resultSet satisfy the given direction predicate.*

Proof: OSS first constructs *dpRegion* as an OpenShape, and then use the function *dpRegion.interiorIntersects* to exclude objects whose interiors do not intersect with the interior of *dpRegion*. In other words, OSS only excludes MBRs of objects which do not intersect the interior of the corresponding direction region, namely, MBRs of the objects which do not satisfy the direction predicate. This guarantees completeness, i.e.,

if MBR of an object satisfies the given direction predicate, then its MBR will not be eliminated by OSS.

An object is added to the *resultSet* if and only if the MBR of the object *interiorIntersects dpRegion*. As we discussed in previous section, the interior of dpRegion represents the actual direction region satisfying the corresponding direction predicate. If the MBR of the target object A *interiorIntersects* dpRegion, MBR of A must satisfy the directional relationship. Therefore, all the MBR-objects in resultSet must satisfy the direction predicate, guaranteeing correctness. ■

We now provide the definition of new ADT **OpenShape**. OpenShape is defined as a base class hierarchy to represent open geometries whose boundaries are not closed. We focus on the data types that are needed to model open direction region. The function interfaces for **OpenShape** related to the processing of direction queries can be defined in C++ notation as follows:

```
class    OpenShape {
public:
         virtual Boolean interiorIntersects(Geometry) const;
         virtual Boolean overlaps(Geometry) const;
         virtual Boolean contains(Geometry) const;
         virtual Boolean crosses(Geometry) const;
         ⋮
};
```

In the following, we will discuss the *OpenShape* and its derived classes and use these classes to represent direction region for each direction predicate in the defined predicate set.

3.3.1.1 Defining OpenShape

Open shapes refer to the geometries whose boundaries are partially defined. Open shape objects have infinite interiors and extend beyond the boundary of the embedding world. Figure 3.8 shows some examples of open shapes that are useful for processing direction queries. Figure 3.8(a)-(b) are open lines with one-end and two-end open respectively, while (c)-(e) are open rectangles. Note that in figure 3.7, the direction region for each

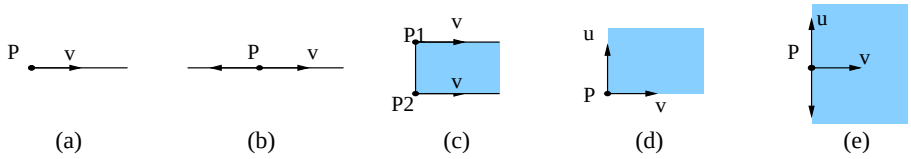


Figure 3.8: Examples of open shapes useful in processing direction queries

direction predicate is an open rectangle.

The *boundary* of an open shape is a set of geometries of the next lower dimension. The boundary of an one-end open line consists of the only endpoints, and the boundary of an two-end open line is empty. The boundary of an open rectangle consists of the set of sides which could be either line segments or open lines. The *interior* of an open shape consists of those points of the open shape that remains when the boundary is removed. For instance, the interior of an one-end open line is the set of points on the line except the endpoints, and the interior of a two-side open rectangle is the rectangle region which removes the boundary and indefinitely extending to the two other directions without boundary.

It would be useful if there were corresponding object classes so that users can use open shapes at abstract level. The new ADT *OpenShape* is defined as a base class of open shapes. New data types for open lines and open rectangles can be defined as derived classes of **OpenShape**, and the attributes can be described using directions and points. Table 3.6 defines ADT for each type of open shapes shown in figure 3.8. In the table, we only show the attributes and constructors of each open shape class. Examples for respective data types using parts of figure 3.8 are provided in the last column of the table. An OpenLine1 is defined by its endpoint and extending direction, and an OpenLine2 extending both ends and needs an intermediate point and a direction to describe it. The attributes needed to represent OpenLine1 and OpenLine2 are identical. The difference between them shows up in the definitions of various operations.

Three new data types are defined for the three subtypes of open rectangles. OpenRect1 is an one-side open rectangle(e.g. figure 3.8c) described by two endpoints of the rectangle and one direction representing the open direction. OpenRect2 is a two-side

ADT	attributes	constructors	examples in figure 3.8
OpenLine1	$startPoint : Point$; $dir : Direction$;	OpenLine1(Point, Direction);	(a): $OpenLine1(P, \vec{v})$
OpenLine2	$interPoint : Point$; $dir : Direction$;	OpenLine2(Point, Direction);	(b): $OpenLine2(P, \vec{v})$
OpenRect1	$vertex_1 : Point$; $vertex_2 : Point$; $dir : Direction$;	OpenRect1(Point, Point, Direction);	(c): $OpenRect1(P_1, P_2, \vec{v})$
OpenRect2	$startPoint : Point$; $dir1 : Direction$; $dir2 : Direction$;	OpenRect2(Point, Direction, Direction);	(d): $OpenRect2(P, \vec{u}, \vec{v})$
OpenRect3	$line : OpenLine2$; $dir : Direction$;	OpenRect3(OpenLine2, Direction);	(e): $OpenRect3(OpenLine2(P, \vec{u}), \vec{v})$

Table 3.6: Abstract Data Types for Open Lines and Open Rectangles

open rectangle(e.g. figure 3.8d) described by an endpoint and an ordered pair of directions. OpenRect3 is a half plane or three-side open rectangle(e.g.figure 3.8e) and could be defined by an OpenLine2 and a direction.

3.3.1.2 Interpreting Direction Predicates using *OpenShape*

After defining the ADTs for open lines and open rectangles, we can construct the open direction region for each direction predicate. As illustrated in figure 3.7, the direction region of each object-orientation-based direction predicate is of type OpenRect1 or OpenRect2. For instance, the direction region for the EF predicate is an OpenRect1. A summary of the correspondence between direction regions and their OpenShapes is shown in table 3.7.

Predicates	Type of Open Shape	Predicate true iff A $interIntersects$ the following open regions
$EF(A, B)$	OpenRect1 1-side open rectangle	$OpenRect1(B_{lf}, B_{rf}, \vec{front})$
$EB(A, B)$		$OpenRect1(B_{lb}, B_{rb}, \vec{front}.reverse())$
$ER(A, B)$		$OpenRect1(B_{rf}, B_{rb}, \vec{right})$
$EL(A, B)$		$OpenRect1(B_{rf}, B_{rb}, \vec{right}.reverse())$
$RF(A, B)$	OpenRect2 2-side open Rectangle	$OpenRect2(B_{rf}, \vec{right}, \vec{front})$
$LF(A, B)$		$OpenRect2(B_{lf}, \vec{right}.reverse(), \vec{front})$
$RB(A, B)$		$OpenRect2(B_{rb}, \vec{right}, \vec{front}.reverse())$
$LB(A, B)$		$OpenRect2(B_{lb}, \vec{right}.reverse(), \vec{front}.reverse())$

Table 3.7: Open shape representation for direction regions

By representing each direction region as an OpenShape, we process direction predicates using topological relationship between MBR objects and OpenShapes. This is the basic idea behind the proposed OSS strategy. When the direction region of a predicate is an open rectangle, the predicate is satisfied if and only if the target object *interiorIntersects* the corresponding open rectangle. For example, RF(A,B) return TRUE[2] if and only if A is interiorIntersected with $OpenRect2(B_{rf}, \vec{right}, \vec{front})$. We will discuss the topological operations next.

3.3.2 Topological Operations on Open Shapes

In this section, we will focus on defining a topological operation, *interiorIntersects*, since this operation is needed to solve direction predicates as shown in table 3.7. We define *interiorIntersects*(open shape O, closed shape C), where O could be an OpenRect1 or an OpenRect2, and closed shape C could be a rectangle. The implementation of *interiorIntersects* uses two other topological operations of *contains* and *crosses*, which are defined in Appendix 3.7. Other possible topological operations (e.g. *touches*, *disjoint*[32]) may be addressed in future work. In the rest of this section, O refers to an open shape, C refers to a closed rectangle. The algorithm may apply to more general cases of C such as simple convex polygon.

3.3.2.1 The *interiorIntersects* Operation

The *interiorIntersects* operation is defined for (OpenRect1, rectangle) and (OpenRect2, rectangle) situations. For any such pair of open shape O and closed shape C, the *interiorIntersects* relation between them is defined as:

$$O.interiorIntersects(C) = TRUE \Longleftrightarrow interior(C) \bigcap interior(O) \neq \emptyset$$

The semantics of *interiorIntersects* covers two topological operations, *contains* and *overlaps*, defined in OGIS[32] for closed geometry pairs. In other words, O interiorIntersects C if C is contained in O, or C overlaps O. Since a closed object C cannot contain an open object O due to infinite extent, we don't have to check for C contains O. We

[2]Note that the boolean value of a topological operations between a closed shape and an OpenShape can be determined without looking at the boundary of embedding world[65].

will discuss the implementation of the OpenRect1.*interiorIntersects*(Rectangle) and OpenRect2.*interiorintersects*(Rectangle) respectively.

The *interiorIntersects* Operation for OpenRect1

An OpenRect1 *interiorIntersects* a rectangle if there exists a set of points which belong to both the interior of the OpenRect1 and the interior of the rectangle. Figure 3.9 shows several situations that an OpenRect1 *interiorIntersects* a rectangle.

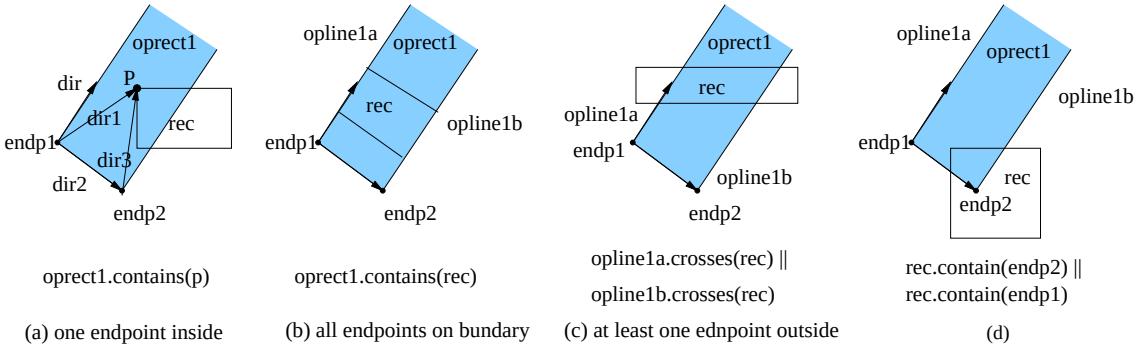


Figure 3.9: OpenRect1 *interiorIntersects* a rectangle

The boolean value of *interiorIntersects* operation can be determined by enumerating all possibilities of the interiorIntersecting relationship between an OpenRect1 and a rectangle. O.*interiorIntersects*(R) is TRUE if and only if at least one of the following three conditions is true:

- At least one of the endpoints of R is contained in O(e.g. figure 3.9(a));
- All endpoints of R are on the boundary of O(figure 3.9(b)).
- At least one endpoint of P is outside O(figure 3.9(c)) and at least one OpenLine1 of the two OpenLine1s boundary of O crosses R, i.e., OpenLine1(O.endp1, O.dir) crosses R or OpenLine1(O.endp2, O.dir) crosses R.

The first case can be tested by using the *OpenRect*1.*contains*(*Point*) operation, and the second case is checked by the *OpenRect*1.*contains*(*Rectangle*) operation. Both

operations are described in Appendix 3.7.2. The third case is checked by using the $OpenLine1.crosses(Rectangle)$ operation defined in Appendix 3.7.1.

Efficient implementation of *interiorIntersects* operation may be other fast filter to reduce computation of above three tests. For example, if any of the endpoints of O is within R, then *interiorIntersects* operation can be determined to be true without further tests. This is illustrated in Figure 3.9(d). This strategy is useful for some special polygon, such as MBR, where the $MBR.contains(Point)$ operation is cheaper. The pseudo-code for the *interiorIntersects* operation is described as Algorithm 4.

Algorithm 4 OpenRect1 *interiorIntersects* a rectangle

```
Input: rec is the rectangle that needs to be checked;
       the current OpenRect1 object, which has attributes (endp1, endp2, dir);
Output: TRUE if interiorIntersects, FALSE otherwise

OpenRect1::interiorIntersects(Rectangle rec) {
    /* case 1, figure 3.9(a) */
    for each ep ∈ endpoints of rec
        if (contains(ep))
            return TRUE;

    /* case 2, figure 3.9(b) */
    if (contains(rec))
        return TRUE;

   /* case 3, figure 3.9(c) */
    opline1a = OpenLine1(endp1, dir);
    opline1b = OpenLine1(endp2, dir);
    if ( opline1a.crosses(rec) || opline1b.crosses(rec) )
        return TRUE;

    return FALSE;
}
```

The *interiorIntersects* Operation for OpenRect2

OpenRect2.*interiorIntersects*(Rectangle) is TRUE if there exists a set of points which belong to both the interior of the OpenRect2 and the interior of the rectangle. Figure 3.10 shows several possible situations that an OpenRect2 *interiorIntersects* a polygon. Similar to the interiorIntersect operation for OpenRect1, the boolean value of $OpenRect2.interiorIntersects(Rectangle)$ operation can be determined by enumerating

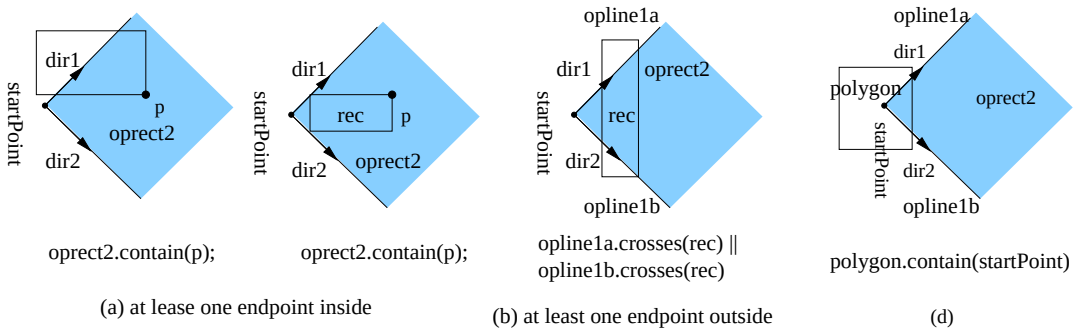


Figure 3.10: An OpenRect2 *interiorIntersects* a Polygon

all possible relationship between an OpenRect2 and a rectangle. O.*interiorIntersects*(R) is TRUE if and only if at least one of the following two conditions is true:

- At least one of the endpoints of R is contained in O(e.g. figure 3.10(a));
- At least one endpoint of P is outside O(figure 3.10(b)) and at least one OpenLine1 of the two OpenLine1s boundary of O crosses R, i.e., OpenLine1(O.startPoint, O.dir1) crosses R or OpenLine1(O.startPoint, O.dir2) crosses R.

The first case can be tested by using *OpenRect*2.*contains*(*Point*) operation described in Appendix3.7.2. The second case is checked by using *OpenLine*1.*crosses*(*Rectangle*) operation defined in Appendix3.7.1. Note that for OpenRect2, the endpoints of any rectangle can not be all on the boundary of the OpenRect2, the above two cases are complete. Efficient implementation of *OpenRect*2.*interiorIntersects*(*rectangle*) operation may be possible to use fast filter. figure 3.10(c) illustrates a cheap checking for special rectangles such as MBRs. The pseudo-code is described by algorithm 5.

Lemma 2 *Algorithm* **OpenRect1::interiorIntersects** *and* **OpenRect2:: interiorIntersects** *are complete and correct.*

Proof: Figure 3.11 shows the flow chart of algorithm 4. The algorithm tests three cases, and it returns TRUE if and only if at least one of these three conditions is true. This guarantees the correctness of the algorithm, i.e., the algorithm returns TRUE only if the OpenRect1 *interiorIntersects* the rectangle.

Algorithm 5 OpenRect2 *interiorIntersects* Rectangle

Input: rec is the rectangle that needs to be checked;
the current OpenRect2 object, which has attributes $(startPoint, dir1, dir2)$;
Output: $TRUE$ if interiorIntersects, $FALSE$ otherwise

```
OpenRect2::interiorIntersects(Polygon aPolygon) {
      /* case 1, figure 3.10(a) */
      for each ep ∈ endpoints of aPolygon
            if (contains(ep))
                  return TRUE;

      /* case 2, figure 3.10(b) */
       opline1a = OpenLine1(startPoint, dir1);
       opline1b = OpenLine1(startPoint, dir2);
       if ( opline1a.crosses(aPolygon) || opline1b.crosses(aPolygon) )
            return TRUE;

       return FALSE;
}
```

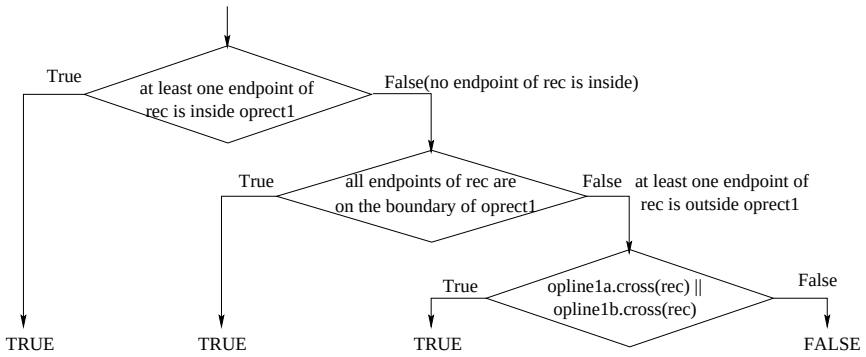


Figure 3.11: Control Flow of the Algorithm

Plane sweep strategy can help to prove the completeness of the algorithm. Let's consider a pair of disjoint OpenRect1 and rectangle as illustrated in figure 3.12(a). Move the rectangle horizontally and the first case rectangle touches the OpenRect1 is illustrated in 3.12(b). When continuously moving the rectangle horizontally, OpenRect1 *interiorIntersects* the rectangle, and the relationship between OpenRect1 and Rectangle falls into the three cases in the flow chart. This guarantees the completeness of the algorithm, i.e., the algorithm returns FALSE only when there is no intersection between the interior of OpenRect1 and the interior of the rectangle.

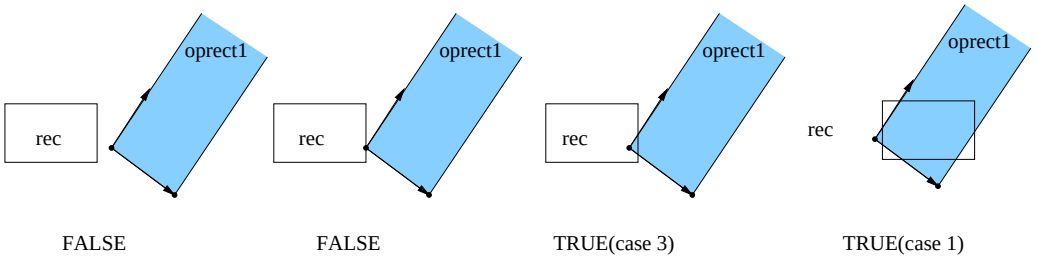


Figure 3.12: Plane sweeping argument

The completeness and correctness for the algorithm OpenRect2::interiorIntersects can be demonstrated similarly. ■

3.4 Algebraic Analysis

In this section, we perform algebraic analysis for the relative performance of the proposed open shape based strategy(OSS) vs. the classical range query strategy(RQS). We will first give a cost model for OSS and RQS, and then characterize the analysis in terms of theorems and lemmas. Since the performance is similar for the predicates whose direction regions correspond to the same OpenShape type, the analysis is performed in terms of two types of predicates, namely, the predicates whose direction regions correspond to OpenRect1 and the predicates whose direction regions correspond to OpenRect2. In the rest of the chapter, we refer the first predicate type as *EF-type*, and the second as *LF-type*. Examples of EF-type predicates include EF, EB, ER, and EL. Examples of LF-type predicates include LF,LB, RF, and RB.

I/O Cost Models

The I/O cost of a query is evaluated in terms of the number of page accesses needed in order to solve the query. RQS accesses the R-tree pages whose MBR interiorIntersects the MBR of the direction region, while OSS accesses the R-tree pages whose MBRs interiorIntersects the actual direction region. Figure 3.13 illustrates the direction region

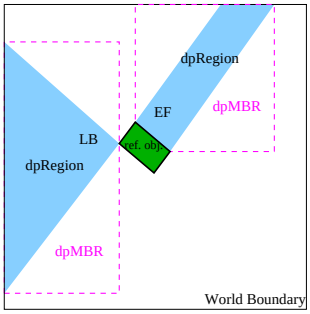


Figure 3.13: The ratio of RQS vs. OSS

and their corresponding MBRs for predicates EF and LB. The shadow regions are actual direction regions and the dashed rectangles are MBRs of the direction regions. We first make the following assumptions before we introduce the cost model:

- Data objects are uniformly distributed on embedding space.
- size of MBR objects $\ll$ size of world
- size of MBR objects $\ll$ size of direction regions

Let dpRegion(dp) represent the actual direction region for direction predicate dp, and dpMBR(dp) refer to the MBR of dpRegion(dp). Under above assumptions, the number of R-tree pages whose MBR interiorIntersects dpRegion can be approximated by the area of the dpRegion. Similarly, the number of R-tree pages whose MBR interiorIntersects dpMBR can be approximated by the area of the dpMBR. This is accurate when objects are uniformly distributed over the space and the object sizes are much smaller than the sizes of direction regions. In general these assumptions do not hold, nevertheless, this simple model is useful in understanding the relative performance of the two strategies OSS and RQS. Formally, the I/O cost is approximated as follows:

```
I/O cost(OSS, dp) ≈ area(dpRegion(dp));
I/O cost(RQS, dp) ≈ area(dpMBR(dp))=area(MBR(dpRegion(dp)));
```

The following lemmas and theories characterize the performance properties using this cost model under same assumptions.

