Efficient Implementation Techniques for Topological Predicates on Complex Spatial Objects: The Exploration Phase

Markus Schneider* & Reasey Praing

University of Florida
Department of Computer & Information Science & Engineering
Gainesville, FL 32611, USA
{mschneid, rpraing}@cise.ufl.edu

Abstract

Topological predicates between spatial objects have for a long time been a focus of research in a number of disciplines like artificial intelligence, cognitive science, linguistics, robotics, and spatial reasoning. Especially as predicates, they support the design of suitable query languages for spatial data retrieval and analysis in spatial database systems and geographical information systems. While, to a large extent, conceptual aspects of topological predicates (like their definition and reasoning with them) as well as strategies for avoiding unnecessary or repetitive predicate evaluations (like predicate migration and spatial index structures) have been emphasized, the development of robust and efficient implementation techniques for them has been rather neglected. Recently, the design of topological predicates for different combinations of *complex* spatial data types has led to a large increase of their numbers and accentuated the need for their efficient implementation. The goal of this article is to develop efficient implementation techniques of topological predicates for all combinations of the complex spatial data types *point2D*, *line2D*, and *region2D* within the framework of the spatial algebra SPAL2D. Our solution is a two-phase approach. In the *exploration phase*, for a given scene of two spatial objects, all *topological events* are recorded in two precisely defined *topological feature vectors* (one for each argument of a topological predicate) whose specifications are characteristic and unique for each combination of spatial data types. These vectors serve as input for the *evaluation phase* which analyzes the topological events and determines the Boolean result of a topological predicate or the kind of topological predicate. This paper puts an emphasis on the exploration phase and in addition presents a first, simple evaluation method.

*This work was partially supported by the National Science Foundation (NSF) under grant number NSF-CAREER-IIS-0347574.

1 Introduction

Topological predicates (like *overlap*, *meet*, *inside*) between spatial objects (like points, lines, regions) have always been a main area of research on spatial data handling, reasoning, and query languages in a number of disciplines like artificial intelligence, linguistics, robotics, and cognitive science. They characterize the relative position between two (or more) objects in space, deliberately exclude any consideration of quantitative, metric measures like distance or direction measures, and are associated with notions like adjacency, coincidence, connectivity, inclusion, and continuity. In particular, they support the design of suitable query languages for spatial data retrieval and analysis in geographical information systems and spatial database systems. The focus of this research has been on the conceptual design of and reasoning with these predicates as well as on strategies for avoiding unnecessary or repetitive predicate evaluations. The two central conceptual approaches, upon which almost all publications in this field have been based and which have produced very similar results, are the 9-*intersection model* [17] and the *RCC model* [11]. Until recently, topological predicates have only been defined for *simple* spatial objects. In this article, we are especially interested in topological predicates for *complex* spatial objects, as they have been recently specified in [45] on the basis of the 9-intersection model.

In contrast to the large amount of conceptual work, implementation issues for topological predicates have been widely neglected. Since topological predicates are *expensive predicates* that cannot be evaluated in constant time, the strategy in query plans has consequently been to avoid their computation. The extensive work on spatial index structures as a filtering technique in query processing is an important example of this strategy. It aims at identifying a hopefully small collection of candidate pairs of spatial objects that could possibly fulfil the predicate of interest and at excluding a large collection of pairs of spatial objects that definitely cannot satisfy the predicate. The main reason for neglecting the implementation issue of topological predicates in the literature is probably the simplifying view that some plane-sweep [38] algorithm is sufficient to implement topological predicates. Certainly, the plane sweep paradigm plays an important role for the implementation of these predicates, but there are at least three aspects that make such an implementation much more challenging. A first aspect refers to the details of the plane sweep itself. Issues are whether each topological predicate implementation necessarily requires an own, tailored plane sweep algorithm, how the plane sweep processes *complex* instead of *simple* spatial objects, whether spatial objects have been preprocessed in the sense that their intersections have been computed in advance (e.g., by employing a *realm-based* approach [27]), how intersections are handled, and what kind of information the plane sweep must output so that this information can be leveraged for predicate evaluation. A second aspect is that the number of topological predicates increases to a large extent with the transition from simple to complex spatial objects [45]. The two implementation alternatives of a single, specialized algorithm for each predicate or a single algorithm for all predicates with an exhaustive case analysis are error-prone, inappropriate, and thus unacceptable options from a correctness and a performance point of view. A third aspect deals with the kind of query posed. This has impact on the evaluation process. Given two objects A and B of any complex spatial data type *point2D*, *line2D*, or *region2D* [41], we can pose at least two kinds of topological queries: (1) "Do A and B satisfy the topological predicate p?" and (2) "What is the topological predicate p between A and B?". Only query 1 yields a Boolean value, and we call it hence a *verification query*. Query 2 returns a predicate (name), and we call it hence a *determination query*.