Lemma 3 *The following statement for I/O cost is true for processing any direction predicate dp: I/O cost(OSS, dp) $\leq$ I/O cost(RQS, dp);*

Proof: By definition of MBR, for any polygon P, area(P) $\leq$ area(MBR(P)). Considering P = dpRegion(dp) for any predicate *dp*, we have

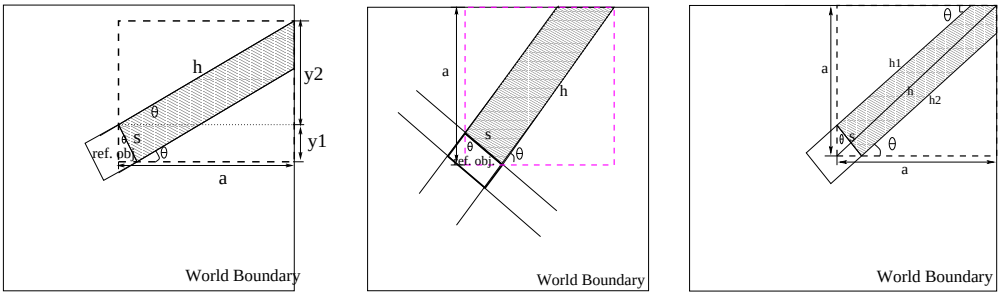
$$area(dpRegion(dp)) \leq area(MBR(dpRegion(dp))$$

Lemma 4 *For any EF-type predicate dp in {EF,EB,EL,ER}, the I/O cost(OSS, dp) and I/O cost(RQS, dp) can be approximated as follows:*

$$\begin{aligned}
\texttt{I/O cost(OSS,dp)} &= \tfrac{as}{\cos\theta};\ (0^0 \leq \theta \leq 45^0) \\
&= \tfrac{as}{\sin\theta};\ (45^0 \leq \theta \leq 90^0) \\
\texttt{I/O cost(RQS,dp)} &= as\cos\theta + a^2\tan\theta;\ (0^0 \leq \theta \leq 45^0) \\
&= as\sin\theta + a^2\cot\theta;\ (45^0 \leq \theta \leq 90^0)
\end{aligned}$$

where a = the distance from the centroid of the reference object to the global boundary that intersects the dpRegion, s = size of the reference object orthogonal to the direction of dp, θ = angle between direction of dp and x-axis.

Proof: Recall the assumption that $s << a$. Assume that the reference object is at the centroid of a square-shape embedding space for simplicity. The change of orientation of the reference object leads to three different cases: *dpRegion* intersects with a boundary on X dimension(figure 3.14(a)); *dpRegion* intersects with a boundary on Y dimension(figure 3.14(b)); *dpRegion* intersects with the boundary on both dimensions(figure 3.14(c)). Since the reference object is located at the center of the embedding



(a) case 1: Intersect with boundary on X dimension

(b) case 2: Intersect with boundary on Y dimension

(c) case 3: Intersect on both dimensions

Figure 3.14: Three cases for different orientations

space, case 1 corresponds to the situation of $0 \leq \theta < 45^0$, case 2 corresponds to the situation of $90^0 \geq \theta > 45^0$, and in case 3, $\theta \rightarrow 45^0$. The exact boundaries of the three cases are sensitive to the ratio s/a. Since we assume $s << a$ for simplicity, case 3 can be approximated by either case 1 or case 2 for $\theta = 45^0$.

case 1(3.14(a)): $area(dpMBR) = a(y_1 + y_2) = a(s\cos\theta + a\tan\theta)$;

$area(dpRegion) \leq sh = \frac{as}{\cos\theta}$

case 2(3.14(b)): $area(dpMBR) = a(s\sin\theta + a\cot\theta) = as\sin\theta + a^2\cot\theta$;

$area(dpRegion) \leq sh = \frac{as}{\sin\theta}$ ■

Theorem 2 *For any EF-type predicate dp* $\in$*{EF, EB, EL, ER}, for a fixed size of reference object located in the center of the embedding space, the maximum I/O improvement is achieved when* $\theta = \frac{\pi}{4} + k \times \frac{\pi}{2}$*(k is an integer), while the minimum improvement is obtained when* $\theta = \frac{\pi}{2} \times k$*(k is an integer).*

Proof: According to lemma 4, the I/O cost ratio can be approximated using one of the two formulas according to the orientation of the reference object:

$$\texttt{I/O cost ratio} = \frac{I/Ocost(OSS,dp)}{I/Ocost(RQS,dp)} \quad = \frac{a}{s}\sin\theta + \cos^2\theta; \;\; (0^0 \leq \theta \leq 45^0)$$
$$= \frac{a}{s}\cos\theta + \sin^2\theta; \;\; (45^0 \leq \theta \leq 90^0)$$

Figure 3.15 shows the I/O cost ratio curve for orientation degree θ in the range of $[0, 90^0]$, for some fixed values of s and a. We can notice that the maximum value of the I/O ratio is achieved when $\theta = 45^0$, and the minimum ratio is obtained when $\theta = 0^0$ and 90^0. We can also differentiate the functions for I/O cost ratio to verify the maximum

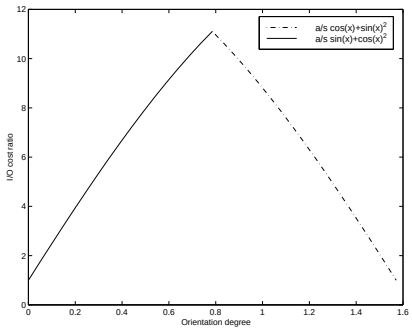


Figure 3.15: The I/O ratio curve for $\theta \in [0^0, 90^0]$ according to lemma 4

and minimum point. ■

Lemma 5 *For any EF-type predicate dp* $\in$*{EF, EB, EL, ER}, I/O cost ratio is affected by the size of reference objects. The smaller the object size, the larger the I/O improvement ratio.*

Proof: According to lemma 4, we have,

$$\texttt{I/O cost ratio} = \frac{I/Ocost(OSS,dp)}{I/Ocost(RQS,dp)} \quad = \frac{a}{s}\sin\theta + \cos^2\theta; \;\; (0^0 \leq \theta \leq 45^0)$$
$$= \frac{a}{s}\cos\theta + \sin^2\theta; \;\; (45^0 \leq \theta \leq 90^0)$$

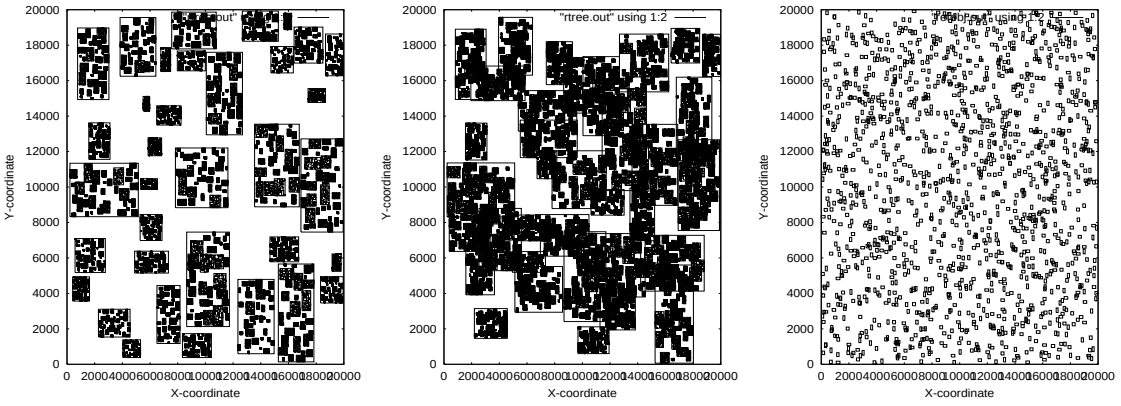
Since in the range of $0 < \theta < 90^0$, both $cos(\theta)$ and $sin(\theta)$ are greater than 0. Therefore, when parameters a and θ are fixed, the decrease of s results into the increase of I/O cost ratio.

For any predicate dp $\in$ { LF,LR,RF,RB}, the I/O cost ratio is between 1 and 2, the maximum ratio is achieved when $\theta = \frac{\pi}{4} + k \times \frac{\pi}{2}$(k is an integer). The argument for this is similar.

3.5 Experimental Evaluation

3.5.1 Experiment Design

We now design an experiment to compare the performance of classical range query strategy(RQS) with the performance of open shape based query strategy(OSS) for the predicate set defined in Table 3.4. We use several randomly generated datasets which contain MBRs of data objects with varying sizes. The query file containing the reference object MBRs has 1500 objects. Example dataset and reference object set are shown in figure 3.16.



(a) Dataset R-tree without overlap among objects

(b) Dataset R-tree with overlap among objects

(c) Set of reference Object MBRs with orientation degree=90^0

Figure 3.16: Example of datasets

A set of queries are selected to cover different types(LF-type and EF-type) of direction predicates. LF family includes the predicates of LF, LB, RF, RB. EF family includes the predicates of EF, EB, ER, EL. On average, selectivity of EF family predicates is lower than that of LF family predicates. We consider the following queries:

- Query A_1: Given a reference object, find all objects that are EF of it.
- Query A_2: Given a reference object, find all objects that are EB of it.

- Query A_3: Given a reference object, find all objects that are ER of it.
- Query A_4: Given a reference object, find all objects that are EL of it.
- Query B_1: Given a reference object, find all objects that are LF of it.
- Query B_2: Given a reference object, find all objects that are LB of it.
- Query B_3: Given a reference object, find all objects that are RF of it.
- Query B_4: Given a reference object, find all objects that are RB of it.

The predicates in query Ai are of EF-type, and the direction predicates involved in query Bi are of LF-type. In general, query Ai has lower selectivity than query Bi.

The variable parameters for the experiment include the orientation of reference objects, overlap degree among objects in the dataset, the size of reference objects and the number of objects in the dataset. The metrics for evaluation include the number of page accesses and the number of operations required by each strategy to retrieve the query.

The orientation of a reference object is defined as the angle between $\vec{front}$ direction of the object and the x-$axis$ of the global coordinate system. When the orientation is equal to 90^0, the $\vec{front}$ direction of the reference object is parallel to the $\vec{north}$ direction in the absolute reference frame. In our experiment, orientation varies from 0^0 to 180^0.

The overlap degree(OPD) between two objects is defined as the ratio of the area of overlapping region to the area of the smaller object. The overlap degree between objects A and B is calculated as $OPD = \frac{area(A \bigcap B)}{min(area(A), area(B))}$. The overlap degree of the dataset is defined as the average overlap degree of the non-disjoint object pairs in the dataset.

The size of the reference object is measured as the relative size of the reference object to the global size. The size varies from $\frac{1}{120}$ to $\frac{1}{10}$ of the global size.

Figure 3.17 shows various steps of the experiment. Given the range of object size and the overlap degree, the data set is randomly generated, and the R-tree of the dataset is created based on the given branching factor. The set of reference objects is randomly generated according to its size range, which can be either equal to the size range of

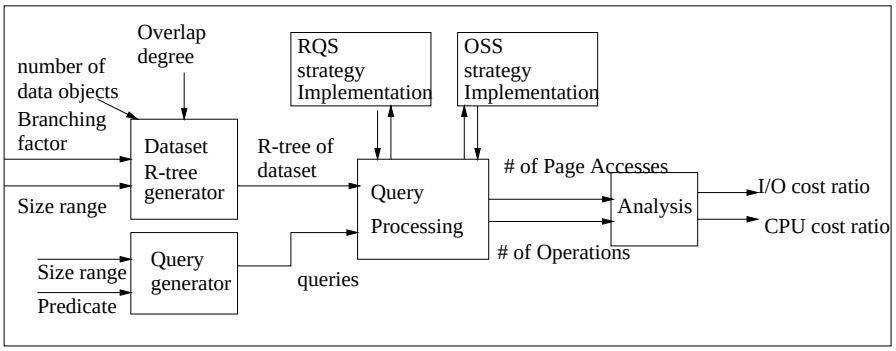


Figure 3.17: Experimental setup and design

dataset or independently given. Using R-tree and reference object set as input, "Query processing" simulates the behaviors of the RQS and OSS algorithms for different direction predicates. The total number of page accesses and the number of operations are tracked for each combination of algorithm, overlap degree, the orientation and size of reference objects and the dataset size to derive the experiment results.

3.5.2 Experiment Results

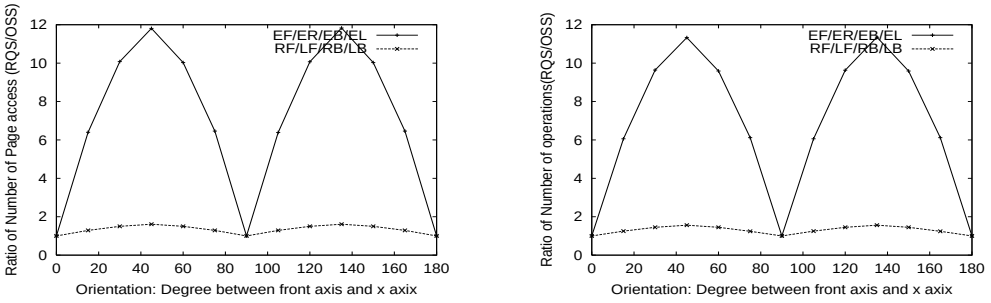
Figure 3.18, 3.19, 3.20 and 3.21 show the comparison between the classical range query strategy (RQS) and the proposed open shape based strategy(OSS) for I/O costs and CPU costs. I/O cost is measured in terms of the number of distinct pages fetched from disk. CPU cost is measured in terms of number of distinct line-line intersection operations used by an algorithm. Since the performance is similar for the predicates of same types, we only show two curves representing the relative performance of OSS and RQS for two different predicate types in each figure. The curve denoted by "EF/ER/EB/EL" illustrates the average relative performance improvement for EF-type predicates. The curve denoted by "LF/LB/RB/RB" illustrates the average relative performance improvement for LF-type predicates. For any predicate dp, the relative performance is defined as the ratio cost(RQS,dp)/cost(OSS,dp). Ratio > 1 means that OSS is cheaper than RQS.

The common observation in all figures is that OSS performs better than RQS both in terms of the number of page accesses(I/O) and the number of operations(CPU). For low selectivity predicates such as EF-type the performance of OSS is even better compared

to RQS.

Orientation of the reference object

Figure 3.18 shows the impact of the orientation of reference objects. The dataset consists of 42875 spatial objects without overlap. We use 1500 different reference objects with identical orientations, and the size of each object equals one percent of the world. The orientations of reference objects vary from 0^0 to 180^0, at an interval of 15 degrees. Recall the orientation of an object is measured by the angle between its $front$ direction and the $x\text{-}axis$ of the global coordinate system. Figure 3.18 (a) and 3.18(b) show the average page accesses and operations performed across 1500 reference objects respectively. OSS



(a) Effect on Number of Page access

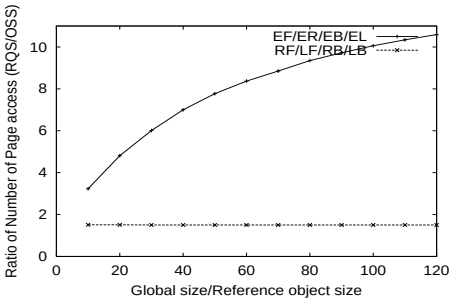
(b) Effect on the Number of Operations

Figure 3.18: Effects of orientations of reference objects(orientation: $0^0, 15^0, \cdots, 180^0$, overlap degree of the data set = 0, reference object size = 1% of global size, dataset size = 42875)

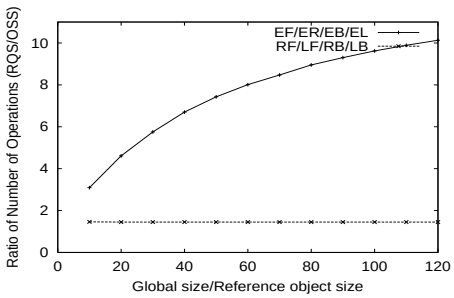
and RQS have the same I/O and CPU cost when the orientation is 0^0, 90^0, or 180^0 with the cost ratio equal to 1. OSS performs better than RQS for most orientations with best relative performance for OSS at 45^0 and 135^0. Relative performance advantage of OSS for predicates of EF-type varies cyclically. This result is consistent with Theorem 2 in section 4.

The size of reference objects

Figure 3.19 shows the effect of the size of the reference object on the I/O and CPU performance with fixed orientation, and fixed dataset. The dataset has 42875 objects. We use 1500 different reference objects without overlap and with fixed orientation of 30^0. The choice of orientation is arbitrary. We vary size of reference objects taking values of $\frac{1}{120}$, $\frac{1}{110}$, $\frac{1}{100}$, $\cdots$, $\frac{1}{10}$ of the total world. The *x-axis* in Figure 3.19 shows the ratio of $\frac{global\ size}{reference\ object\ size}$, thus the reference object size of 1% of the world corresponds to 100 in *x-axis*. Note that relative gain of OSS for size = 1% is consistent with the relative gain in Figure 3.13 for orientation of 30^0. Obviously, OSS consistently outperforms RQS. The relative performance advantage for the EF-type predicates increases steadily with the decrease of the reference object size. This is consistent with Lemma 5.



(a) Effect on the number of page access



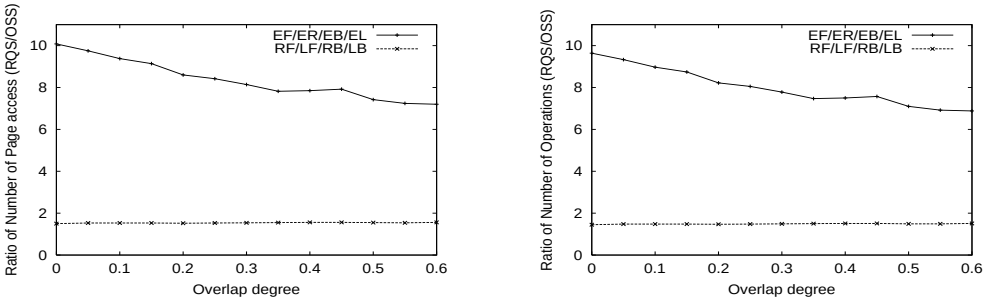
(b) Effect on the number of operations performed

Figure 3.19: Effects of the size of reference objects (orientation: 30^0, overlap degree of data set =0, dataset size = 42875 objects, the size of reference object varies from $\frac{1}{120}$ to $\frac{1}{10}$ of the global size)

Overlap degree

Figure 3.20 shows the effect of overlap degree of the dataset on the I/O and CPU performance. We used 1500 different reference object of fixed size(1%) and fixed orientation(30^0). Average overlap between objects in the data set varies from 5 percent to 60 percent. Recall that overlap degree between two objects is the ratio of common area to the smaller

area of the two objects. Again, OSS consistently outperforms RQS. The relative performance advantage of OSS diminishes slightly with the increase in the overlap degree among objects. Our algebraic cost models(section 4) do not predict the effect of overlap degree. We propose to extend our model towards this purpose in future work.

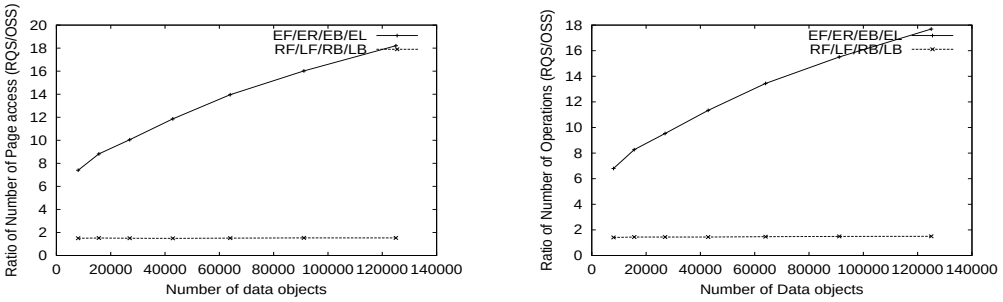


(a) Effect on Number of Page access (b) Effect on Number of Operations

Figure 3.20: Effects of the overlap degree of the data set(Orientation = 30^0, the size of reference object = %1 of global size, dataset size = 42875, overlap degree=$0, 0.05, \cdots, 0.6$)

The number of objects in dataset

Figure 3.21 shows the effect of the number of objects in dataset on I/O and CPU perfor-



(a) Effect on Number of Page access (b) Effect on Number of Operations

Figure 3.21: Effects of the number of objects in dataset (Orientation=30^0, Overlapping degree =0, the size of reference object = %1 global size)

mance. We used 1500 different reference object of fixed size(1%) and fixed orientation(30^0). The objects in dataset have no overlap. The dataset size varies from 8000 to 125000.

The relative performance of OSS improves with the increase in the number of objects in dataset. This result demonstrates that algorithm OSS scales better than RQS as the size of dataset increases. Our cost model does not explain why the relative advantage of OSS over RQS improves with the increase in dataset size. We would like to address this issue in future work.

3.6 Conclusions and Future Work

In this chapter, we focus on processing queries based on object-orientation-based directional relationships. Open Shape based Strategy(OSS) is proposed for processing object-orientation-based direction queries. OSS models direction regions as *OpenShape*s, and uses actual direction region to exclude potential false hits as early as possible. Since OSS models the direction region as an OpenShape, it does not need to know the boundary of the embedding world, thus eliminating the computation related to the world boundary. The algebraic analysis and experiment results demonstrate that the OSS consistently outperforms range query strategy(RQS) in terms of both I/O and CPU cost.

We assume the data objects are all MBR objects in this chapter, but the framework is applicable when the objects are arbitrary convex polygons. As a result, the improvement on the CPU cost will be more significant when applying to arbitrary objects. This is because the computational cost is much more expensive for RQS to exclude the large number of false hits at object level.

In future work, we would like to consider extending OSS to other direction predicate sets[49, 25, 31]. We would also like to address the processing of queries involving viewer-orientation-based direction predicates by deriving more class and operations for ADT *OpenShape*. We would also like to investigate processing orientation-based direction queries in a mobile environment. Applying OSS to robotics and motion planning applications[69] is another direction of future work.