The goal of this (and a follow-up) article is to develop and present efficient implementation strategies for topological predicates between all combinations of the three complex spatial data types *point2D*, *line2D*, and *region2D* within the framework of the spatial algebra SPAL2D. We distinguish two phases of predicate execution: In an *exploration phase*, a plane sweep scans a given configuration of two spatial objects, detects all *topological events* (like intersections), and records them in so-called *topological feature vectors*. These vectors serve as input for the *evaluation phase* which analyzes these topological data and determines the Boolean result of a topological predicate (query 1) or the kind of topological predicate (query 2). This

paper puts an emphasis on the exploration phase and in addition presents a simple evaluation method. A follow-up article [37] deals in detail with improved and efficient methods for the evaluation phase. The two-phase approach provides a direct and sound interaction and synergy between conceptual work (9-intersection model) and implementation (algorithmic design).

The special goals of the exploration phase explain the introduction of topological feature vectors. The design of individualized, ad hoc algorithms for each topological predicate of each spatial data type combination turns out to be very cumbersome and error-prone and raises correctness issues. To avoid these problems, we propagate a systematic exploration approach which identifies all topological events that are possible for a particular type combination. These possible topological events are stored in two precisely defined topological feature vectors (one for each argument object of the topological predicate) whose specifications are characteristic and thus unique for each type combination. However, the specification of the topological feature vector of a particular spatial data type is different in different type combinations. The exploration approach leads to very reliable and robust predicate implementations and forms a sound basis for the subsequent evaluation phase. Further goals of the exploration phase are the treatment of complex and not only simple spatial objects and an integrated handling of general and realm-based spatial objects.

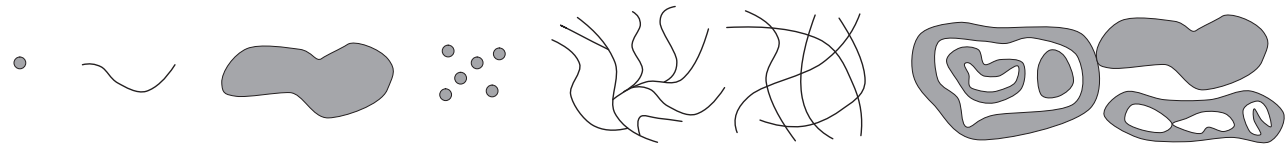
Section 2 discusses related work about spatial data types as well as available design and implementation concepts for topological predicates. In Section 3, we present the data structures used for all spatial data types in SPAL2D. Section 4 sketches some basic algorithmic concepts needed for the exploration algorithms. Section 5 deals with the exploration phase for collecting topological information on the basis of the plane sweep paradigm. It introduces a collection of *exploration algorithms*, one for each type combination. The algorithms extract the needed topological information from a given scene of two spatial objects and determine the topological feature vectors that are specific to each type combination. We also determine the runtime complexity of each algorithm in Big-O notation. Section 6 gives a first, simple evaluation method leveraging topological feature vectors. In Section 7, we provide a brief overview of the implementation environment and present our testing strategy of the topological feature vectors. Since we are not aware of any other, published implementation description of topological predicates to compare with, we do not provide an empirical performance analysis. Finally, Section 8 draws some conclusions and discusses future work.