3.7 Appendix: Topological operations for OpenShapes

3.7.1 The *crosses* Operation: *crosses*(OpenLine O, Closed shape C)

The *crosses* operation applies for object pair between an open line O and a closed shape C, where O could be OpenLine1 or OpenLine2 and C could be points, line segments and polygons. For any such pair of O and C, the *crosses* relation between them is defined as:

$O.crosses(C) = TRUE$

$\Longleftrightarrow O.\vec{dir} \odot Direction(C - O.startPoint) = 1$; (if C is a point)

$\Longleftrightarrow \exists$ a point P s.t. $P \in interior(O) \cap interior(C)$;(if C is not a point)

Figure 3.22 shows some examples of the *crosses* operations. We approximate polygonal closed shape by rectangle for simplicity, however, the algorithm will work for convex polygon as well. Figure 3.22(a) shows an OpenLine1 O crosses a point C, figure 3.22(b) shows O crosses a line segment C, and Figure 3.22(c) consists of the five possible cases that an OpenLine1 crosses a polygon C. To implement the *crosses* operation, we use

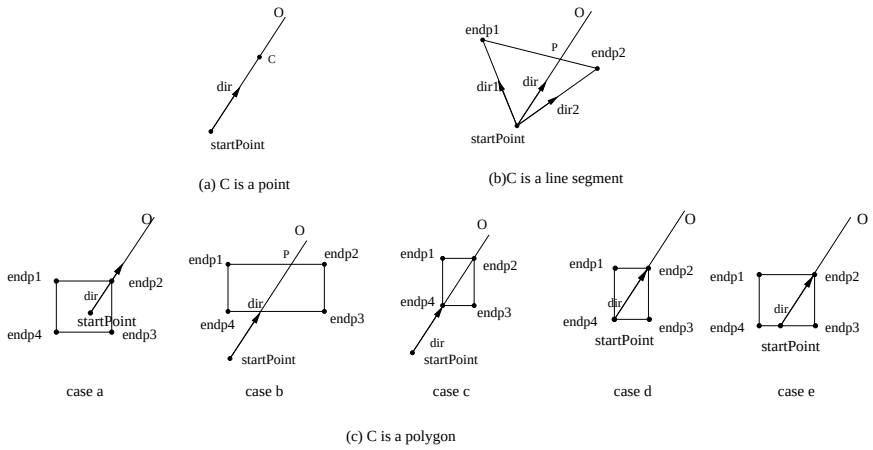


Figure 3.22: OpenLines crosses Point/Line/Rectangle

vector dot-product and the *isBetween* operation of direction objects. For the case of OpenLine1 crosses a given point, such as in figure 3.22(a), the OpenLine1 O *crosses* the given point C if and only if the vector-product between $O.\vec{dir}$ and the direction $Direction(C - O.startPoint)$ equals 1. For the case of OpenLine1(O) *crosses* a line(C)

as in figure 3.22(b), $O.\vec{dir}$ should be between the two directions constructed by the two endpoints and the O.startPoint, i.e. $\vec{dir}1$ and $\vec{dir}2$ in the figure. The *crosses* relation between an OpenLine1 and a polygon is more complicated for there are several situations possible as shown in figure 3.22(c). We solve them by enumerating all possible situations that O.crosses(C) will be TRUE. Algorithm 6 describes pseudocode of OpenLine1::crosses. The topological operations *contains* and *touches* for closed geometry pairs defined by OGIS [32] are used in the algorithm for checking if O.startPoint is in the interior or on the boundary of the polygon.

Lemma 6 *Algorithm* **OpenLine1::crosses** *is complete and correct.*

Proof: The completeness and correctness of the algorithm OpenLine1::crosses for Point and Line are proved by definition. In the following, we will demonstrate that OpenLine1::crosses(Polygon) is complete and correct.

- Completeness:

  For an OpenLine1(O) and a Polygon(C), there are two possibilities:

  - O crosses at least one of the sides of the polygon(figure 3.22(c) case b) $\Longrightarrow$ O.crosses(C) is TRUE;
  - O doesn't cross any side of C. The algorithm solves this possibility by considering the location of the startPoint.
    1. O.startPoint is within the polygon(figure 3.22(c) case a) $\Longrightarrow$ O.crosses(C) is TRUE ;
    2. O.startPoint is outside C, O.crosses(C) iff O crosses two non-adjacent endpoints of C(figure 3.22(c) case c) ;
    3. O.startPoint is on the boundary of C:

       Case c: O.startPoint is an endpoint of C, and O crosses an endpoint *ep* which is not adjacent to O.startPoint $\Longrightarrow$ O.crosses(C) is TRUE

       case d: O.startPoint falls on side s of C, O.crosses(C) is TRUE iff O crosses endpoint *ep* of C and $ep \notin$ endpoints of s.

For any other situation that is not described above, OpenLine1 does not cross the polygon. Since the algorithm *crosses* checked all the above situations, the algorithm will find all the *crosses* relation between an OpenLine1 and a polygon. Therefore, the algorithm is complete.

- Correctness:
  The algorithm returns TRUE if and only if one of the situation is satisfied, in which case, the OpenLine1 crosses the polygon. The algorithm only returns TRUE when the operation *crosses* hold, therefore, the algorithm is correct. ■

The *crosses* operation between an *OpenLine*2(two-end openline) opline2 and a closed shape can be implemented by converting the opline2 to two *OpeLine*1s: opline1a and opline1b. The *startPoint* of each *OpenLine*1 is the *interPoint* of opline2, and the *dir* of each *OpenLine*1 is *opline*2.*dir* and *opline*2.*dir.reverse*() respectively. The opline2 crosses the closed shape if and only if either opline1a or opline1b crosses the object.

3.7.2 The *contains* Operation for Open Rectangles

For any open rectangle O and closed shape C, $O.contains(C)$ returns TRUE if and only if C is wholly contained within O. O could be an OpenRect1 or an OpenRect2, and C could be a point, a line segment, or a polygon. The formal definition for O.contains(C) is :

$$O.contains(C) = TRUE \Longleftrightarrow (O \cap C = C) \text{ and } (interior(O) \cap interior(C) \neq \emptyset)$$

Figure 3.23 shows some examples of the *contains* operations. Figure 3.23(a) are examples that satisfy *contains* relationship, where figure 3.23(b) are examples that C is on the boundary of O, but not contained in O.

Since we are only interested in the operations that are used for directional queries, we restrict our discussion in the cases where O is an OpenRect1 or OpenRect2, and C is a point or a polygon.

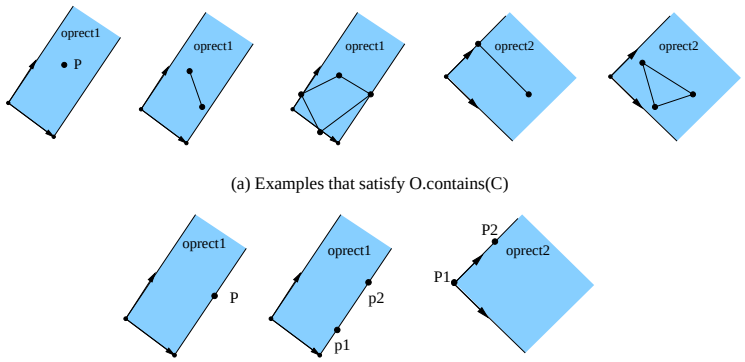


Figure 3.23: Examples of contains relationship

The *contains* Operation for OpenRect1

OpenRect1 is an one-side open rectangle. Figure 3.24 shows the examples that an OpenRect1 contains a point/polygon. The implementation of the *contains* operation can be

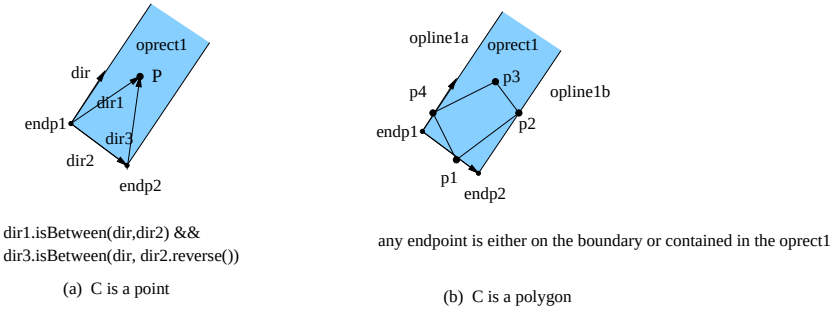


Figure 3.24: OpenRect1 contains point/line/polygon

accomplished by using *isBetween* operator and the *crosses* operation. In order to check if a point is contained in OpenRect1(figure 3.24a), two direction objects are constructed by subtracting the point C from each endpoints of the OpenRect1, and *isBetween* operations is applied.

For any convex polygon C, if there does not exist any endpoint that is outside the OpenRect1, i,e., every endpoint of C is contained in the OpecRect1 or located on the boundary of the OpenRect1, then C is contained in the OpenRect1. Algorithm 7 describes the operation in pseudocode.

The *contains* Operation for OpenRect2

OpenRect2 is a two-side open rectangle, which is described by one endpoint and two directions. Figure 3.25 shows several situations that an OpenRect2 contains a Point or a Polygon. Similar to the case of OpenRect1, the *isBetween* operation of the direction

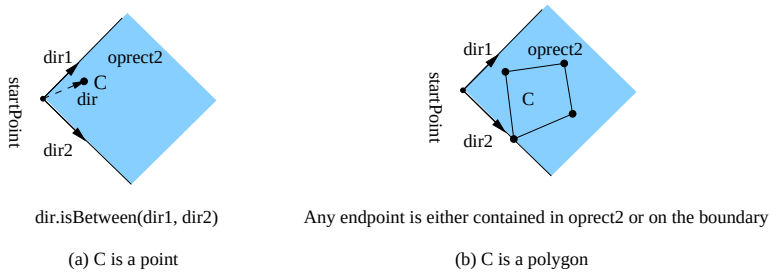


Figure 3.25: OpenRect2 contains point/line/polygon

objects is used to implement the *contains* operation between an OpenRect2 and a point. A convex polygon is contained in the OpenRect2 if every endpoints of the polygon is either contained in the OpenRect2 or on the boundary. The pseudocode is described in algorithm 8.

Completeness and Correctness of the algorithms

Lemma 7 *Algorithm* **OpenRect1::contains** *and* **OpenRect2::contains** *are complete and correct.*

Proof:

- Completeness:
  For object pair of OpenRect1(O) and Point(C), there is only one case that O.contains(C) is true, that is C is in the interior of O. The completeness of the algorithm is clear by definition. For the case of checking OpenRect1.contains(Polygon), the algorithm 7 excludes only the cases that there exist some endpoints of the polygon outside the OpenRect1, when the polygon is absolutely not wholly contained in OpenRect1. Therefore, the algorithm will find all cases when the relationship *contains*

is satisfied. The algorithm find all possible contain relations, so is complete.

- Correctness:

  The algorithm OpenRect1.contains(Point) returns TRUE if and only if the point is contained in the interior of the OpenRect1. For the object pair of an OpenRect1 and a convex polygon, the algorithm returns TRUE if and only if every endpoint of the polygon is either contained or on the boundary of the OpenRect1, in which case, there does not exist any point in the interior of the polygon which is not contained in the OpenRect2. therefore, the algorithm will return TRUE only when polygon is wholly contained in OpenRect1.

The argument of the completeness and correctness for algorithm 8 is similar. ■

Algorithm 6 OpenLine1 Crosses Points/Lines/Polygons

Input: closeShape is the topologically closed object that needs to be checked;
the current OpenLine1 object, which has attributes $(dir, startPoint)$;
Output: $TRUE$ if crossed, $FALSE$ otherwise

```
OpenLine1::crosses(Point closeShape) {
    dir1 = Direction(closeShape − startPoint);
    if (dir.deviate(dir1))
        return TRUE;
    return FALSE;
};
OpenLine1::crosses(Line closeShape) {
    dir1 = Direction(closeShape.endp1 − startPoint);
    dir2 = Direction(closeShape.endp2 − startPoint);
    if (dir.isBetween(dir1, dir2))
        return TRUE;
    return FALSE;
};
OpenLine1::crosses(Polygon closeShape){
    for each side ∈ sides of closeShape
        if crosses(side)
            return TRUE;
    /* The following deal with the cases O does not cross any side of the polygon */
    if (closeShape.contains(startPoint))      /* figure 3.22(c) case a: startPoint in the polygon */
        return TRUE;
    if (closeShape.touches(startPoint)) {
        if startPoint ∈ endpoints of closeShape {      /* figure 3.22(c) case d */
            for each ep ∈ endpoints of closeShape
                if crosses(ep) && !adjacent(startPoint, ep)
                    return TRUE;
        } else {      /* figure 3.22(c) case e: startPoint is on one side of the polygon */
            for each ep ∈ endpoints of closeShape
                if crosses(ep) && ep ∉endpoints of the side that crosses startPoint
                    return TRUE; }
        return FALSE;
    }
    /* figure 3.22(c) case b and c : startPoint is outside of the polygon */
    for each ep ∈ endpoints of closeShape {
        if crosses(ep) {
            for each ep2 ∈ endpoints of closeShape -{ep}
                if (!adjacent(ep,ep2) && crosses(ep2))
                    return TRUE;    /*3.22(c) case d: O crosses non-adjacent endpoints.*/
        break; }
    return FALSE;
};
```

Algorithm 7 OpenRect1 contains Point/Polygon

Input: closeShape is the topologically closed object that needs to be checked;
the current OpenRect1 object, which has attributes $(endp1, endp2, dir)$;
Output: $TRUE$ if contain, $FALSE$ otherwise

```
OpenRect1::contains(Point closeShape) {
    dir1 = Direction(closeShape − endp1);
    dir2 = Direction(endp2 − endp1);
    if (dir.isBetween(dir1, dir2)) {
        dir3 = Direction(closeShape − endp2);
        if dir3.isBetween(dir,dir2.reverse())
            return TRUE;
    }
    return FALSE;
};
OpenRect1::contains(Polygon closeShape){
    opline1a = OpenLine1(endp1, dir);
    opline1b = OpenLine1(endp2, dir);
    aLine = Line(endp1, endp2);
    for ep ∈ endpoints of closeShape
        if !(endp1 == ep || endp2 == ep || contains(ep) ||
                opline1a.crosses(ep) || opline1b.crosses(ep) || aLine.crosses(ep))
            return FALSE;
    return TRUE;
};
```

Algorithm 8 OpenRect2 contains Point/Line/Polygon

Input: closeShape is a topologically closed object;
the current OpenRect2 object, which has attributes $(startPoint, dir1, dir2)$;
Output: $TRUE$ if contain, $FALSE$ otherwise

```
OpenRect2::contains(Point closeShape) {
    dir = Direction(closeShape − startPoint);
    if (dir.isBetween(dir1, dir2))
        return TRUE;
    return FALSE;
};

OpenRect2::contains(Polygon closeShape){
    opline1a = OpenLine1(startPoint, dir1);
    opline1b = OpenLine1(startPoint, dir2);
    for ep ∈ endpoints of closeShape
        if !(ep == startPoint || contains(ep) || opline1a.crosses(ep) || opline1b.crosses(ep))
            return FALSE;
    return TRUE;
};
```

Chapter 4

Consistency Checking for Euclidean Spatial Constraints: A Dimension Graph Approach

4.1 Introduction

A spatial database [28, 35, 75] management system aims at the effective and efficient management of data related to a space such as the physical world (geography, urban planning, astronomy); parts of living organisms (anatomy of the human body); engineering design (very large scale integrated circuits, the design of an automobile or the molecular structure of a pharmaceutical drug); and conceptual information space (a multi-dimensional decision support system, fluid flow, or an electro-magnetic field). The distinguishing features of a spatial database management system are the use of complex data types like points, lines and polygons to represent spatial objects and the existence of many potential relationships between spatial objects.

Consistency checking is an important concept to maintain the integrity of databases in general and spatial databases in particular[17]. Consistency checking is the process of identifying contradictory information in a database. For example, if A, B and C are three spatial objects and if B is west of A and C is west of B and if the database indicates that C is east of A then the information is inconsistent. The existence of many potential spatial relationships implies that consistency checking in the context of spatial

databases is more challenging *vis-a-vis* its traditional relational counterpart.

Consistency checking can also be used for spatial query processing via semantic query optimization[59] and reasoning for spatial qualitative relationships. For example, given a spatial query S and a set of of spatial constraints SC, if the query S is inconsistent with respect to SC then the answer to S is null. Currently most of consistency checking is based on Allens's algorithm [4] which was originally devised for checking temporal relationships on one-dimensional objects. Spatial relationships are typically formulated among multi-dimensional objects and straightforward extension of Allen algorithm for spatial relationships are prohibitively expensive.

In this chapter, we address the problem of consistency checking for directional spatial constraints in the two-dimensional Euclidean space. We propose a dimension graph representation for maintaining the spatial constraints among objects. Basically, the spatial constraints is projected on each dimension(X and Y), the constraints that must be satisfied for each dimension(X/Y) are maintained in different graph(X/Y graph). One graph records the constraints on one dimension. The problem of constraint consistency checking is transformed to a graph cycle detection problem on the dimension graph. The cycle detection could be solved by traversing the graph in linear time. As we will see in later sections, the proposed consistency checking algorithm is efficient in terms of both time and space. The algorithm also guarantees the global consistency of system.

4.1.1 Spatial Data Model

Recent reports[28, 35, 75, 2] have described the accomplishments of spatial database research and have prioritized research needs. A broad survey of spatial database requirements and an overview of research results is provided by [75, 28, 2, 60].

A spatial data model is defined by geometric entities, spatial operations on these entities and spatial relationships between them. The three basic geometric entities are point, line and polygon which represent spatial objects in zero, one and two dimension respectively. To facilitate rapid query processing polygons are often represented by their Minimum Bounding Rectangles(MBRs). An MBR of a spatial object is the smallest axis

parallel rectangle which covers the polygon.

The spatial relationships between the geometric entities are categorized according to the mathematical properties of the relationships. For example, topological relationships like *overlap* and *adjacent* are relationships which are invariant under a change of coordinate or projection system. Other types of relationships include *Metric*, *Set* and *Directional*. Some important relationships categories and their representative example are shown in 4.1. In this chapter we will focus on consistency checking of absolute directional relationships like North, South, East, West, Northeast, Northwest, Southeast and Southwest. These relationships can be defined between any combination of point, line and polygon objects but we will restrict our attention to homogeneous point, line and MBR pairs. For constraints among intervals, we deal with Allen's 13 relationships [4]. The constraints are represented in terms of disjunctions and/or conjunctions of spatial predicates. The predicates are logically atomic.

Data model	Operator Group	Operation
Vector Object	Set-Oriented	equals, is a member of, is empty, is a subset of, is disjoint from, intersection, union, difference, cardinality
	Topological	boundary, interior, closure, meets, overlaps, is inside, covers, connected, components, extremes, is within
	Metric	distance, bearing/angle, length, area, perimeter.
	Direction	east, north, left, above, between.
	Network	successors, ancestors, connected, shortest-path

Table 4.1: A Sample of Spatial Operations

4.1.2 Problem Definition

In this chapter, we intend to explore the consistency checking for Euclidean qualitative spatial constraints among points, intervals, and 2-D Minimum Bounding Rectangles(MBRs) objects in spatial database.

Consistency Checking Problem:

Given: A collection of 0-th order[1] spatial constraints in terms of disjunctions and/or conjunctions

of spatial predicates

Find: Consistency, i.e., return TRUE if the constraints are consistent, False otherwise

Objective: Reduce computation complexity

Constraint: (a) 2-D Euclidean space.

(b) Two-dimensional extended objects are approximated by MBRs

(c) Spatial objects refereed to given spatial constraints are of homogeneous types,

(i.e., point pairs, interval pairs, or MBRs in 2D)

Consider an example of direction constraints among point objects A, B, and C. Assume one constraint says A is Southeast of B, B is Northwest of C, and A is Northeast of C. This constraint is consistent, and hence the result will be TRUE. An example of the possible spatial configurations satisfying this constraint is shown in Figure 4.1.

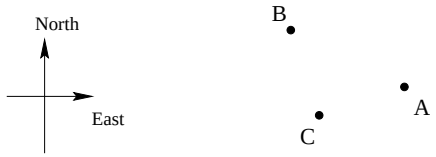


Figure 4.1: One possible spatial configuration for point objects A, B, and C

Suppose we have another set of constraint among A, B, and C: A is strictly north of B, B is Northwest of C, and A is Northeast of C. This constraint is inconsistent since there does not exist any spatial configuration of A, B, and C satisfying this constraint. The consistency checking algorithm should return FALSE.

[1] 0-th order spatial constraints means that there are no free variables but only constant objects

4.1.3 Related work and our contributions

Most of previous study on consistency checking is based on Allen's consistency checking algorithm [4] for constraints among intervals. The basic approach for consistency checking is to use transitive closure algorithm, which incurs high order of time and space complexity. Hernandez [30] presented mechanisms to maintain the consistency of a knowledge base of spatial information based on a qualitative representation of 2-D positions. His approach improved Allen's algorithm by using the heuristic of rich structure of the spatial domain. Bowman and etc.[8, 7] addressed the problem of consistency checking between multiple viewpoints using strategies based on unification. Some other work[37, 19, 45] focused on the problem of consistency checking for more general constraints, such as, 1-th order constraints. Manandhar[44] discussed deterministic consistency checking for LP constraints. Beneventano and etc.[6] focused on the problem of providing a theoretical framework for consistency checking of integrity constraints in a complex object database environment.

The consistency checking algorithm is dependent on the set of spatial predicates. In this chapter, we deal with basic direction predicates for absolute directions[65]. There are some other directional predicates. The research work on direction modeling has been carried out in several areas such as geographic information systems and image analysis. Most of the studies is on how to capture the semantics of direction relations, and further, how to do spatial reasoning based on direction predicates [15, 18, 20]. There are two major direction reference frames used to model direction in 2D space: the cone-based model[51], and the projection-based model[18, 20]. Frank[3] compared these two models and found the projection-based reference frame to be better in many aspects. The most common way to model directions between extended objects is through the object's Minimum Bounding Rectangle(MBR), where direction relations are obtained by applying Allen's [4] interval relations along the x and y axis, in which case, 169 different relations[15] can be distinguished. We'll briefly describe Allen's Algorithm since it is basic to many algorithms.

4.1.3.1 Allen's Propagation Algorithm

Allen[4] summarized thirteen mutually exclusive relationships to express any possible relationships between intervals as shown in table 4.2.

Relationships	symbol	Symbol for Inverse	Pictorial Example
X before Y	<	>	XXX YYY
X equal Y	=	=	XXX YYY
X meets Y	m	mi	XXXYYY
X overlaps Y	o	oi	XXX YYY
X during Y	d	di	XXX YYYYYY
X starts Y	s	si	XXX YYYYY
X finishes Y	f	fi	XXX YYYYY

Table 4.2: Thirteen possible relationships proposed by Allen

In Allen's work[4], the relationships between intervals are maintained in a network where each node N_i represents individual interval i, and each arc N(i,j) is associated with possible relationships between the corresponding interval pair i and j. The basic algorithm he used for maintaining relationships was propagating new relationships by computing the transitive closure of the relationships between intervals.

Figure 4.2 shows the network used by Allen for consistency checking. 4.2(a) is the network for two inputs, i.e. *S overlaps or meets L*, and *S is before, meets, is metby, or after R*. After the second input was added, the algorithm computed the constraint

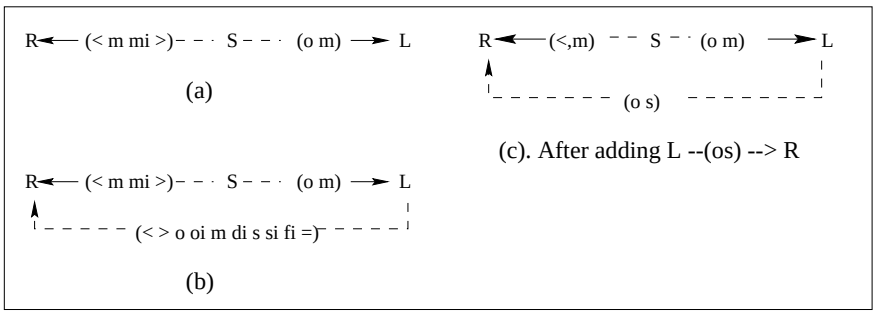


Figure 4.2: Examples of Allen's algorithm

between L and R, and the resulting network is shown as 4.2(b). If we add a new fact *L overlaps, starts, or is during R*, we need to propagate its effect through the network, and thus obtaining the resulting network 4.2(c).

As explained in Allen's paper[4], the time complexity of this algorithm is calculated as: $13 \times \frac{(N-1)(N-2)}{2}$ for N intervals, i.e., $O(N^2)$. The space requirement for the algorithm is also $O(N^2)$.

As Allen stated in [4], one problem with this algorithm is that it does not detect all inconsistencies in its input. Quote the phrases in [4] "In fact, it only guarantees consistency between three node subnetworks. There are networks that can be added which appear consistent by viewing any three nodes, but for which there is no consistent overall labeling of the network." In other words, the algorithm can not guarantee global consistency.

4.1.3.2 Our Approach

In this chapter, we propose a new strategy to process consistency checking for Euclidean spatial constraints among objects. We use a geometric approach by incorporating the spatial domain information in consistency checking. We propose dimension graphs to maintain the spatial constraints among objects. Each conjunctive constraint is projected on both dimensions, and a dimension graph is constructed for the constraint on each dimension. For spatial constraints in general format, the constraints can be converted to its Disjunctive Normal Form(DNF)[56] , and dimension graphs are constructed for each conjunction. By using the dimension graph representation, the problem of constraint consistency checking is then transformed to a graph cycle detection problem on each dimension graph. The cycle detection could be solved efficiently with O(N+E) time as well as space complexity, where N is the number of spatial objects, and E is the number of spatial predicates in the constraint. Recall that Allen's algorithm has a time and space complexity of $O(N^2)$. The proposed approach to consistency checking for spatial constraints is faster when the number of predicates is much smaller than N^2 and there are few disjunctions in the spatial constraint. The dimension graph and consistency

checking algorithm can be used for points, and MBRs in 2 dimensional space. Since the algorithm returns TRUE if and only if the dimension graph of at least one conjunction contains no cycle, which means there exist at least one consistent overall constraint. The algorithm can guarantee the global consistency.

4.1.4 Scope and Outline

In this chapter, we address the problem of consistency checking for spatial constraints in the two-dimensional Euclidean space. We deal with only 0-th order constraints. We focus on the qualitative constraints among objects, which include topological and direction relationships(See Table 4.1). Distance-based constraints are not discussed. We consider the constraints among point objects, interval objects, and region objects approximated by MBRs. We focus on addressing the consistency checking for homogeneous types of objects, i.e., the constraints among point objects, or intervals, or MBRs. The consistency checking for the constraints specified among mixed types of objects(e.g. the constraints between a point and an interval) is out of the scope of the chapter, which may be addressed in the future work. We only discuss a specific set of predicates defined in the chapter. The predicates are defined in Euclidean space. Different predicate sets or in different space may be different. Some constraints that are inconsistent in Euclidean space may be consistent in spherical space. The consistency checking for predicates in other space is out of the scope of this chapter.

The organization of this chapter is as follows: In section 2, we propose dimension graph representation for the conjunctive constraints among points, intervals. The consistency checking for conjunctive constraint based on the dimension graph representation is introduced in section 3. In section 4, we discussed the dimension graph construction and consistency checking for conjunctive constraints among MBRs. Finally, the consistency checking for constraints in general format is described in section 5. The chapter ends with conclusions and recommendations for future work.

4.2 Dimension Graphs for Conjunctive Spatial Constraints

In this section, we will describe the construction of dimension graph for conjunctive spatial constraints among points and intervals. The basic idea is to projected the conjunctive spatial constraint onto each dimension and record the spatial constraints that much be satisfied on each dimension in a dimension graph respectively. The dimension graph for points in 2D space contains X-graph and Y-graph. The dimension graph for intervals in 1D space contains only one graph. We also analyze the computation complexity for the dimension graph construction algorithm.

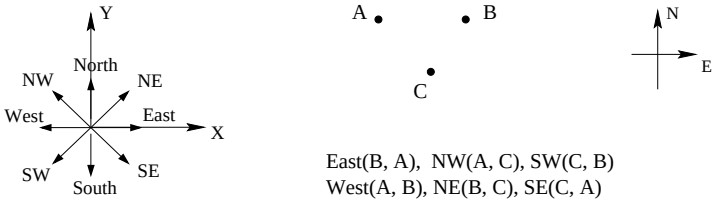
4.2.1 Dimension Graphs for Constraints Among Point Objects

We start by defining a set of absolute direction predicates for point objects in terms of coordinates. Here, We assume the global coordinate systems aligned with the reference frame of absolute directions, i.e., $North$ aligns with y-axis, and $East$ align with x-axis. The definition for each predicate is given in Table 4.3. The first column of the

Direction predicates	A_x, B_x	A_y, B_y
$SP(A,B)$	$=$	$=$
$North(A,B)$	$=$	$>$
$South(A,B)$	$=$	$<$
$East(A,B)$	$>$	$=$
$West(A,B)$	$<$	$=$
$NE(A,B)$	$>$	$>$
$NW(A,B)$	$<$	$>$
$SE(A,B)$	$>$	$<$
$SW(A,B)$	$<$	$<$

Table 4.3: Direction Predicates for point objects in terms of coordinates

table enumerates the direction predicates. The second and third columns represent the relationships between two point objects on X and Y dimensions respectively. A_y, B_y are the y-components of A and B, and A_x, B_x are x-components of A and B. Figure 4.3(a) illustrates the intuition of the definition. The predicates are defined using direction equivalence classes [65] by partitioning the space. SP(A,B) means A and B are on the same position, $North$, $South$, $East$, and $West$ represent exact directions, while NE, NW, SE, and SW can point to any direction in their respective quadrants. This



(a) Definitions of the predicates

(b) Examples of predicates

Figure 4.3: Illustration of the predicates

predicate set is complete. Figure 4.3(b) shows examples of predicates described the directional relationships among points A, B and C, i.e., B is east of A, A is northwest of C, and C is southwest of B.

The spatial constraints represented in terms of conjunctions of predicates can be maintained in two graphs: X-graph and Y-graph. The nodes in both graphs represent the objects forming the constraints. The direction constraints are represented as directed edges in each graph according to the symbol in the column 2 and 3 of Table 4.3. The edge goes from the node with smaller values to the node with larger values. If the symbol is '=', the two nodes are merged to one node. Figure 4.4 shows the graph representation

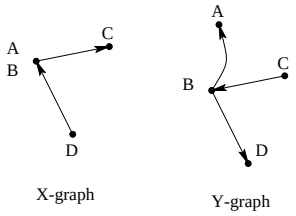


Figure 4.4: Dimension graph for North(A,B) $\wedge$ NW(B,C) $\wedge$ SE(B,D)

for the constraint North(A,B) $\wedge$ NW(B,C) $\wedge$ SE(B,D). The dimension graph is a union of X-graph and Y-graph, and is constructed according to the definition of each predicate in Table 4.3. Algorithm 9 is the pseudo-code of the graph constructing procedure. The input of the algorithm is the conjunctive constraint and the output is the dimension graph. For each predicate in the conjunctionConstraint, the algorithm invokes the sub-function **add_a_predicate_point** to add the predicate to the dimension graph. This

Algorithm 9 Constructing Dimension Graph from conjunctive spatial constraints for points: **constructGraphPoint**

```
Input: conjunctionConstraint is a set of conjunctive predicates
Output: constraintGraph consists of the corresponding X/Y-graphs.

Graph constructGraphPoint(Set of Predicates conjunctionConstraint) {
      Graph constraintGraph = ∅;
      for each entry p ∈ conjunctionConstraint
            add_a_predicate_point(p, &constraintGraph);
      return constraintGraph;
}

add_a_predicate_point(Predicate aPredicate, Graph* aGraph ) {
      firstObject = getFirstObject(p);
      secondObject = getSecondObject(p);
      addNode(firstObject, secondObject, aGraph.graphX);
      symbol =findXconstraint(Table 4.3, p);
      addEdge(symbol, firstObject, secondObject, aGraph.graphX);
      addNode(firstObject, secondObject, aGraph.graphY);
      symbol =findYconstraint(Table 4.3, p);
      addEdge(symbol, firstObject, secondObject, aGraph.graphY);
}

addNode(Node n1, Node n2, Graph aGraph) {
      if n1 ∉ aGraph          Add n1 to aGraph;
      if n2 ∉ aGraph          add n2 to aGraph;
}

addEdge(char symbol, Node n1, Node n2, Graph, aGraph) {
      if (symbol == '=')
            merge nodes n1, n2 to one;
      else if (symbol == '<')
            add directed edge of (n1, n2) to aGraph;
      else
            add directed edge of (n1, n2) to aGraph;
}
```

function calls function *addNode* to add nodes that do not in the dimension graph into the graph, and calls function *addEdge* to add the spatial relationships between nodes into dimension graph.

We can easily summarize the computation complexity for this algorithm. Let N be the number of spatial objects involved and E be the number of spatial predicates in the conjunction.

- Time complexity = O(N+E);

  Any of the sub-functions **addNode** and **addEdge** takes constant time(O(1)).

findXconstraint and **findYconstraint** are essentially table lookup functions, which could also be accomplished in constant time. The whole algorithm therefore have the time bound of O(E). The generated graph has at most N nodes and E edges each in X-graph and Y-graph.

- Space complexity O(N+E).

  We can process on each dimension graph at a time, the space requirement is also linear to the graph elements.

4.2.2 Dimension Graph for Conjunctive Spatial Constraints Among Intervals

The relationships between intervals proposed by Allen [4] can be defined in terms of the endpoints of intervals. Table 4.4 shows the definition of the 13 relationships, where A_1, A_2 and B_1, B_2 represent the start and end points of the intervals A and B respectively. Allen's symbols are used here.

Predicate name	predicates	point relationships
before	$<(A,B)$	$A_2 < B_1$
equal	$=(A,B)$	$(A_1 = B_1) \wedge (A_2 = B_2)$
overlaps	$o(A,B)$	$(A_1 < B_1) \wedge (A_2 > B_1) \wedge (A_2 < B_2)$
meets	$m(A,B)$	$(A_2 = B_1)$
during	$d(A,B)$	$(A_1 > B_1) \wedge (A_2 < B_2)$
starts	$s(A,B)$	$(A_1 = B_1) \wedge (A_2 < B_2)$
finishes	$f(A,B)$	$(A_1 > B_1) \wedge (A_2 = B_2)$
after	$bi(A,B)$	$B_2 < A_1$
overlapby	$oi(A,B)$	$(B_1 < A_1) \wedge (B_2 > A_1) \wedge (B_2 < A_2)$
metby	$mi(A,B)$	$(B_2 = A_1)$
duringby	$di(A,B)$	$(B_1 > A_1) \wedge (B_2 < A_2)$
startby	$si(A,B)$	$(B_1 = A_1) \wedge (B_2 < A_2)$
finishedby	$fi(A,B)$	$(B_1 > A_1) \wedge (A_2 = B_2)$

Table 4.4: Spatial relationships for intervals, where $<$ and bi describe directional relationships and others are topological relationships

The relationships represented in terms of conjunctions of predicates can be maintained in directed graphs, where the nodes represent start or end points of the individual intervals, and the directed edges represent the constraints between the two points. Each edge is added to the graph according to the definition in Table 4.4. The edge point to the

nodes with larger values from the nodes with smaller values. The nodes with same values are merged into one node. It is worth noting that there is an intrinsic constraint between the start point and the end point of an individual interval, i.e. start-point < end-point. Figure 4.5 shows the graph representation for the constraint $before(S, R) \wedge meets(S, L)$.

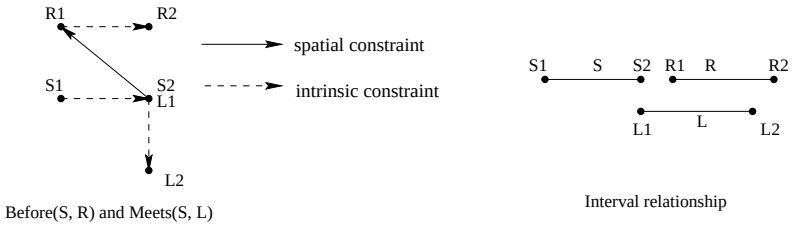


Figure 4.5: $before(S, R) \wedge meets(S, L)$

The graph is constructed according to the definition of each predicate in Table 4.3. The dashed arrow represents the intrinsic constraint of the start point and end points. The pseudo-code of the graph constructing procedure is given as in Algorithm 10.

Algorithm 10 Constructing Graph from conjunctive constraints for intervals: **constructGraphInterval**

```
Input: conjunctionConstraint is a set of conjunctive predicates
Output: constraintGraph is the corresponding graph

Graph constructGraphInterval(Set of Predicates conjunctionConstraint) {
    constraintGraph = ∅;
    for each entry p ∈ conjunctionConstraint
        add_a_predicate_interval(p, &constraintGraph);
    return constraintGraph;
}

add_a_predicate_interval(Predicate p Graph* aGraph ) {
    firstObject = getFirstObject(p);
    secondObject = getSecondObject(p);
    addIntervalNode(firstObject, secondObject, constraintGraph);
    pointPredicates = convertToPoint(p); //according to Table 4.4
    for each r(k, l) in pointPredicates {
        if (k < l) add directed edge (k, l) to aGraph;
        if (k > l) add directed edge (l, k) to aGraph;
        if (k = l) merge node k and l in aGraph;
    }
}
```

The input of the algorithm is the conjunctive constraint and the output is the corresponding dimension graph. For each predicate in the conjunctionConstraint, the al-

gorithm calls the sub-function **add_a_predicate_interval** to add the predicate to the dimension graph. This function adds nodes that do not in the dimension graph into graph, and adds the spatial relationships between nodes into dimension graph.

Let N be the number of objects(intervals) and E be the number of interval predicates in conjunctionConstraint. We can summarize the complexity as follows:

- Time complexity = O(E);

  According to table 4.4, there are at most three point predicates should be satisfied for each interval predicate, and hence at most three edges added for each interval predicate. The time complexity for this algorithm is therefore O(E).

- Space complexity O(N+E).

  The resulting constraintGraph consists of at most 2N nodes and 3E edges. The space requirement is linear to the number of nodes and edges, roughly 2N+3E, which is essentially O(N+E).

4.3 Consistency checking for conjunctive constraints

In the previous section, we construct dimension graph for conjunctive Euclidean spatial constraints among point objects or intervals. In this section, we will describe the consistency checking algorithm based on the dimension graph representations.

Let's revisit the example given in Figure 4.4. If we add a new constraint $NE(A, C)$, the new constraint graph is given in Figure 4.6. A new edge represented by dot line is

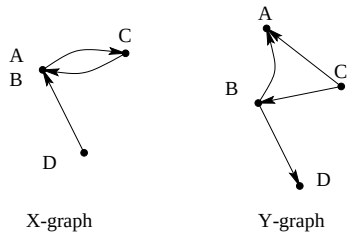


Figure 4.6: North(A,B) $\wedge$ NW(B,C) $\wedge$ SE(B,D)$\wedge$ NE(A,C)

added into the graph. As can be seen, a cycle is constructed in the X dimension graph in Figure 4.6. In other words, the constraint of $North(A, B) \wedge NW(B, C) \wedge SE(B, D) \wedge NE(A, C)$ requires that the x-value of A is smaller than the x-value of C, and at the same time, the x-value of C is smaller than the x-value of A. This is a contradiction, which means the constraint is inconsistent. In general, we can characterize this feature as Theorem 3.

Theorem 3 *A conjunctive constraint is consistent if and only if there exists no cycle in its corresponding dimension graphs, i.e., the graphs are all directed acyclic graphs(DAG).*

Proof:

$\Longrightarrow$: Suppose that the dimension graph contains no cycle, i.e., the graph is a directed acyclic graph(dag), we can construct a topological sort of the dag [13] using depth-first search algorithm. A topological sort of a dag is a linear ordering of all its nodes such that if the dag contains an edge (u, v), then u appears before v in the ordering. In other words, there exists at least one spatial configuration satisfying the constraints, the constraint is consistent.

$\Longleftarrow$: Suppose that the dimension graph contains a cycle. For the nodes involved in the cycle, there exist no topological sort, i.e., no linear ordering of the nodes is possible. In other words, we can not give a global labeling for the nodes to meet the requirement that all directed edges go from left to right. No spatial configuration can satisfy the constrain. Therefore, the constraint is inconsistent. ■

4.3.1 Basic Algorithm

We now describe the algorithm for consistency checking for conjunctive spatial constraints among a set of points or a set of intervals based on dimension graph representation. The consistency checking for conjunctive constraints contains two steps: 1. Construct the corresponding dimension graph; 2. Perform cycle detection on each graph. The constraint is consistent if none of the graph contains cycle. The pseudo-code is described as in Algorithm 11.

Algorithm 11 Consistency checking: **conjunctionConsistencyCheck**

```
Input: conjunctionConstraint is a set of conjunctive predicates
Output: TRUE if consistent, FALSE otherwise

conjunctionConsistencyCheck(Set of Predicates conjunctionConstraint) {
    Graph constraintGraph = ∅;
    if the constraint is among points
        constraintGraph = constructGraphPoint (conjunctionConstraint);
    else //the constraint is among intervals
        constraintGraph = constructGraphInterval (conjunctionConstraint);
    if !detectCycle(constraintGraph)
        return TRUE;
    return FALSE;
}
```

The algorithm first constructs the dimension graph by invoking subfunction **constructGraphPoint** or **constructGraphInterval** according to the types of the objects. The function **detectCycle** performs cycle detection on the corresponding dimension graph. The algorithm return TRUE if no cycle is detected.

A nice property of this algorithm is its efficiency. The consistency checking is just a graph cycle detection which can easily be done in linear time. As in previous section, let

N be the number of spatial objects and E be the number of spatial predicates involved in the conjunctive constraint.

- Time complexity = O(N+E);
  O(N+E) is the time for cycle detection in a directed graph with N nodes and E edges[13]. As we explained in section 2, each of X-graph and Y-graph for a set of points has at most N nodes and E edges, and the dimension graph for intervals has at most 2N nodes and 3E edges. Therefore, O(2N+3E) is the upper bound time complexity for consistency checking for a conjunctive constraint, which is same as O(N+E).

- Space complexity =(N+E).
  We can process on each dimension graph at a time, the space requirement is also linear to the graph elements.

4.3.2 Examples

Consider the example of spatial constraints among intervals:

(S meets L) and (S is metby R).

The consistency checking for this constraint is then accomplished by two steps. The function **constructGraphInterval** is first invoked to construct the corresponding dimension graph, which is illustrated in Figure 4.7 (a). The **detectCycle** function checks

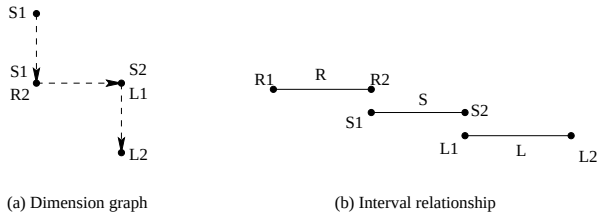


Figure 4.7: Dimension Graph for $meets(S, L) \wedge metby(S, R)$

the cycle in the graph. Since there is no cycle found, the constraint is consistent. We can construct an interval configuration which satisfies the constraint as in Figure 4.7 (b).

Assume a new fact $overlaps(L, R)$ is added into the system. We call the function of

add_a_predicate_interval to add this new constraint to the dimension graph, resulting in a new dimension graph as in Figure 4.8.

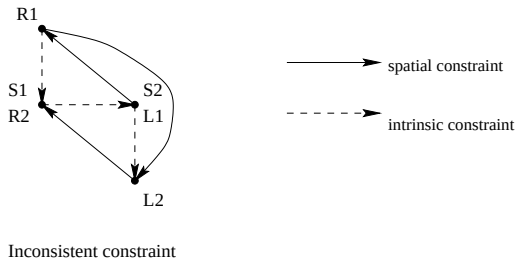


Figure 4.8: Dimension Graph after adding overlaps(L,R)

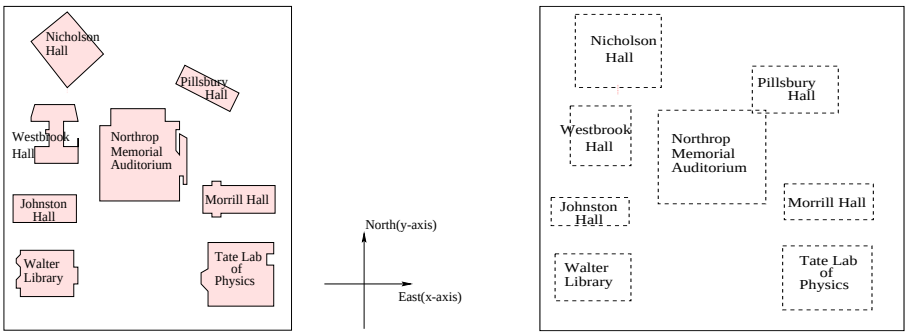
The cycle detection is performed on the new dimension graph to check consistency of the status. Since cycles of (R1, S1, L1, R1) and (R2, L1, L2, R2) are detected in the figure as shown. The new constraint added is not consistent with the system, therefore, the new constraint is inconsistent. There exist no interval configuration satisfying this constraint.