2 Related Work

In this section we present related work on spatial objects as the operands of topological predicates (Section 2.1), sketch briefly the essential conceptual models for topological predicates (Section 2.2), and discuss implementation aspects of topological predicates (Section 2.3).

2.1 Spatial Objects

In the past, numerous data models and query languages for spatial data have been proposed with the aim of formulating and processing spatial queries in databases and GIS (e.g., [15, 25, 27, 36, 40, 41]). *Spatial data types* (see [41] for a survey) like *point*, *line*, or *region* are the central concept of these approaches. They provide fundamental abstractions for modeling the structure of geometric entities, their relationships, properties, and operations. Topological predicates operate on instances of these data types, called *spatial objects*. Until recently, these predicates have only been defined for simple object structures like single points, continuous lines, and simple regions (Figure 1(a)-(c)). However, it is broad consensus that these simple geometric structures are inadequate abstractions for real spatial applications since they are insufficient to cope with the variety and complexity of geographic reality. Consequently, universal and versatile type specifications are needed for (more) complex spatial objects so that they are usable in many different applications. With regard to *complex points* (Figure 1(d)), we allow finite collections of single points as point objects (e.g., to gather the positions of all lighthouses in the U.S.). With regard to *complex lines* (Figure 1(e)), we permit



(a) (b) (c) (d) (e) (f)

Figure 1: Examples of a simple point object (a), a simple line object (b), a simple region object (c), a complex point object (d), a complex line object (e), and a complex region object (f).

arbitrary, finite collections of one-dimensional curves, i.e., spatially embedded networks possibly consisting of several disjoint connected components, as line objects (e.g., to model the ramifications of the Nile Delta). With regard to *complex regions* (Figure 1(f)), the two main extensions relate to separations of the exterior (holes) and to separations of the interior (multiple components). For example, countries (like Italy) can be made up of multiple components (like the mainland and the offshore islands) and can have holes (like the Vatican). From a formal point of view, spatial data types should be closed under the geometric operations *union*, *intersection*, and *difference*. This is guaranteed for complex but not for simple spatial data types. Our formal specification of complex spatial data types given in [45] generalizes the definitions that can be found in the literature, rests on point set theory and point set topology [23], and is the basis of our spatial data type implementation. We call this specification the *abstract model* of our spatial type system SPAL2D.

So far, only a few models have been developed for complex spatial objects. The works in [6, 26, 27, 46] are the only formal approaches; they all share a number of structural features with our type specifications in [45]. The OpenGIS Consortium (OGC) has proposed geometric structures called *simple features* in their OGC Abstract Specification [33] and in their Geography Markup Language (GML) [34], which is an XML encoding for the transport and storage of geographic information. These geometric structures are similar to ours but only described informally and called *MultiPoint*, *MultiLineString*, and *MultiPolygon*. Another spatial data type specification is provided by ESRI's Spatial Database Engine (ArcSDE) [22]. It is also similar to ours but only informally described. Its importance comes from the fact that several database vendors have more or less integrated ArcSDE functionality into their spatial extension packages through extensibility mechanisms. Examples are the Informix Geodetic DataBlade [31], the Oracle Spatial Cartridge [35], and DB2's Spatial Extender [12].

Our main interest regarding complex spatial data types in this article relates to the development of effective geometric data structures that are able to support the efficient implementation of topological predicates. Descriptions of such data structures for spatial data types are rare. We partially borrow concepts from the implementation of the ROSE algebra [28] but generalize these concepts in the sense that we accommodate realm-based and general spatial objects. We describe our data structures in Section 3 and call this specification the *discrete model* of SPAL2D.