Since the consistency checking is performed based on the cycle detection of the corresponding dimension graph, the algorithm can always detect inconsistent constraints. As we stated in Theorem 3, for any dimension graph without cycle, there is a consistent configuration among objects. The algorithm can guarantee global consistency.

4.4 Consistency Checking for Conjunctive Constraints Among MBRs

In the previous section, we discuss the dimension graphs for conjunctive constraints among 2D points and 1D intervals, and also describe the dimension graph based consistency checking algorithm. In this section, we will extend our dimension graph based approach to 2D spatial objects approximated by MBRs. Examples of 2D spatial objects includes polygon regions.

It is common in spatial databases to approximate 2-D regions by minimum bounding rectangles(MBRs) which are orthogonal with respect to the global coordinate system. Figure 4.9(a) shows a small portion of the campus maps of University of Minnesota. Figure 4.9(b) replaces all the buildings in Figure 4.9(a) by their corresponding $MBRs$.



(a) Campus map of University of Minnesota

(b) Approximate buildings by MBRs

Figure 4.9: 2D region objects vs. MBRs

By using MBR approximation, we can use two representative points, namely lower-left and upper-right corners to determine the corresponding object. In the rest of the chapter, we use the notation of A_{ll} and A_{ur} to represent the lower-left and upper-right corners of the MBR for any object A. The notations of $A_{ll.x}, A_{ll.y}, A_{ur.x}$ and $A_{ur.y}$ are used to represent the x and y coordinates for lower-left and upper-right corners of MBR A.

The direction relationship between MBRs can be determined by the relationships between the representative points. Table 4.5 shows the definitions of the direction predicates based on the representative points of MBR of the objects. The first column of the

Direction predicates	conditions
$SP(A,B)$	$(A_{ll.x} = B_{ll.x}) \wedge (A_{ll.y} = B_{ll.y}) \wedge (A_{ur.x} = B_{ur.x}) \wedge (A_{ur.y} = B_{ur.y})$
$North(A,B)$	$(A_{ll.y} \geq B_{ur.y}) \wedge (A_{ll.x} \geq B_{ll.x}) \wedge (A_{ur.x} \leq B_{ur.x})$
$South(A,B)$	$(A_{ur.y} \leq B_{ll.y}) \wedge (A_{ll.x} \geq B_{ll.x}) \wedge (A_{ur.x} \leq B_{ur.x})$
$East(A,B)$	$(A_{ll.x} \geq B_{ur.x}) \wedge (A_{ll.y} \geq B_{ll.y}) \wedge (A_{ur.y} \geq B_{ur.y})$
$West(A,B)$	$(A_{ur.x} \leq B_{ll.x}) \wedge (A_{ll.y} \geq B_{ll.y}) \wedge (A_{ur.y} \geq B_{ur.y})$
$NE(A,B)$	$(A_{ll.x} \geq B_{ur.x}) \wedge (A_{ll.y} \geq B_{ur.y})$
$SE(A,B)$	$(A_{ll.x} \geq B_{ur.x}) \wedge (A_{ur.y} \leq B_{ll.y})$
$NW(A,B)$	$(A_{ur.x} \leq B_{ll.x}) \wedge (A_{ll.y} \geq B_{ur.y})$
$SW(A,B)$	$(A_{ur.x} \leq B_{ll.x}) \wedge (A_{ur.y} \leq B_{ll.y})$

Table 4.5: Direction Predicates for MBR

table enumerates the direction predicates. The second column gives the constraints that should be satisfied by their representative points.

For example, in Figure 4.9(b), MBR of " Morrill Hall" is north of MBR of "Tate Lab", and the MBR of "Pillsbury hall" is northeast of the MBR of "Northrop Auditorium". The same relationships hold for buildings in Figure 4.9(a). These are described by predicates North(Morrill Hall, Tate Lab), and NE(Pillsbury hall, Northrop Auditorium).

4.4.1 Dimension Graphs for Conjunctive Constraints Among MBRs

The dimension graph for each direction predicate contains X-graph and Y-graph which maintain the constraints that must be satisfied on X and Y dimension respectively. The nodes in these graphs represent the objects forming the constraints. The constraints are represented as directed edges according to the conditions specified in column 2 of Table 4.5. Similar to the X-graph and Y-graph for points, an directed edge goes from the node with smaller values to the node with larger values. If the value associated with two nodes are same, the two nodes are merged to one node. We introduce a new '**thick arrow**' edge here to represent new relationships of $\leq$ or $\geq$. If the relationship between two nodes p and q is $p \leq q$, a 'thick arrow' directed edge is added from p to q. If the relationship between p and q is $p \geq q$, a 'thick arrow' directed edge is added from q to

p. The dashed arrow is used for intrinsic constraint between the lower-left point and the upper-right point of a MBR.

The dimension graph for a conjunctive constraint is constructed by adding the constraints specified in each predicate in X-graph and Y-graph. Figure 4.10(a) and (b) shows an example of dimension graph for a conjunctive constraint of $North(A, B) \wedge NE(B, C) \wedge SW(C, A)$. Figure 4.10(c) is a possible spatial configuration among MBR A, B and C.

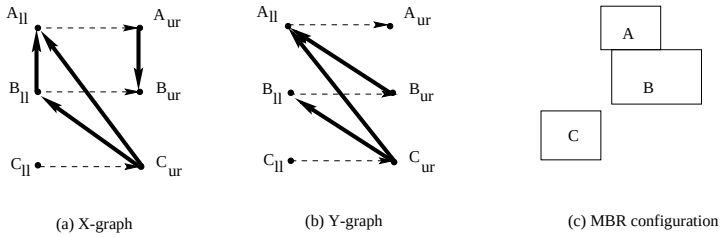


Figure 4.10: Example MBR configuration and its corresponding dimension graphs, thick arrow represents $\leq$, thin arrow represents $<$

We now describe the dimension graph construction for conjunctive constraints for MBRs. The pseudo-code of the graph constructing procedure is given as in Algorithm 12. For each predicate in the conjunctionConstraint, the algorithm invokes the subfunction **add_a_predicate_MBR** to add it to the dimension graph. The basic strategy of this function is first to transform the MBR predicate to point predicate set, and add each point predicate into the graph. To accomplish this, *addNode* and *addEdge* are invoked.

Let N be the number of objects(i.e. MBRs) and E be the number of predicates in conjunctionConstraint. We can summarize the complexity as follows:

- Time complexity = O(E);

  According to table 4.5, there are at most four point predicates should be satisfied for each MBR direction predicate, and hence at most four edges added for each predicate. The time complexity for this algorithm is therefore O(E).

- Space complexity = O(N+E).

Algorithm 12 Constructing Graph for MBR constraints: **constructGraphMbr**

Input: *ConjunctionConstraint* is a set of conjunctive predicates between MBRs
Output: *constraintGraph* consists of the corresponding X/Y-graphs.

```
Graph constructGraphMbr(Set of Predicates conjunctionConstraint) {
    Graph constraintGraph = ∅;
    for each entry p ∈ conjunctionConstraint
        add_a_predicate_MBR(p, &constraintGraph);
    return constraintGraph;
}

add_a_predicate_MBR(Predicate aPredicate, Graph* aGraph ) {
    pointPredicates= findfromTable(aPredicate);
    for each predicate r(k,l) ∈ pointPredicates
    addNode(k, l, aGraph.graphX);
    addEdge(r(k,l), aGraph.graphX);
    addNode(k, l, aGraph.graphY);
    addEdge(r(k,l), aGraph.graphY);
}

addEdge(Point Predicate pp, Graph* aGraph ) {
    r(k,l)= getRelationSymbol(pp);
    if (r ==′<′) add directed edge (k,l) to aGraph;
    if (r ==′>′) add directed edge (l,k) to aGraph;
    if (r ==′=′) merge node k and l in aGraph;
    if (r ==′≤′) add thick directed edge (k,l) to aGraph;
    if (r ==′≥′) add thick directed edge (l,k) to aGraph;
}
```

The resulting constraintGraph consists of at most 2N nodes and 4E edges. The space requirement is linear to the number of nodes and edges, roughly 2N+4E, which is O(N+E).

4.4.2 Consistency checking for conjunctive constraints Among MBRs

After constructing dimension graphs for conjunctive constraints among MBRs, we can check the consistency by detecting cycle in the dimension graphs. There are three possible situations and corresponding results:

Case 1: There exist no cycle (constraint predicates are consistent)

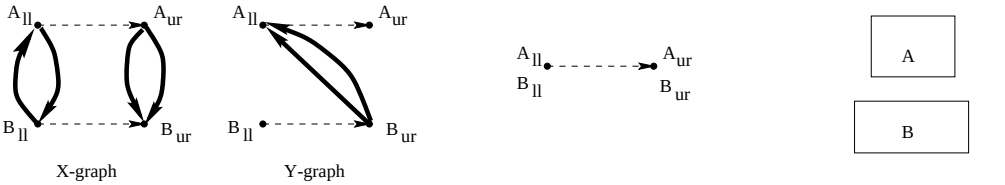
Case 2: Every cycle consists of only thick edge (constraint predicates are consistent)

Case 3: At least one cycle consists of non-thick edges(constraint predicates are incon-

sistent)

Case 1 is obvious. No cycle means there exist a consistent spatial configuration among all objects. Figure 4.10 is an example of consistent constraints whose dimension graphs contain no cycle.

In case 2, since every cycle detected contains only thick edges. Recall that any thick edge $p \rightarrow q$ represents that the relationships between p and q is '$\leq$', i.e., either $p = q$ or $p < q$ could hold. If we label all thick edges as '=', all the nodes involved in the cycle will have same value. We can then merge this cycle with a big node consisting of all nodes involved in the cycle. The transformed graph contains no cycle, and the corresponding constraint is consistent. Figure 4.11 shows such an example. Figure 4.11 (a) is the



(a) Dimension graph (b) X-graph after merging the cycle (c) MBR configuration

Figure 4.11: Dimension graphs for $North(A, B) \wedge South(B, A)$

dimension graph for constraint $North(A, B) \wedge South(B, A)$, it contains a cycle with only thick edges in X-graph. Figure 4.11 (b) is the transformed X-graph after merging each cycle to one node. There is no cycle in this resulting graph. This constraint is obviously consistent. Figure 4.11 (c) is a sample configuration.

Case 3 is easy to understand. If there is a cycle containing at least one non-thick edges, no matter how we label the thick edges in the cycle, we can not remove the cycle. There must be conflicts among coordinates of the objects involved in the cycle, and hence, the constraint is inconsistent. Figure 4.12 shows an example of the constraints among A, B, and C. The constraint is: $North(A, B) \wedge NE(B, C) \wedge SW(A, C)$. As we can noticed, there exist cycles in both X and Y dimension graphs of the constraint. No spatial configuration of A, B, C can satisfy this constraint. The constraint is inconsistent.

Based on the above arguments, we can now describe the consistency checking al-

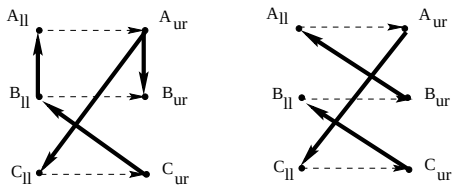


Figure 4.12: Dimension graphs for $North(A, B) \wedge NE(B, C) \wedge SW(A, C)$

gorithm for MBRs as in Algorithm 13. The algorithm first calls function **construct-**

Algorithm 13 Consistency checking for conjunctive MBR predicates: **MBRconjunctionConsistencyCheck**

```
Input: conjunctionConstraint is a set of conjunctive predicates
Output: TRUE if consistent, FALSE otherwise

MBRconjunctionConsistencyCheck(Set of Predicates conjunctionConstraint) {
      Graph constraintGraph = ∅;
      constraintGraph = constructGraphMbr (conjunctionConstraint);
      if !detectCycle(constraintGraph)
            return TRUE;
      else if the cycle contains thin edge
            return FALSE;
      else return TRUE;
}
```

GraphMbr to construct the dimension graph, and then use **detectCycle** to detect cycles in the dimension graph. It returns TRUE if no cycle is detected. If the detected cycle contains thin edges, the algorithm returns FALSE, otherwise, returns TRUE.

Let N be the number of objects(i.e. MBRs) and E be the number of predicates in conjunctionConstraint.

- Time complexity = O(N+E);

  The time complexity for cycle detection is O(N+E). Checking the edge type in a cycle can be done easily by turning on a flag if the traverse passes a thin edge. Therefore, this checking does not require extra time complexity, the time complexity is then O(N+E).

- Space complexity = O(N+E).

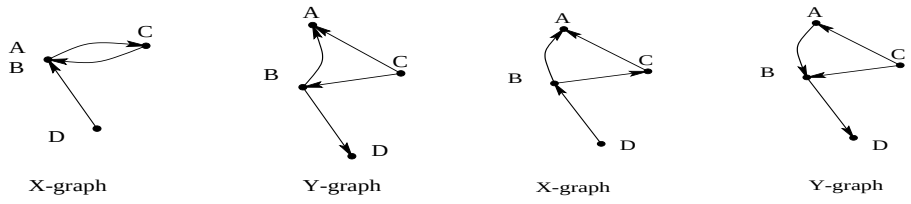
The dimension graph consists of at most 2N nodes and 4E edges. In order to record the edge type, each edge may need an extra flag, which will add another E space. The total space required is roughly 2N+4E+E, which is O(N+E).

4.5 Consistency Checking for Constraints in General Format

In the previous sections, we describe the dimension graph representation and the consistency checking algorithms for conjunctive constraints among points, intervals and MBRs. In this section, we will extend the dimension graph based consistency checking algorithm to the constraints in general format.

4.5.1 Dimension Graphs for Constraints in General Format

The constraints in general format can be transformed into Disjunctive Normal Form(DNF), which is a disjunction of conjunctions where no conjunction contains a disjunction. Each conjunction in the DNF constraint can be represented by a dimension graph by applying the algorithm **constructGraphPoint** or **constructGraphInterval**. The dimension graph of the general format constraint therefore are collection of all the dimension graphs constructed from all its conjunctions. The number of graph sets(X/Y-graphs or interval graph) is the same as the number of conjunctions in the DNF.



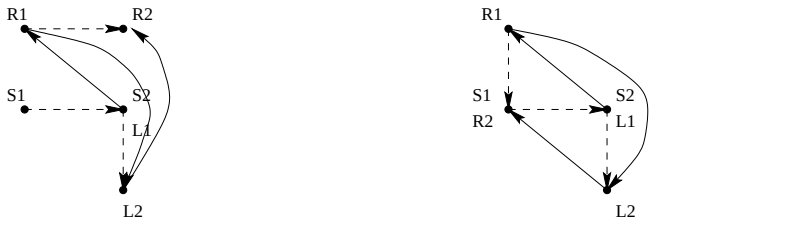
(a) North(A,B) ∧ NW(B,C) ∧ SE(B,D) ∧ NE(A,C)

(b) SE(A,B) ∧ NW(B,C) ∧ SE(B,D)∧ NE(A,C)

Figure 4.13: X/Y-graphs for $(North(A,B) \vee SE(A,B))$ and $NW(B,C)$ and $SE(B,D)$and $NE(A,C)$

Figure 4.13 is an example of the dimension graph maintaining the constraint of $North(A,B) \vee SE(A,B)$ and $NW(B,C)$ and $SE(B,D)$and $NE(A,C)$ for point objects A, B, C and D. The DNF format of the constraint is: (North(A,B) ∧ NW(B,C) ∧ SE(B,D) ∧ NE(A,C)) ∨ (SE(A,B) ∧ NW(B,C) ∧ SE(B,D) ∧ NE(A,C)). Figure 4.13(a) and (b) are X/Y-graphs for the two conjunctions, which are $(North(A,B) \wedge NW(B,C) \wedge$

$SE(B,D) \wedge NE(A,C))$ and $(SE(A,B) \wedge NW(B,C) \wedge SE(B,D) \wedge NE(A,C))$.



(a) before(S,R) and meets(S, L) and overlaps(L,R) (b) metby(S,R) and meets(S, L) and overlaps(L,R)

Figure 4.14: Constraint graphs for (S meets L) and (S before or metby R) and (L overlaps R)

Figure 4.14 is an example of the dimension graph maintaining the constraint of *"S meets L and S before or metby R and L overlaps R"* for intervals S, L and R. The DNF format of the constraint is: $(before(S,R) \wedge meets(S,L) \wedge overlaps(L,R)) \vee (metby(S,R) \wedge meets(S,L) \wedge overlaps(L,R))$. The two conjunctions correspond to Figure 4.14(a) and (b) respectively.

4.5.2 Consistency Checking for General Constraints

After constructing the corresponding dimension graph for the constraints, the cycle detection function is performed on each subgraph representing a conjunctive constraint. If each set of subgraph contains cycle, the constraint is inconsistent. If a subset of graphs contains cycle, the constraint combinations corresponding to those graphs containing cycles are inconsistent. The combinations corresponding to the graphs without cycles are consistent. Algorithm 14 is the pseudo-code for consistency checking for general format constraints among points or intervals. The consistency checking algorithm contains three steps: Normalizing the constraints to standard DNF formats; Construct dimension graphs for each conjunction in DNF; Perform cycle detection on each graph.

The subfunction **normalize** preprocesses the constraint and transform the general format constraint into its DNF format. Secondly, check the consistency for each conjunction of the DNF representation by calling the Algorithm **conjunctionConsistency-Check** or **MBRconjunctionConsistencyCheck** according to the type of the objects.

Algorithm 14 Consistency checking: **consistencyCheck**

```
Input: constraint is the constraint need to be checked
Output: TRUE if consistent, FALSE otherwise

consistencyCheck(Set of Predicates constraint) {
    DNF dnfConstraint= normalize(constraint);
    Set of Predicates consistentConstraint = ∅;
    for each conjunction p ∈ dnfConstraint {
        if objects are MBRs
            if MBRconjunctionConsistencyCheck(p)
                add p to consistentConstraint;
        else if conjunctionConsistencyCheck(p)
            add p to consistentConstraint;
    }
    if consistentConstraint == ∅
        return FALSE;
    return TRUE;
}
```

If the conjunction constraint is consistent, add the conjunction to consistentConstraint. The final result of consistentConstraint contains all the consistent constraint combination, each of which represents a global consistent constraint.

In the example of constraint among point A, B, C and D shown in Figure 4.13, the dimension graph corresponding to conjunction North(A,B) $\wedge$ NW(B,C) $\wedge$ SE(B,D) $\wedge$ NE(A,C) contains a cycle, and hence this conjunction constraint is inconsistent. The consistentConstraint will only include the consistent conjunction, which is SE(A,B) $\wedge$ NW(B,C) $\wedge$ SE(B,D)$\wedge$ NE(A,C)(Figure 4.13(b)). Similarly, in the example for intervals illustrated in Figure 4.14, the conjunction depicted in Figure 4.14(b) contains cycle and hence the corresponding conjunction is inconsistent. Figure 4.15 shows the only consistent conjunction and one of its possible interval configurations.

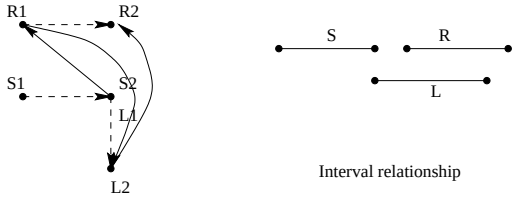


Figure 4.15: S meets L and S before R and L overlaps R)

Complexity Analysis

Similarly, let N be the number of spatial objects and E be the number of spatial predicates being checked for consistency, and COR be the number of all possible combinations of disjunctive predicates, i.e., the number of conjunctions in DNF.

- Time complexity = O(N+E)*O(COR);
  The consistency checking algorithm for general format constraints first generates the DNF for the constraints, and then call the **conjunctionConsistencyCheck** algorithm for each conjunction. O(N+E) is the time for consistency checking for a conjunction constraint. Therefore, the total time bound is O(N+E)*O(COR).

- Space complexity = O(N+E).
  COR does not contribute to space factor, since one may process one graph at a time.

4.6 Conclusions and Future Work

In this chapter, We propose a new strategy to process consistency checking for Euclidean spatial constraints among objects. We use a geometric approach by incorporating the spatial domain information in consistency checking. Dimension graphs are proposed to maintain the spatial constraints among objects. Each conjunctive constraint is projected on both dimensions, and a dimension graph is constructed for the constraint on each dimension. The spatial constraints in general format are maintained in a set of dimension graph constructed from its conjunctions. By using this framework, constraint consistency checking is then transformed to a graph cycle detection problem on its dimension graph. The cycle detection could be solved efficiently with O(N+E) time as well as space complexity, where N is the number of spatial objects, and E is the number of spatial predicates in the constraint. Recall that Allen's algorithm has a time and space complexity of $O(N^2)$. The proposed approach to consistency checking for spatial constraints is faster when the number of predicates is much smaller than N^2 and there are few disjunctions in the spatial constraint. The dimension graph and consistency checking algorithm can be used for points, and MBRs in 2 dimensional space. Since the algorithm returns TRUE if and only if the dimension graph of at least one conjunction contains no cycle, which means there exist at least one consistent overall constraint. The algorithm guarantees the global consistency.

In future work, we would like to explore the consistency checking among mixed types of object, e.g. consistency checking for constraints between point and intervals. We would also like to apply the dimension graph based cycle detection algorithm to the application of image similarity retrieval based on the similarity of spatial configuration.

Chapter 5

Equivalence Classes of Direction Objects and Applications

5.1 Introduction

Direction is a common spatial concept that is used everywhere in daily life. When people communicate about a geographic space, such as giving route descriptions[1], direction is necessary to convey the information. Direction is also frequently used as a selection condition in spatial databases[28, 15] or used for similarity accessing in image databases[53]. There are three commonly used directional reference frames, i.e., absolute directions, viewer-orientation-based directions and object-orientation-based directions. Absolute direction of a target object relative to a reference object is defined by their locations in the embedding space. Viewer-orientation-based direction and object-orientation-based direction are described with respective to the orientation of the view and the orientation of the reference object respectively. It is important to provide a framework to unify the representation of directional predicates and their reasoning for these three reference frames.

In this chapter we introduce the notation of equivalence classes of direction objects to model direction predicates. The object view of direction simplifies reasoning with direction predicates as it reduces the large number of inference rules that is commonly needed. This is useful in the processing and optimization of spatial queries that contain

direction constraints. Many axioms for the qualitative reasoning with direction predicates in previous work can now be understood easily in terms of direction equivalence classes. In addition, new reasoning with object-orientation-based direction predicates can be performed with our results.

5.1.1 Related work and our contributions

Research on direction modeling has been carried out in several areas such as geographic information systems and image analysis. Most of the previous work is on how to capture the semantics of directional relationships, and further conduct spatial reasoning on directions[15, 18, 20]. There are two major direction reference frames used to model direction in 2D space: the cone-based model[51], and the projection-based model[18, 20]. Frank[3] compared these two models and found the projection-based reference frame to be better in many aspects.

The previous work modeled direction as a boolean spatial relationship between objects[18, 60, 25, 76, 15, 51, 26, 20, 62, 48]. Qualitative spatial reasoning is then performed by applying large number of inferencing rules [4, 15, 18]. Frank[18] discussed a method for formal, qualitative spatial reasoning for cardinal directions, and list properties of cardinal directions. Freksa [20] developed a representation schema which incorporated orientation information. All these studies lacked an independent interpretation model, and thus the framework was hard to understand, and only useful for specific direction predicates. On the other hand, most attention of the previous studies has been paid to modeling and reasoning about absolute directions. The results may be applicable to viewer-orientation-based reference frames, however, reasoning in object-orientation-based reference frame needs more study. The complexity of object-orientation-based direction is due to the property that the referenced orientation framework may be different for different reference objects.

In this chapter, we extend the object model proposed in our recent work [64], where direction is modeled as a spatial object, i.e., a unit vector. We decompose direction space into disjointed partitions and define equivalence classes of direction objects to

unify the reasoning of different set of directional predicates. We also define an algebra on the equivalence classes to formalize reasoning with direction predicates. The equivalence classes of direction objects together with the algebra defined on it provide an independent interpretation model for qualitative direction reasoning. The properties of direction predicates can be understood easily in terms of the properties of direction equivalence classes. Many Lemmas discussed here were used as desirable properties(axioms) of direction predicate sets in previous work, whereas these are derived properties from equivalence class of direction objects in our framework. Lemma 14 shows the lack of closure under composition, which was often assumed in previous work [18]. We show a simple reason for this using the *between* operator on direction object proposed in [64]. Many inferencing rules in previous work can be derived easily using our framework and thus reducing the large number of inference rules needed. We also explore inferencing in object-orientation-based reference frame using our framework.

5.1.2 Scope and Outline

In this chapter, we explore the problem of spatial reasoning within different direction reference frames. We focus on constructing a unified framework for reasoning different set of direction predicates and providing an independent interpretation model for spatial reasoning. We restrict our discussion to two dimensional point objects. The framework can be refined to apply to higher dimensions.

The organization of this chapter is as follows: In section 2, We introduce the basic concept of direction equivalence classes and their properties. Qualitative directional reasoning for viewer-orientation-based and object-orientation-based reference frames are introduced in section 3, where the properties of direction predicates are also characterized by a set of lemmas. In section 4, as a future work, we apply our direction model to the landmark based route description problem, proposing a set of path predicate. Finally, we conclude the chapter with discussions and suggestions for future research.

5.2 Equivalence Classes of Direction Objects

We have previously introduced the concept of 'direction object' [64]. In this section, we decompose the space into disjointed partitions and identify each 'direction' with the partition in which it lies. All directions in a particular partition are considered 'equivalent'. We also define an algebra on the set of partitions and use the algebraic operations to solve 'equations', where the variables are 'direction objects'.

There are several ways of partitioning the space of directions. Figure 5.1 illustrates example partitioning for a two-dimension space of direction unit vectors. Figure 5.1(a) and 5.1(b) represent the common cone-shape divisions[51]. Figure 5.1(a) divides direction unit vectors into 4 groups, namely, $0^0 \leq \theta < 90^0$, $90^0 \leq \theta < 180^0$, $180^0 \leq \theta < 270^0$ and $270^0 \leq \theta < 360^0$, where θ denotes the angle from a coordinate axis parallel to the edge of a sheet of paper to the direction vector. An alternative partitioning imposed by Figure 5.1(a) may have eight group, namely, 0^0, 90^0, 180^0, 270, $0^0 < \theta < 90^0$, $90^0 < \theta < 180^0$, $180^0 < \theta < 270^0$ and $270^0 < \theta < 360^0$. The term "partition" is used in a set-theoretical sense, i.e., partitions are pair-wise disjointed and together they cover the space of all direction unit vectors(in 2D space). Figure 5.1(c) represents a clock-based division used in defense related applications. Some of the partitions imposed are $0^0 \leq \theta < 30^0$, $30^0 \leq \theta < 60^0$, and etc.

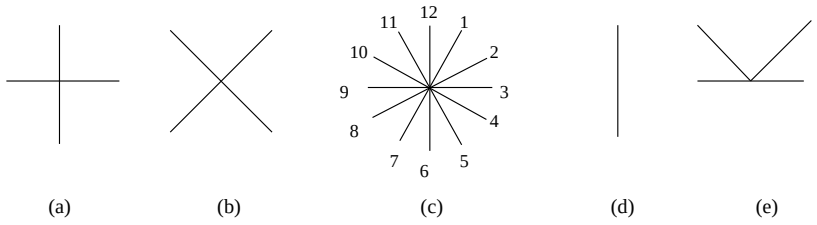


Figure 5.1: All except (e) are examples of symmetric partitioning. (d) is useful for path reasoning

After a partition scheme has been identified, a given $\vec{p}$ may be identified with a unique partition $[\vec{p}]$ in the partition set. Figure 5.2 below shows an example where the plane has been decomposed into four partitions. Each unit vector anchored at the origin is mapped into one of the four partitions. Each partition is identified with a symbol. In

the example of Figure 5.2 these symbols are N, E, S and W, standing for North, East, South and West.

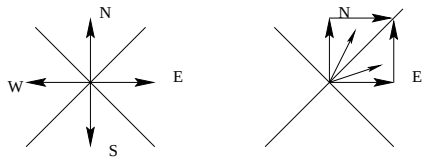


Figure 5.2: (a) Direction equivalence class, (b) Composition is *between*("vector sum") operation

Formally this is done by defining a relation $\sim$ on the set of all directions as follows:

$$\vec{A} \sim \vec{B} \iff \vec{A} \text{ and } \vec{B} \text{ share the same partition.}$$

Lemma 8 $\sim$ *is an equivalence relation.*

Proof: reflexive: By the definition of $\sim$, $\vec{A} \sim \vec{A}$ for any $\vec{A}$ in the set of direction predicates.

symmetric: If $\vec{A}$ and $\vec{B}$ are direction predicates and $\vec{A} \sim \vec{B}$, by definition, $\vec{A}$ and $\vec{B}$ share the same partition, which implies $\vec{B} \sim \vec{A}$.

transitive: If $\vec{A}, \vec{B}$ and $\vec{C}$ are direction predicates and $\vec{A} \sim \vec{B}$ and $\vec{B} \sim \vec{C}$, then $\vec{A}$ and $\vec{B}$ share the same partition and $\vec{B}$ and $\vec{C}$ share the same partition. This clearly implies that $\vec{A}$ and $\vec{C}$ share the same partition. Thus $\vec{A} \sim \vec{C}$. ■

We now define a composition operation $\oplus$ on the equivalence classes. $[\vec{P}]$ represents a direction equivalence class which contains the direction $\vec{P}$. If $\vec{P} \sim \vec{Q}$ then $[\vec{P}] = [\vec{Q}]$.

- Composition: $[\vec{P}] \oplus [\vec{Q}] = \{[\vec{R}] \mid \exists d_1 \geq 0 \; and \; d_2 \geq 0 \; s.t. \; \vec{R} = d_1\vec{P} + d_2\vec{Q}\}$

The composition operation $\oplus$ generalizes the ordinary vector sum operation to direction equivalence classes. Since direction equivalence classes only concern with directions, the composition operation defined here is related to the *between* operator[64]. It is for this reason that direction equivalence classes may not be closed under operation $\oplus$. In figure 5.2(b), $[\vec{N}] \oplus [\vec{E}] = \{[\vec{N}] \vee [\vec{E}]\}$.

Lemma 9 *Composition is commutative and associative.*

$$[\vec{P}] \oplus [\vec{Q}] = [\vec{Q}] \oplus [\vec{P}] \ , \ [\vec{P}] \oplus ([\vec{Q}] \oplus [\vec{R}]) = ([\vec{P}] \oplus [\vec{Q}]) \oplus [\vec{R}]$$

Proof: Follows from the commutative and associative properties of the vector sum operation. ■

We explicitly define a *reverse* operation $rv([\vec{P}])$ on the direction equivalence classes which are based on a symmetric partitioning. To do this, we first define a symbol $\vec{O}$ to represent a null vector:

$$\text{Reverse} : rv([\vec{P}]) = \{[\vec{Q}] \mid \forall \vec{P} \in [\vec{P}], \exists \vec{Q} \in [\vec{Q}] \ s.t. \ \vec{P} + \vec{Q} = \vec{O}\}.$$

Each direction equivalence class has a unique reverse if the partitioning is symmetric, e.g. Fig 5.1(a)-(d).

Lemma 10 *Reverse distributes over Composition. i.e.,* $rv([\vec{P}] \oplus [\vec{Q}]) = rv([\vec{P}]) \oplus rv([\vec{Q}])$

Proof: Follows from the distributive property of scalar multiplication over vector sum in vector algebra. Note that $rv([\vec{P}])$ can also be defined in terms of the equivalence class of a scalar product, i.e., $[(-1) \times \vec{P}]$.

Table 5.1 shows the notations used in the rest of the chapter.

Notation	meaning
$\vec{P}$	direction object(unit-vector)
$[\vec{P}]$	equivalence class of $\vec{P}$
$\vec{O}$	a null vector
$\oplus$	composition of equivalence classes
rv	reverse operator of equivalence classes

Table 5.1: Notation table

5.3 Orientation-based Direction Inferencing

In the previous section, we defined direction equivalence classes. Here we model direction predicates and reasoning for orientation-based reference frame, which includes viewer-orientation-based and object-orientation-based. We work with Figure 5.1(a)-like partitioning. Many of the theorems and lemmas are more general and can apply to predicate sets based on other symmetric partitioning. An interesting implication of direction predicates as direction equivalence classes is lemma 15 on predicate interchangeability in composition. This property is surprising on the surface yet easy to understand given our interpretation using direction equivalence classes.

5.3.1 Viewer-orientation-based Direction Inferencing

Viewer-orientation-based direction refers to the direction relationship which is measured

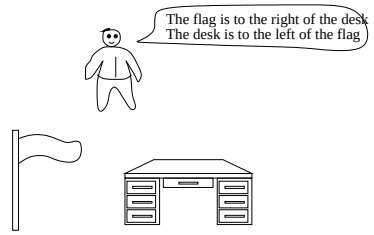


Figure 5.3: Orientation-based Direction

from a single viewer's perspective. There are three related components: target object A, reference object B, and the viewer. The viewer has his/her own orientation, where objects A and B may or may not be oriented objects. In Figure 5.3, the flag is to the right of the desk from the viewer's perspective.

Direction predicates for viewer-orientation-based reference frame

We begin by defining a sample set of direction predicates based on Figure 5.1(a)-like partitioning in the viewer-orientation-based reference frame as shown in Table 5.2. The corresponding direction equivalence classes are shown in Figure 5.4(b). While the arrows $\vec{EF}, \vec{EL}, \vec{EB}, \vec{ER}$ have exact directions, the other four $\vec{RF}, \vec{LF}, \vec{LB}, \vec{RB}$ can point to

any direction in their respective quadrants. As illustrated in Figure 5.4(a), $O_v.\vec{front}$ and $O_v.\vec{right}$ are two orthogonal unit vectors which represent the orientation of viewer V. $\vec{BA}$ refers to the direction object constructed from the vector $A - B$. The predicate

Predicates	$\vec{BA} \odot O_V.\vec{front}$	$\vec{BA} \odot O_V.\vec{right}$	equivalence classes	Partitions
$SP(A,B)$	Undefined		$[\vec{O}]$	
$EF(A,B)$	1	0	$[\vec{EF}]$	$\theta = 90^0$
$EB(A,B)$	-1	0	$[\vec{EB}]$	$\theta = 270^0$
$ER(A,B)$	0	1	$[\vec{ER}]$	$\theta = 0^0$
$EL(A,B)$	0	-1	$[\vec{EL}]$	$\theta = 180^0$
$RF(A,B)$	> 0	> 0	$[\vec{RF}]$	$0^0 < \theta < 90^0$
$LF(A,B)$	> 0	< 0	$[\vec{LF}]$	$90^0 < \theta < 180^0$
$RB(A,B)$	< 0	> 0	$[\vec{RB}]$	$270^0 < \theta < 360^0$
$LB(A,B)$	< 0	< 0	$[\vec{LB}]$	$180^0 < \theta < 270^0$

Table 5.2: Predicates for Viewer-orientation-based Direction

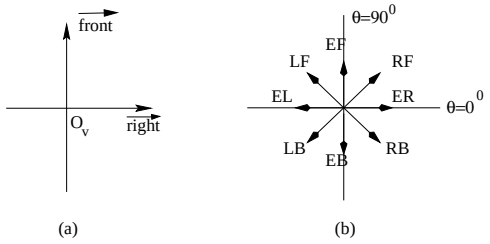


Figure 5.4: Direction equivalence classes for viewer-orientation-based direction predicates

set consists of nine predicates, each of which states whether the direction $\vec{BA}$ belongs to the equivalence class representing the corresponding partition. The predicate $SP(A,B)$ returns TRUE if and only if object A and B are in the same location. Each of the four predicates $EF(A,B)$, $EB(A,B)$, $EL(A,B)$ and $ER(A,B)$ returns TRUE if object A is located on the straight line of the exactly front, behind, right, and left of B respectively. On the other hand, direction range predicates $RF(A,B)$, $LF(A,B)$, $RB(A,B)$, and $LB(A,B)$ represents that object A is within corresponding direction range of B from V's the viewpoint. For instance, $RF(A,B)$ Returns TRUE iff object A is to the Right and Front of B from V's viewpoint.

Theorem 4 *The direction predicates are invariant with respect to viewer translation.*

Proof: Given objects A , B and viewer V with orientation $(O_V.\vec{front}, O_V.\vec{right})$, all the predicates are determined by the dot product of $\vec{BA}$ with the individual orientation components $O_V.\vec{front}$ and $O_V.\vec{Right}$. Now if the viewpoint changes from V to V' with new orientation $(O_{V'}.\vec{front}, O_{V'}.\vec{right})$, then $O_{V'}.\vec{front} = rot(\theta_1)(O_V.\vec{front})$ and $O_{V'}.\vec{right} = rot(\theta_1)(O_V.\vec{right})$, where $rot(\theta_1)$ is the rotation-matrix: $\begin{pmatrix} \cos(\theta_1) & \sin(\theta_1) \\ -\sin(\theta_1) & \cos(\theta_1) \end{pmatrix}$. Thus the new orientations of V' are derived exclusively using the orientations of V. Hence all the predicates defined are translation invariant. ■

Reasoning with Direction Predicates

We can easily derive Table 5.3 from the definition of rv for corresponding direction equivalence classes. For example, if A is exactly left of B, then B is exactly right of C from the same viewpoint.

Predicates	SP(A,B)	EF(A,B)	EB(A,B)	ER(A,B)	EL(A,B)
Reverse	SP(B,A)	EB(B,A)	EF(B,A)	EL(B,A)	ER(B,A)
Predicates	RF(A,B)	RB(A,B)	LF(A,B)	LB(A,B)	
Reverse	LB(B,A)	LF(B,A)	RB(B,A)	RF(B,A)	

Table 5.3: Reverse Relationship Between Direction Predicates

Applying the composition operation $\oplus$ on the corresponding direction equivalence classes gives us the composition table for the direction predicate set as shown in Table 5.4. The first row enumerates the eight possible direction predicates of A w.r.t to B, and the first column consists of the eight possible direction predicates for B relative to C from the same viewpoint. The contents of the table stand for the direction predicates for A relative to C from the viewer's viewpoint. For instance, $EL(A,B) \oplus EF(B,C) = LF(A,C)$.

This composition table is similar to the composition table obtained from previous work, where the table was constructed in brute-force manner. Many properties characterized as lemmas in previous work can now be easily understood by the definition of the algebra on equivalence classes. Some interpretations of the properties are shown as following. Here, DPS denotes the set of direction predicates, which is {SP, EF, EB, ER,

$\oplus$	EF(A,B)	EB(A,B)	EL(A,B)	ER(A,B)	LF(A,B)	LB(A,B)	RF(A,B)	RB(A,B)
EF(B,C)	EF	EF $\vee$ EB $\vee$ SP	LF	RF	LF	EL $\vee$ LF $\vee$ LB	RF	ER $\vee$ RF $\vee$ RB
EB(B,C)	EF $\vee$ EB $\vee$ SP	EB	LB	RB	EL $\vee$ LF $\vee$ LB	LB	ER $\vee$ RF $\vee$ RB	RB
EL(B,C)	LF	LB	EL	EL $\vee$ ER $\vee$ SP	LF	LB	LF $\vee$ EF $\vee$ RF	EB $\vee$ LB $\vee$ RB
ER(B,C)	RF	RB	ER $\vee$ EL $\vee$ SP	ER	EF $\vee$ LF $\vee$ RF	EB $\vee$ LB $\vee$ RB	RF	RB
LF(B,C)	LF	LB $\vee$ EL $\vee$ LF	LF	EF $\vee$ LF $\vee$ RF	LF	LB $\vee$ EL $\vee$ LF	LF $\vee$ EF $\vee$ RF	U
LB(B,C)	LF $\vee$ LB $\vee$ EL	LB	LB	EB $\vee$ RB $\vee$ LB	LB $\vee$ EL $\vee$ LF	LB	U	EB $\vee$ LB $\vee$ RB
RF(B,C)	RF	RB $\vee$ RF $\vee$ ER	EF $\vee$ LF $\vee$ RF	RF	EF $\vee$ LF $\vee$ RF	U	RF	ER $\vee$ RF $\vee$ RB
RB(B,C)	RF $\vee$ RB $\vee$ ER	RB	EB $\vee$ LB $\vee$ RB	RB	U	EB $\vee$ LB $\vee$ RB	ER $\vee$ RF $\vee$ RB	RB

Table 5.4: Composition table for viewer-orientation-based Direction Predicates P(A,C), where P is one of EF, EB, EL, ER, LF, LB, RF, and RB

EL, RF, RB, LF, LB }.

Lemma 11 *Reverse operations on direction predicates is its own reverse, i.e.,*

$$reverse(P(A,B)) = Q(B,A) \implies reverse(Q(B,A)) = P(A,B)$$

Proof: Follows from the definition of operation rv on direction equivalence classes. rv requires a symmetric partitioning. ■

Lemma 12 *No direction predicate except SP in DPS is symmetric, i.e.* $\forall P,\ P \in DPS$, $P(A,B)$ *and* $P(B,A)$ *cannot both hold.*

Proof: Follows from having more than one direction equivalence class based on symmetric partitioning. ■

Lemma 13 *Each direction predicate is transitive. i.e.* $\forall P, P \in DPS$, $P(A,B) \oplus P(B,C) \Longrightarrow P(A,C)$.

Proof: If $P(A,B)$ and $P(B,C)$ both hold, then there exists a $[\vec{R}]$ such that $P(A,B) \oplus P(B,C) \in [\vec{R}]$. By definition of $\oplus$, $[R] \oplus [R] = [R]$. Therefore the predicates are transitive. ■

Lemma 14 *Set of direction predicate is closed under reverse but not under composition. i.e.* $\forall P$, $P \in DPS$ *and* $P(A,B)$, $\exists P' \in DPS$, *s.t., P'(B,A) holds.*

Proof: Symmetric partitioning provides closure under reverse. The *between* semantic of the $\oplus$ operation leads to the lack of closure for the composition. ■

Note that previous work [18] stated that the composition operation yields a set of values (not a single value) to avoid problems related to uniqueness assumptions. The equivalence classes on directions provide a clear explanation for this. The $\oplus$ operation uses the *between* operator on directions leading to a set of values.

Lemma 15 *Predicates in composition are interchangeable despite different arguments. i.e.,* $\forall P_1 \in DPS, P_2 \in DPS$, $P_1(A,B) \oplus P_2(B,C) = P_2(A,B) \oplus P_1(B,C)$.

Proof: According to the definition of equivalence classes, if a predicate holds for different argument pairs, the vector formed by the two arguments falls into the same direction equivalence class. Since $\oplus$ is commutative, thus the predicates, i.e., direction equivalence classes, are interchangeable. Figure 5.5 is an example. Here, $EF(A,B) \oplus EL(B,C) = LF(A,C) = EL(A,B) \oplus EF(B,C)$. ■

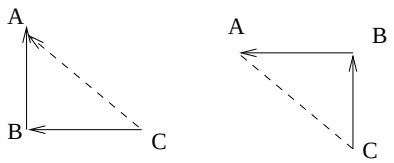
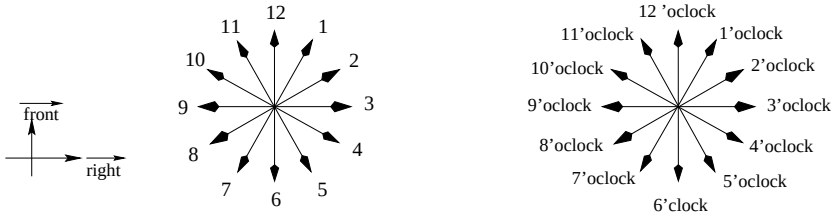


Figure 5.5: Interchangeability of direction predicates

5.3.2 Direction Inferencing on Different Predicate Sets

The direction inferencing on other different predicate sets can be performed easily by applying the algebra on direction equivalence classes without needing more inference rules. Figure 5.6(a) shows a direction reference frame that is frequently used in battlefield applications. This clockwise reference frame shows more refined direction informations than the reference frames defined based on Figure 5.1(a)-like partitions.



(a)reference frame (b) examples of equivalence classes

Figure 5.6: clockwise reference frame

A sample predicate set for this reference frame can be defined to consist of 24 predicates, e.g., *1o'clock, 2o'clock, 3o'clock, ···, 12o'clock, 1-2o'clock, 2-3o'clock, ···, 11-12o'clock, 12-1o'clock*. Each predicate corresponds to one equivalence class that represents one partition on the space. Figure 5.6(b) illustrates a subset of equivalence classes corresponding to the 12 exact directions of the reference frame. For example, 1o'clock(A, B) means that object A is in the direction of 1o'clock to B, which corresponds to the equivalence class of [1o'clock]. Table 5.5 shows the definitions of the equivalence classes

Exact direction		Range	
Equivalence class	Partition	Equivalence class	Partition
12o'clock	90^0	12-1o'clock	$(60^0, 90^0)$
1o'clock	60^0	1-2o'clock	$(30^0, 60^0)$
2o'clock	30^0	2-3o'clock	$(0^0, 30^0)$
3o'clock	0^0	3-4o'clock	$(-30^0, 0^0)$
⋮	⋮	⋮	⋮
11o'clock	120^0	11-12o'clock	$(90^0, 120^0)$

Table 5.5: Direction equivalence classes for clockwise reference frame

based on the partitions, which are described in terms of the angle between the directions

and the 3o'clock directed vector. 1-2o'clock(A, B) means that A is between the direction of 1o'clock and 2o'clock of B, which corresponds to the equivalence class of [1-2o'clock].

Based on the algebra defined on equivalence classes, we can perform the direction reasoning on this predicate set. For instance, the reverse of 1o'clock can be obtained by applying the rv operation on [1o'clock], resulting in 7o'clock. Figure 5.7 gives an example of inferencing for the directional relationships among three point targets A, B and C. Suppose we have the knowledge that point target A is 1o'clock to the target B, and target B is 3o'clock to the target C, we can infer the directional relationship between target A and C by applying the $\oplus$ operation on corresponding equivalence classes, which results in three possible relationships: 1-2o'clock or 2o'clock or 2-3o'clock.

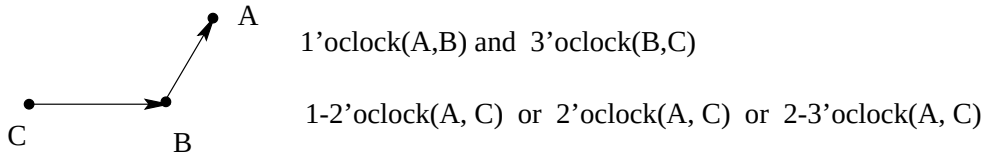


Figure 5.7: Example of direction inferencing

The direction reasoning for the predicates defined on mixed reference frames can also be performed using direction equivalence classes once the relative orientations of the two reference frames are fixed. Figure 5.7 shows such an example.

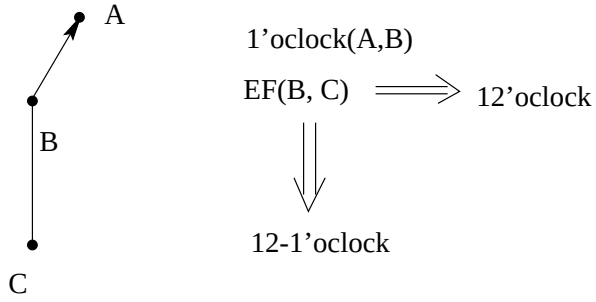


Figure 5.8: Example of reasoning for predicates defined on mixed reference frames

5.3.3 Object-orientation-based Direction Inferencing

Object-orientation-based direction is the direction of the target object with respect to the orientation of the reference object. The reference object is an oriented object, while the target object may or may not have orientation. In Figure 5.3, the person and the desk are oriented objects, and the person is behind the desk from the orientation of desk, and the desk is in front of the person w.r.t person's orientation.

The main difference between viewer-orientation-based and object-orientation-based reference frame is that the direction relationship is measured from a common viewer's viewpoint in viewer-orientation-based frame, on the contrary, in object-orientation-based reference frame, it is calculated with respective to the orientation of the reference object, and hence the referenced orientation framework may be different for different reference objects. Due to this property, direction inferencing in object-orientation-based reference frame is more difficult, and few studies addressed this problem.

Direction predicates for Object-orientation-based frame

We also begin the discussion by defining a set of direction predicates for object-orientation-based reference frame as in Table 5.6. Similarly, we restrict our discussion in 2D. The direction predicates are similar to those in viewer-orientation-based direction predicates table, but now with respective to the orientation of object B, which is $(O_B.\vec{front}, O_B.\vec{right})$. If the orientation of viewer is different from the orientation of reference object B, then the direction predicate P(A,B) that describes the direction of A related to B seen from viewer will be different from the direction predicate P'(A,B) which describes the direction related to B from B's orientation.

Direction predicates	SP	EF	EB	ER	EL	LF	LB	RF	RB
$\vec{BA} \odot O_B.\vec{front}$		1	-1	0	0	> 0	< 0	> 0	< 0
$\vec{BA} \odot O_B.\vec{right}$		0	0	1	-1	< 0	< 0	> 0	> 0

Table 5.6: Direction Predicates for Object-orientation-based frame for Object B

Viewpoint Transformation

As we discussed above, direction changes if the orientation of the reference object changes. Figure 5.9 illustrates one example. Object A is to the exactly right of B in 5.9(a), but it is between right and behind B if B's orientation rotates 45^0, as shown in 5.9(b). Table 5.7 lists the corresponding direction predicates after changing the orien-

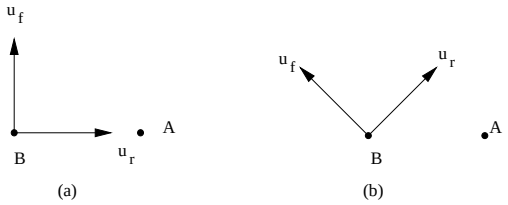


Figure 5.9: (a) A is ER B. (b) A is RB B

tation of the viewer. The first row enumerates eight direction relationship of A relative to B w.r.t. object B's orientation. The first column lists the relative orientation predicates that transforms the orientation of B to orientation of object C. And the content of the table illustrates the corresponding direction predicates that correctly represent the direction relationship of A relative to B with respective to the orientation of object C.

	Direction predicate as viewed from B							
orientation of B wrt C	EF	EB	ER	RL	RF	LF	RB	LB
$AntiClock90(B,C)$	ER	EL	EB	EF	RB	RF	LB	LF
$AntiClock180(B,C)$	EB	EF	EL	ER	LB	RB	LF	RF
$AntiClock270(B,C)$	EL	ER	EF	EB	LF	LB	RF	RB
$AntiClock0to90(B,C)$	RF	LB	RB	LF	$RB \vee RF$	$LF \vee RF$	$RB \vee LB$	$LB \vee LF$
$AntiClock90to180(B,C)$	RB	LF	LB	RF	$RB \vee LB$	$RB \vee RF$	$LF \vee LB$	$RF \vee LF$
$AntiClock180to270(B,C)$	LB	RF	LF	RB	$LB \vee LF$	$LB \vee RB$	$RF \vee LF$	$RB \vee RF$
$AntiClock270to360(B,C)$	LF	RB	RF	LB	$LF \vee RF$	$LF \vee LB$	$RB \vee RF$	$LB \vee RB$

Table 5.7: Transformation table when the viewpoint transforms from B to C.

Inference rules for object-orientation-based reference frame

The basic strategy of inferencing in object-orientation-based frame is to transform all predicates to a common viewer by looking up table 5.7. After the transformation, the problem then is reduced to the problem of inferencing in viewer-orientation-based frame,

which can be solved by using the inference rules as in table 5.4. Figure 5.10 illustrates this strategy.

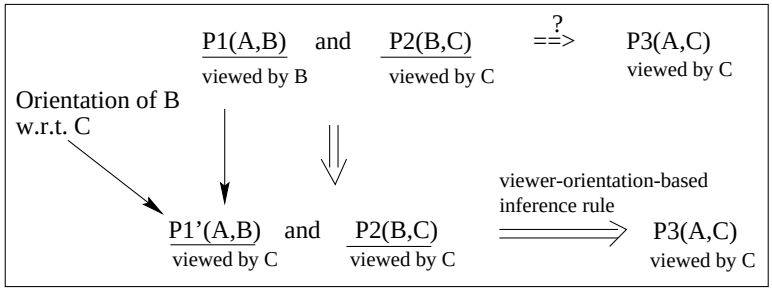


Figure 5.10: Strategy for Object-orientation-based direction inferencing

Here, given two relationships $P1(A,B)$(w.r.t. B's orientation) and $P2(B,C)$ (w.r.t. C's orientation), we want to infer the direction predicate P3(A,C), which describing the direction of A related to C from C's orientation. The first step is to do the viewpoint transformation by looking up in the table 5.2 and get the predicate P'(A,B), the problem then reduces to the inferencing within viewer-orientation-based frames.

Infer relative orientation given directions from two different viewers

One interesting application is that we can infer the relative orientation of two different viewers if we know the directions of an object from these two viewers. Table 5.8 illustrates this inference. The first row enumerates the directions of an object seen from the first

$\theta(V_1, V_2)$	EF	EB	ER	RL	RF	LF	RB	LB
EF	0	180	270	90				
EB	180	0	90	270				
ER	90	270	0	180				
EL	270	90	180	0				
RF	(0,90)	(180,270)	(270,360)	(90,180)	0	90	270	180
LF	(270,360)	(90,180)	(180,270)	(0,90)	270	0	180	90
RB	(90,180)	(270,360)	(0,90)	(180,270)	90	180	0	270
LB	(180,270)	(0,90)	(90,180)	(270,360)	180	270	90	0

Table 5.8: relative orientation from two viewpoints

viewer(V_1), and the first column enumerates the directions of the object seen from the second viewer(V_2). The contents of the table are the relative angles rotate from V_1 to V_2.

For example, if the object is *exactly front* from viewpoint of V_1, and *right and behind* from the viewpoint of V_2, then the relative orientation changed from V_1 to V_2 is $(90^0, 180^0)$.

5.3.4 Examples

1. *Viewer based inference:* In Figure 5.11, from the viewer V's perspective, A is in front of and to the right of B, i.e. $RF(A, B)$ and B is exactly in front of C, $EF(B, C)$. From table 5.2 we draw the conclusion that a viewer at V "sees" A as in front of and right of C. Now what will happen when the viewer's perspective changes to $V^{'}$? While the transformation from V to $V^{'}$ involves both a rotation and a translation, for inferencing only the rotation (see theorem below) transformation plays a role. Thus the viewpoint $V^{'}$ is derived from V by applying an anti-clockwise $270 degree$ rotation matrix. Consulting the table 5.7 we note that from viewer's $V^{'}$ perspective the relationships have undergone the following transformation:

$$RF(A, B) \Rightarrow LF(A, B)$$

$$EF(B, C) \Rightarrow EL(B, C).$$

Now we again consult the table 5.2 and conclude that from the vantage point of $V^{'}$, the relationship between A and C is defined by $LF(A, C)$.

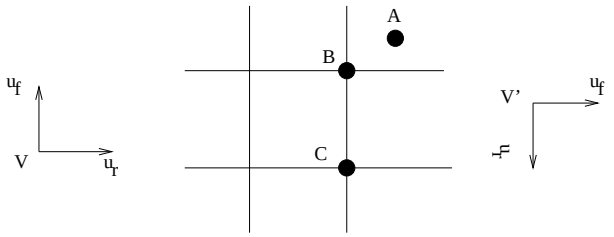


Figure 5.11: Viewer based inference: Only rotation plays a role.

2. *Absolute direction inference:* This is a special case of viewer-orientation-based inferencing. Here the viewer's perspective remains fixed and the orientation vectors are $(u_a, u_r) := (N, E)$.

3. *Object based inference:* This too is a special case of viewer based inferencing. Here the different viewers' orientation are assigned to the reference objects. For example,

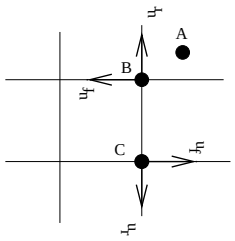


Figure 5.12: Object based inference: viewers perspective are attached to the reference objects'

in Figure 5.12, from $B's$ orientation A is behind and to the right of A, $RB(A, B)$. Similarly from $C's$ perspective B is exactly left of C, $EL(B, C)$. The relationship between A and C, from $C's$ perspective can again be derived by consulting Table 5.7:

$$RB(A, B) \overset{180^\circ}{\Rightarrow} LF(A, B) \text{ from } C's \text{ perspective.}$$

Now all the viewpoints coincide with that of C. A lookup in Table A and C is $LF(A, C)$.

5.4 Landmark Based Route Description

An important application in GIS is route navigation. The problem consists of two aspects: route planning and route description. There are many web routing servers that plan routes for users based on a roadmap with named roads. Given source and destination, routing servers can plan a route for users and show the resulting route as a route map, as well as route navigation description. There are several ways to present the route instructions: coordinate based node description, street sequence description, descriptions using absolute directions, and descriptions using relative directions. Most of the web routing servers try to give route descriptions using as much relative directions as possible plus necessary absolute directions.

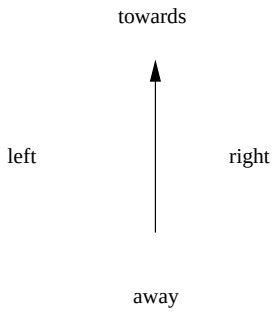
The problem gets complicated if the given roadmap has only named landmarks, but the road has no name, such as campus maps, or skyway maps. The route descriptions should be given by only using the named landmarks with proper direction predicates. There is few existing routing servers help this task. We will define a set of predicates which can be used to describe a route based only on landmarks. These predicates will be defined in terms of the direction predicates discussed in Section 2 and 3.

5.4.1 Path Predicates

The direction equivalence classes discussed in the previous section are for static directions in the sense that the reference object and the target object are stationary. In some cases, we have to measure the direction between a moving viewer and objects, such as route description problem.

We define a new equivalence class to characterize the direction relationship between a moving viewer(a directed edge) and point objects. We begin by partitioning the plane with respect to a directed edge as shown in Figure 5.13.

There are 4 direction equivalence class here: *towards, away, left, right*. Since the orientation of the viewer is aligned with the direction of the edge, *towards* and *away* are actually EF and EB of the viewer respectively. The equivalence class *left* consists of three



(a)

Figure 5.13: Plane partition determined by the edge

equivalence classed in 8-way partitioning system, which are LF, LB, EL. Similarly, *right* is composed of RF,RB, and ER. We can define the path predicates based on this 4-way partitioning. Table 5.9 give the definition for each path predicate by using the predicates we defined in the 8-way partitioning for static directions. Object X here could be an

path predicate	definition in terms of direction predicate
$towards(\vec{e}, X)$	$\exists$ point P $\in$ X, $\forall$ point $P_1 \in \vec{e}$,s.t., $EF(P, P_1)$
$left(\vec{e}, B)$	$\forall$ point P $\in$ X, $\forall$ point $P_1 \in \vec{e}$, s.t. $LF(P, P_1) \vee EL(P, P_1) \vee LB(P, P_1)$
$right(\vec{e}, B)$	$\forall$ point P $\in$ X, $\forall$ point $P_1 \in \vec{e}$, s.t. $RF(P, P_1) \vee ER(P, P_1) \vee RB(P, P_1)$
$away(\vec{e}, X)$	$\exists$ point P $\in$ X, $\forall$ point $P_1 \in \vec{e}$,s.t., $EB(P, P_1)$

Table 5.9: Defining Path Predicates

object of arbitrary shape, and the path predicates describes the relationship between the viewer moving along the edge with respect to the object. $towards(\vec{edge}, object)$ describes that the edge goes towards X when the object intersects an orientation preserving extension of the edge. On the other hand, if the object intersects a reverse orientation preserving extension of the edge then the object can be be described as moving "away" from the edge. If the object is to the left of the oriented edge then the edge is described as keep the object to its "left", otherwise, keep the object is to the "right". In figure 5.13b, the edge goes away from D, towards A, and keep B to the left, and C to the right.

Applying the reverse operation on the path predicates, we have the table:

The composition operation $\oplus$,defined in Section 2, assumes the role of *concatenation* on edge descriptions. We notice that because of the dynamic feature of path predicates,

path operator	reverse
towards	away
left	right
right	left
away	towards

Table 5.10: reverse operation for path predicate

some properties, such as commutativity and transitivity, are not valid here.

Lemma 16 *Concatenation is associative but not commutative.*

Lemma 17 *Reverse distributes over Concatenation.*

5.4.2 Basic Concepts: *Voronoi regions* and *Homotopic paths*

Two important issues related to the route description problem are the "choice" of landmarks for describing the path and the definition of an "acceptable" path description. We use the notion of *Voronoi regions* and *Homotopic paths* to address these issues.

Voronoi regions are geometric structures that record information about proximity. Let $P = \{p_1, p_2, \cdots, p_n\}$ be a set of points (called sites) in the two-dimensional Euclidean plane. Partition the plane by assigning every point in the plane to its nearest site. All those points assigned to p_i form the Voronoi region $V(p_i)$, which contains all the points at least as close to p_i as to any other site [47], i.e.,

$$V(p_i) = \{x | |p_i - x| \leq |p_j - x|, \forall j \neq i\}.$$

In this chapter, we generalize the voronoi diagram by allowing the set of point sites to be a set of polygon sides. The polygons are treated as obstacles, and all the points outside polygons form free space. The points in free space are partitioned into voronoi regions based on their distance to polygon sides. The voronoi region corresponding to one polygon side ps_i contains all the points on the free space which is at least as close to the polygon side as to any other polygon side.

$$V(ps_i) = \{x : |ps_i - x| \leq |ps_j - x|, \forall j \neq i\}$$

Here, $|ps_i - x|$ is defined as the shortest distance from x to ps_i. For two adjacent polygon side, the points on the extension of the bisector of the angle formed by these two sides are defined to be equal distance to these two sides, and the extension of the bisector separates the voronoi regions of these two sides. The dash line in figure 5.16b represents the voronoi diagram of the roadmap.

From the definition, we can notice that voronoi region and the polygon side has a one-one match relationship. Several basic functions can be defined based on this property:

$identifyRegion(polygonSide)$: Return the corresponding region that the parametric polygon side is defined.

$identifyPolygonSide(region)$: Return the polygon that this region is closest to.

$identifyLandmark(region)$: Return the landmark that this region is closest to.

$ptVoronoiRegion(point)$: Return the voronoi region to which the point belongs.

Homotopic paths Since we are dealing with qualitative direction and distance measures no path description will exactly replicate a given path. We therefore need to define what an "acceptable" path description is. We invoke the notion of homotopy from Topology [54] to define acceptable paths. Without getting into a technical definition, intuitively, two paths in a plane, with a coincident start point and end point, are Homotopic if they can be "continuously deformed" into one another without any cutting or lifting. For example in Figure 5.14(a) f and g are Homotopic, but in Figure 5.14(b) f and g are not Homotopic because there is a "hole" in the space which obstructs the continuous deformation of f into g or vice-versa. Landmarks are viewed as holes in the plane.

Our goal is to give a path description such that the resulting path is homotopic

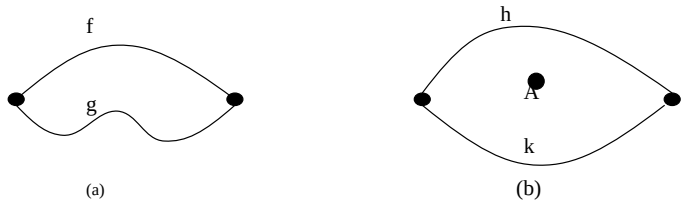


Figure 5.14: (a) f and g are homotopic. (b) f and g are not homotopic because of obstacle A

5.4.3 Problem Definition

Given: An oriented map(2D) with named landmarks and a route in terms of node-edge-node sequences in the free space , where the coordinates for all points are known.

Find: Route descriptions using landmarks and qualitative direction and distance measures.

Constraints: The given exact path and the described paths are homotopic.

Assumption: (1) All routes considered are simple, i.e, they are not self-intersecting. (2) The start and end landmarks of the path are distinguishable point objects.

Definition 1: A *path* is a coordinate based node-edge-node sequence.

Definition 2: A *path description* is a set of directional/distance predicates and conditions using named landmarks.

Definition 3: *Correctness:* A path description is correct if the path described is homotopic to the given path.

Theorem 5 *Given a path P and the five predicates {towards, away, left, right, past}, a path can be described which is correct i.e., the path resulting from the description and the original path are homotopic.*

Proof: We first show that the five predicates are sufficient to describe a path. As shown in the previous section, a directed edge partitions the space into four classes: **towards, away, left and right.** The notion of movement is implicit within the meaning of predicates *towards* and *away.* Situations where *towards* and *away* are not applicable can

be described using a conjunction of *past* and *left* or *right*. For example, "go *past* building A, keeping it your *left*." Since the begin and the end landmarks are point objects, the described path description begins with *away* and ends with *towards*.

We now show that the paths are homotopic. Let P be the given path and Q the

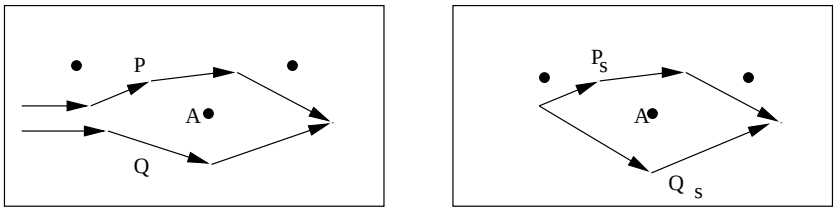


Figure 5.15: (a) P is the given path and Q is the described path. (b) P_s and Q_s are not homotopic.

approximate path generated using the path predicates. Assume P and Q are not homotopic. Then there exists a subpath P_s and Q_s such that P_s and and Q_s are not homotopic. Without loss of generality we can assume that P_s and Q_s have coincident start and end point. This means there exists a landmark A such that it has opposite orientation with respect to P_s and Q_s, i.e., if A is to the *left* of P_s then it is to the right of Q_s or vice-versa. See Figure 5.15. We can then define a new subpath $Q_s^{'}$ such that A has the same orientation with respect to $Q_s^{'}$ and P_s. Removing Q_s from Q and replacing it with $Q_s^{'}$ results in homotopy between P and Q. ■

Example of the problem

Figure 5.16 a gives an example road map of this problem. The figure shows a path generated using the freeway method[38]. We can describe the path as follows:

1. Go AWAY from *start* TOWARDS C.
2. Make a LEFT and go PAST B keeping it to the RIGHT.
3. Turn RIGHT and go PAST B keeping it to the RIGHT.

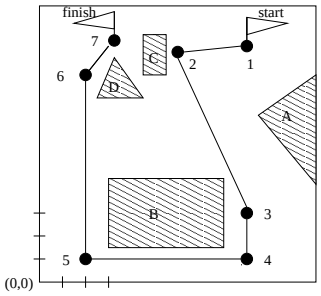


Figure 5.16: Freeway net with a generated path

4. Turn RIGHT and go PAST B and C keeping them to the RIGHT.

5. Go TOWARDS *finish*

If we use the aggregate predicates *around* and *between* we can have a more *natural* description.

1. Go AWAY from *start* TOWARDS C.

2. Make a LEFT and go BETWEEN A and B.

3. Go AROUND B.

4. Go past C keeping it to the RIGHT.

5. Go TOWARDS *finish*.

5.5 Conclusions and Future work

Directional relationships are important in many application fields such as GIS and image database. The object model of the direction proposed in our work[64] makes it easy to define more operations on direction and facilitates the processing of queries involving directional conditions. In this chapter we extend the object model of direction by defining the equivalence classes of direction objects and model the qualitative direction predicates as equivalence classes of direction objects. We also define an algebra on the equivalence classes to formalize the reasoning with direction predicates. The equivalence classes of direction objects together with the algebra defined on it provide an independent interpretation model for qualitative direction reasoning. Many inferencing rules in previous work can be derived easily using our framework and thus reducing the large number of inference rules needed.

In future work, we will attempt to extrapolate the definition of direction equivalence classes to define 'path' equivalence classes with an application to the landmark based route description problem.

Chapter 6

Conclusions and Future Work

This thesis dealt with a framework of modeling, maintaining and processing directional relationships in the context of spatial databases. The motivation was to enhance spatial database management systems with intelligent mechanisms to provide richer spatial abstract data types, to efficiently process directional relationships and to maintain the integrity consistency of spatial constraints among spatial objects in the database.

We design a framework to model and process directional relationships. The framework consists of several basic functional components: direction modeling, direction query processing, consistency checking for directional constraints and reasoning about directions. We propose an object direction model, based on which new ADTs for directions, orientations, and other spatial objects can be incorporated into spatial databases. We also investigate the query processing and consistency checking for a set of direction predicates on 2D Euclidean Coordinate system. For the query processing, we focus on processing object-orientation-based direction queries. We propose a new open shape based strategy which converts the processing of direction queries to the processing of topological operations between open shapes and objects. A dimension graph based consistency checking algorithm is proposed to maintain the consistency of the spatial constraints among objects.

6.1 Major Results

The primary contributions of this thesis are:

- An object model of direction. As a departure from traditional methods which model directions as binary boolean functions between objects, a novel vector based object direction model is proposed. The basic idea is to model directions as spatial objects. This object view of direction has several advantages. Being modeled as a spatial object, a direction object can have its own attributes and operation sets. Secondly, new spatial data types such as oriented spatial objects and open shapes can be defined at the abstract object level. Finally, the object view of direction makes direction reasoning easy and also reduces the need for a large number of inference rules.

- A new open shape based strategy (OSS) is proposed to process direction queries in spatial databases. OSS converts the processing of directional queries to the processing of topological operations between open shapes and objects. It eliminates false hits at the earliest opportunity while recursively searching hierarchical indices like R-tree. Since OSS models the direction region as an open shape, it also eliminates the computation related to the embedding world boundary. The algebraic analysis and experimental results demonstrate that OSS consistently outperforms classical strategies (often by an order of magnitude) in terms of both I/O and CPU cost. The experimental results also shows that OSS is scalable for large data sets.

- A dimension graph representation is proposed to maintain the Euclidean spatial constraints among objects. By using the dimension graph representation, the problem of consistency checking is then transformed to a graph cycle detection problem. The consistency checking can be achieved with O(N+E) time as well as space complexity, where N is the number of spatial objects, and E is the number of spatial predicates in the constraint. The proposed approach to consistency checking for spatial constraints is faster than $O(N^2)$ when the number of predicates is much smaller than N^2 and there are few disjunctions in the spatial constraint. The

dimension graph and consistency checking algorithm can be used for points, intervals and polygons in 2 dimensional space. The algorithm also guarantees the global consistency.

- The concept of equivalence classes of direction objects is proposed to unify the reasoning of different set of directional predicates. By defining an algebra on equivalence classes we construct a framework to model the semantics of direction predicates. The equivalence classes of direction objects together with the algebra defined on it provide an independent interpretation model for qualitative direction reasoning. We also discuss the inferencing for viewer-orientation-based and object-orientation-based reference frames using equivalence classes. Many inferencing rules in previous work can be derived easily using our framework and thus reducing the large number of inference rules needed.

6.2 Future Work

Beyond the particular results described above, this thesis provides a starting point for future work in the area of modeling and processing directional relationships. In this section, we identify some new research problems, which, combined with the results in this thesis, would enhance spatial database management systems with comprehensive mechanisms capable of handling different set of direction predicates in the context of different reference frames.

Modeling Directional Relationships Between Mixed-Dimension Objects

Current direction models and direction predicates deal with homogeneous type of objects. The direction predicate set defined in this thesis are focus on homogeneous type of objects, i.e., the relationship between point pairs, or polygon pairs. New study is needed to model the directional relationships between objects with different dimensions. For instance, the directional relationship between a point and a line, or the directional relationships between a line and a polygon. Figure 6.1 shows some configurations of a

line and a point. The relationships between curve and the points outside the curve are easily decided, while the relationship between the curve and the point in middle is hard to decide. More complicated predicates in different level of granularity needs more study.

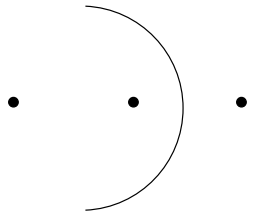


Figure 6.1: A curve and points

Query Processing for Viewer-orientation-based Direction Queries

In this thesis, we extend the previous work of processing absolute direction queries to process object-orientation-based direction queries. Another orientation-based direction frame is viewer-orientation-based directions. For example, consider the query: *List all the facilities behind the ridge.* Here, the ridge itself has no orientation, and direction predicate *behind* is with respect to the speaker's orientation. OSS strategy can be extended to process viewer-orientation-based direction queries by deriving more class and operations for ADT *OpenShape*.

Query Processing and Optimization

In this thesis, we focus on the processing strategies on selection queries which involve directional relationships related to a specific reference objects in the selection criteria. For queries related to more than one reference object, heuristics for optimizing the queries are useful. For instance, consider the query: *Find all the enemy targets which is to the right-front of the tank A and to the left of defense facility B.* Here, the selection criteria consist of two direction predicates: *right-front* of Tank A, and $left$ of the facility B. The result should contain the targets which satisfy the two conditions. The basic strategy is first to find all the targets to the right-front of the tank A, and find all the targets to the left of defense facility B, and finally return the intersection of the two sets. It is useful to

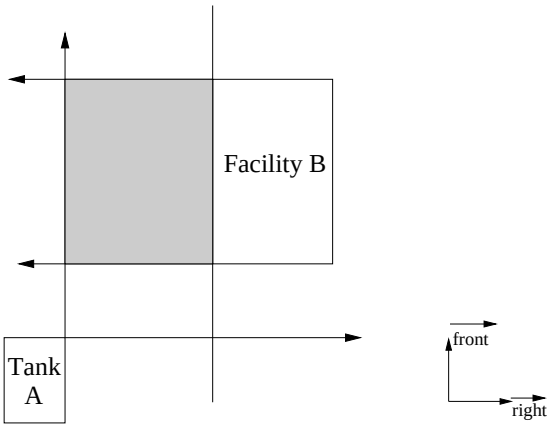


Figure 6.2: New filter for *right-front* of A and $left$ of B

construct a fast filter to reduce I/O cost and CPU cost. Extending OSS strategy could provide a fast filter. Figure 6.2 shows a new filter for this query in shaded region. More study is needed for extending the operations between open shapes and close objects in Chapter 3 to the operations between open shape pairs.

Spatial Indexing access methods

There are a lot of spatial indexing methods proposed. Classical one-dimensional indices such as the B^+ tree can be used for spatial data by linearizing a multi-dimensional space using a space-filling curve such as the Z-order. A large number of spatial indices [57] have been explored for multi-dimensional Euclidean space. Representative indices for point objects include Grid files, multi-dimensional grid files [39], Point-Quad-Trees, and Kd-trees. Representative indices for extended objects include the R-tree family, the Field tree, Cell tree, BSP tree, and Balanced and Nested grid files.

Which indexing mechanism is efficient for direction predicates is worth studying. In this thesis, we use R-tree as our indexing methods, it is useful to analyze the effect of different indexing methods such as Quad-Tree, Cell-tree, etc. Moreover, we deal with only one indexing mechanism in the database. However, in the case of orientation-based queries, the performance would be improved if there are multiple indices based on different orientation. Figure 6.3 shows the effect if we have two R-tree indices on the ori-

entation of 90^0 and 45^0 respectively. The multiple-index could be of same types(e.g., all are R-trees) or mixed types(R-tree in some orientation, Quad-tree in other orientation). The effect of different combinations needs further study.

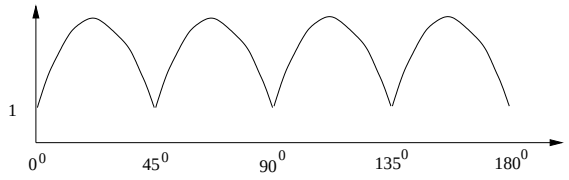


Figure 6.3: Effect of multiple indices

Bibliography

[1] Maps on us. http://www.MapsOnUs.com.

[2] N. Adam and A. Gangopadhyay. *Database issues in Geographical Information Systems.* Kluwer Academics, 1997.

[3] A.Frank. Qualitative Spatial Reasoning: Cardinal Directions as an Example. *International Journal of Geographical Information Systems*, 10(3):269–290, 1996.

[4] J. Allen. Maintaining Knowledge about Temporal Intervals. *Communications of the ACM*, 26(11):832–843, 1983.

[5] P. Baumann. Management of multidimensional discrete data. *VLDB Journal, Special issue on Spatial Database Systems*, 3(4):401–444, October 1994.

[6] Domenico Beneventano and etc. Consistency Checking in Complex Object Database Schemata with Integrity. *IEEE Trans. on Knowledge and Data Eng.*, 10(4), July/August 1998.

[7] E.A. Boiten, J. Derrick, H. Bowman, and M.W.A. Steen. Constructive Consistency Checking for Partial Specification in Z. In *Science of Computer Programming*, number 1, pages 29–75, September 1999.

[8] H. Bowman, E.A.Boiten, J. Derrick, and M.Steen. Strategies for Consistency Checking Based on Unification. In *Science of Computer Programming*, pages 261–298, April 1999.

[9] Thomas Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. Efficient processing of spatial joins using R-trees. In *Proceedings of the 1993 ACM-SIGMOD Conference on the Management of Data*, pages 237–246, Washington D.C., June 1993.

[10] D. Chamberlin. *Using the New DB2: IBM's Object Relational System.* Morgan Kaufmann, 1997.

[11] N. Chrisman. *Exploring Geographic Information Systems.* John Wiley and Sons, 1997.

[12] E. Clemintini and P. Di Felice. Topological invariants for lines. *IEEE Transactions on Knowledge and Data Engineering*, 10(1):38–54, 1998.

[13] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* The MIT press, McGraw-Hill Publishing Company, 1994.

[14] David J. DeWitt, Navin Kabra, Jun Luo, Jignesh M. Patel, and Jie-Bing Yu. Client-Server Paradise. In *Proceedings of the 20th International Conference on Very Large Data Bases,(VLDB'94)*, pages 558–569, Santiago de Chile, Chile, September 1994.

[15] D.Papadias, M. Egenhofer, and J. Sharma. Hierarchical Reasoning about Direction Relations. In *Fourth ACM Workshop on Advances in Geographic Information Systems*, pages 105–112. ACM, 1996.

[16] M. Egenhofer. Spatial SQL: A Query and Presentation Language. *IEEE Transactions on Knowledge and Data Engineering*, 6(1):86–95, 1994.

[17] M. Egenhofer. Consistency Revisited. *GeoInformatica*, 1(4):323–325, 1997.

[18] A. Frank. Qualitative Spatial Reasoning about Cardinal Directions. In *Auto carto 10, D.Mark and D. White, eds., Baltimore, MD*, pages 148–167, 1991.

[19] C. Freksa. Temporal Reasoning Based on Semi-Intervals. *Artificial Intelligence*, 54:199–227, 1992.

[20] C. Freksa. Using Orientation Information for Qualitative Spatial Reasoning. *Theories and Methods of Spatio-Temporal Reasoning Geographic Space*, 639:162–178, 1992.

[21] S. Freundschuh and M. Egenhofer. Human Conceptions of Spaces: Implications for GIS. *Transactions in GIS*, 2(4):361–375, 1997.

[22] Volker Gaede and Oliver Gunther. Multidimensional Access Methods. *Computing Surveys*, 30(2):170–231, 1998.

[23] Klaus-Peter Gapp. Basic Meanings of Spatial relations: Computation and Evaluation in 3D space. *AAAI*, 2:1393–1398, 1994.

[24] K. Gardels. Open GIS and On-Line Environmental Libraries. *ACM SIGMOD Record*, 26(1):32–38, March 1997.

[25] R. Goyal and M. Egenhofer. The Direction-Relation Matrix: A Representation of Direction Relations for Extended Spatial Objects . In *UCGIS Annual Assembly and Summer Retreat, Bar Harbor, ME*, 1997.

[26] G.Retz-Schmidt. Various Views on Spatial Prepositions. *AI Magazine*, 9(2):95–105, 1988.

[27] John Gurney and Elizabeth Kilpple. Composing Conceptual Structure for Spoken Natural Language in a Virtual Reality Environment . 1997.

[28] R.H. Güting. An Introduction to Spatial Database Systems. *VLDB Journal, Special issue on Spatial Database Systems*, 3(4):357–399, 1994.

[29] Antonin Guttman. R-trees: A Dynamic Index Structure For Spatial Searching. *Proceedings of ACM SIGMOD Conference*, pages 47–57, 1984.

[30] Daniel Hernandex. Maintaining Qualitative Spatial Knowledge. *Proc. of the European Conference on Spatial Information Theory, Elba, Italy*, pages 19–22, Sept. 1993.

[31] Gerd Herzog. Coping with Static and Dynamic Spatial Relations. In *Proceeding of the 5th International Workshop Time, Space and Movement*, 1995.

[32] Open GIS Consortium Inc. Opengis simple features specification for sql. http://www.opengis.org.

[33] Open GIS Consortium Inc. Spatial reference systems. http://www.opengis.org/techno/specs.htm.

[34] E. Jungert. The observer's point of view: An Extension of Symbolic Projections. *Theories and Methods of Spatial-Temporal Reasoning in Geographic Space*, 639:179–195, 1992.

[35] W. Kim, J. Garza, and A. Kesin. Spatial Data Management in Database Systems. In *Advances in Spatial Databases, 3rd International Symposium, SSD'93 Proceedings , Lecture notes in Computer Science, Vol. 692, Springer, ISBN 3-540-56869-7*, pages 1–13, Singapore, 1993.

[36] M. Kornacker and D. Banks. High-Concurrency Locking in R-Trees. In *Proceedings of 21st International Conference on Very Large Data Bases(VLDB'95)*, pages 134–145, Zurich, Switzerland, September 1995.

[37] V. Kumar. Algorithms for constraint satisfication problems: A Survey. *AI Magazine*, 13(1):32–44, 1992.

[38] Jean-Claude Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, 1991.

[39] J. Lee, Y. Lee, K. Whang, and I. Song. A physical database design method for multidimensional file organization. *Information Sciences*, 120(1):31–65, October 1997.

[40] Willem J. M. Levelt. Perspective Taking and Ellipsis in Spatial Descriptions. In *Language and Space, Bloom, Peterson, and etc. eds*, pages 77–107, 1996.

[41] D-R. Liu and S. Shekhar. A Similarity Graph-Based Approach to Declustering Problems and Its Application Toward Parallelizing Grid Files. In *Proceedings of the 11th International Conference on Data Engineering*, pages 373–381, Taipei, Taiwan, March 1995.

[42] Xuan Liu, Shashi Shekhar, and Sanjay Chawla. Consistency Checking for Euclidean Spatial COnstraints: A Dimension Graph Approach. Tech. Report TR00-039, University of Minnesota, Minneapolis, MN.

[43] Xuan Liu, Shashi Shekhar, and Sanjay Chawla. Processing Object-orientation-based Direction Queries. Tech. Report TR00-030, University of Minnesota, Minneapolis, MN.

[44] S. Manandhar. Deterministic consistency checking of LP constraints. In *Proceesings of the 7th Conference of the European Chapter of the Association for Computional Linguistics*, pages 165–172, Dublin, Ireland, March 1995.

[45] P. Meseguer. Constraint satisfication problems: An overview. *AI Communications*, 2(1):3–17, 1989.

[46] US Army Corps of Engineers. Topographic engineering center. http://www.tec.army.mil/gis-internet2.html.

[47] Joseph O'Rourke. *Computational Geometry in C.* Cambridge University Press, 1994.

[48] D. Papadias and T. Sellis. Qualitative representation of Spatial Knowledge in Two-Dimensional Space. *VLDB Journal*, 3(4):479–516, 1994.

[49] D. Papadias, Y. Theodoridis, and T. Sellis. The Retrieval of Direction Relations Using R-trees. *Proceedings of the 5th Conference on Database and Expert Systems Applications(DEXA)*, 1994.

[50] Jignesh M. Patel and David J. DeWitt. Partition Based Spatial-Merge Join. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 259–270, Montreal, Quebec, CA., 1996.

[51] Donna J. Peuquet and Zhan Ci-Xiang. An Algorithm to Determine the Directional Relationship Between Arbitrarily-shaped Polygons in the plane. *Pattern Recognition*, 20(1):65–74, 1987.

[52] M.H. Protter and C.B. Morrey. *Modern Mathematical Analysis.* Addison-Wesley Publishing Company, Inc., 1996.

[53] P.W.Huang and Y.R. Jean. Using 2D C^+-String As Spatial Knowledge Representation For Image Database Systems. *Pattern Recognition*, 30(10):1249–1257, 1994.

[54] R.Geroch. *Mathematical Physics.* The university of Chicago Press, 1985.

[55] David F. Rogers. *Mathematical Elements for Computer Graphics.* McGraw-Hill Publishing Company, 1990.

[56] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, Inc., 1995.

[57] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, 1990.

[58] S. Shekhar and S. Chawla. *Spatial Databases: Concepts, Implementation and Trends.* First draft, http://www.cs.umn.edu/Research/shashi-group/Book/index.html, 1998.

[59] S. Shekhar and B. Hamidzadeh. Learning Transformation Rules for Semantic Query Optimization: A Data-Driven Approach. *IEEE Trans. Knowledge and Data Eng.(Spatial Issue on Discovery in Databases)*, October 1993.

[60] S. Shekhar, S. Ravada, A.Fetterer, X.Liu, and C.T. Lu. Spatial Databases: Accomplishments and Research Needs. *IEEE Trans. Knowledge and Data Eng.*, 11(1):45–55, 1999.

[61] S. Shekhar, S. Ravada, V. Kumar, and etc. Parallelizing a GIS on a Shared-Address Space Architecture. *IEEE Computer*, 29(12):42–48, December 1996.

[62] Shashi Shekhar, Mark Coyle, and etc. Data Models in Geographic Information Systems. *CACM*, 40(4):103–111, 1997.

[63] Shashi Shekhar and Duen-Ren Liu. CCAM: A Connectivity-Clustered Access Method for Networks and Network Computations. *IEEE Trans. Knowledge and Data Eng.*, 9(1):102–119, 1997.

[64] Shashi Shekhar and Xuan Liu. Direction As a Spatial Object: A summary of Results. In *Sixth International Symposium on Advances in Geographic Information Systems*, pages 69–75. ACM, 1998.

[65] Shashi Shekhar, Xuan Liu, and Sanjay Chawla. Equivalence Classes of Direction Objects and Applications. Tech. Report TR99-027, University of Minnesota, Minneapolis, MN 55455.

[66] M. Stonebraker, J. Frew, and J. Dozier. The Sequouia 2000 Storage Benchmark. In *Proceedings of ACM SIGMOD Conference on the Management of Data*, pages 2–11, Washington D.C., May 1993.

[67] M. Stonebraker and G. Kennitz. POSTGRES Next-Generation Database Management System. *Communication of the ACM*, 34(10):78–92, 1993.

[68] M. Stonebreaker and D. Moore. *Object Relational DBMSs: The Next Great Wave.* Morgan Kaufmann, 1997.

[69] Eva Stopp, Klaus-Peter Gapp, and etc. Utilizing Spatial Relations for Natural language Access to an Autonomous Mobile Robot. In *Preceedings of KI-94*, 1994.

[70] Y. Theodoridis and D. Papadias. Range Queries Involving Spatial Realtions: A performance Analysis. *Proceedings of the 2nd Conference on Spatial Information Theory(COSIT)*, 1995.

[71] Yannis Theodoridis, D. Papadias, and E. Stefanakis. Supporting Direction relations in Spatial database Systems. *Proceedings of the 7th International Symposium on Spatial Data Handling(SDH'96)*, August 1996.

[72] C.D. Tomlin. *Geographic Information systems and Cartographic Modeling.* Englewood Cliffs, NJ:Prentice-Hall, 1990.

[73] UCGIS. Ucgis congressional breakfast. http://urban.rutgers.edu/ucgis, 1998.

[74] Jeffrey D. Ullman and Jennifer Widom. *A First Course in Database Systems.* Prentice Hall, 1997.

[75] M.F. Worboys. *GIS: A Computing Perspective.* Taylor and Francis, 1995.

[76] Y.I.Chang and B.Y. Yang. A Prime-Number-Based Matrix Strategy for Efficient Iconic Indexing of Symbolic Pictures. *Pattern Recognition*, 30(10):1–13, October 1997.