2.2 Conceptual Models for Topological Relationships

The conceptual study and determination of topological relationships has been an interdisciplinary research topic for a long time. Two fundamental approaches have turned out to be the starting point for a number of extensions and variations; these are the 9-*intersection model* [17], which is based on point set theory and point set topology, and the *RCC model* [11], which is based on spatial logic. Despite rather different foundations, both approaches come to very similar results. The implementation strategy described in this article relies heavily on the 9-intersection model. The reason is that we are able to create a direct link between the concepts of this model and our implementation approach, as we will see later. This enables us to prove the correctness of our implementation strategy.

$$\begin{pmatrix} A^\circ \cap B^\circ \neq \varnothing & A^\circ \cap \partial B \neq \varnothing & A^\circ \cap B^- \neq \varnothing \\ \partial A \cap B^\circ \neq \varnothing & \partial A \cap \partial B \neq \varnothing & \partial A \cap B^- \neq \varnothing \\ A^- \cap B^\circ \neq \varnothing & A^- \cap \partial B \neq \varnothing & A^- \cap B^- \neq \varnothing \end{pmatrix}$$

Figure 2: The 9-intersection matrix

Based on the 9-intersection model, a complete collection of mutually exclusive topological relationships can be determined for each combination of simple and recently also complex spatial data types. The model is based on the nine possible intersections of the boundary (∂A), the interior (A°), and the exterior (A^-) of a spatial object A with the corresponding components of another object B. Each intersection is tested with regard to the topologically invariant criteria of non-emptiness. The topological relationship between two spatial objects A and B can be expressed by evaluating the 3×3-matrix in Figure 2.

For this matrix, $2^9 = 512$ different configurations are possible from which only a certain subset makes sense depending on the *definition* and *combination* of the types of the spatial objects considered. For each combination of spatial types, this means that each of its predicates is associated with a unique intersection matrix so that all predicates are mutually exclusive and complete with regard to the topologically invariant criteria of emptiness and non-emptiness.

Topological relationships have been first investigated for simple spatial objects and especially for simple regions [9, 13, 16, 17]. For two simple regions, eight meaningful configurations have been identified which lead to the well known eight topological predicates called *disjoint*, *meet*, *overlap*, *equal*, *inside*, *contains*, *covers*, and *coveredBy*. A total of 33 topological relationships have been found for two simple lines [7, 14, 18]. Topological predicates between simple points are trivial: either two simple points are *disjoint* or they are *equal*. A simple point can be located *on* one of the endpoints of a simple line, *in* the interior of a simple line, or be *disjoint* from a simple line. For a simple point and a simple region, we obtain the three predicates *disjoint*, *meet*, and *inside*. For a simple line and a simple region, 19 topological relationships [19] can be distinguished.

The two works in [8, 21] are the first but restricted attempts to a definition of topological relationships on complex spatial objects. In [8] the so-called TRCR (Topological Relationships for Composite Regions) model only allows sets of disjoint simple regions without holes. Topological relationships between these composite regions are defined in an ad hoc manner and are not systematically derived from the underlying model. The work in [21] only considers topological relationships of simple regions with holes; multi-part regions are not permitted. A main problem of this approach is that it depends on the number of holes of the operand objects. In [45] with two precursors in [2] and [44] we have given a thorough, systematic, and complete specification of topological relationships for all combinations of complex spatial data types. Details about the determination process and prototypical drawings of spatial scenarios visualizing all topological relationships can be found in these papers. This approach, which is also based on the 9-intersection model, is the basis of our implementation. Table 1(b) shows the increase of topological predicates for complex objects compared to simple objects (Table 1(a)) and underpins the need for sophisticated and efficient predicate execution techniques.

2.3 Implementation Aspects of Topological Predicates

In queries, topological predicates usually appear as filter conditions of spatial selections and spatial joins. At least two main strategies can be distinguished for their processing. Either we attempt to avoid the execution of topological predicates since they are expensive predicates and cannot be executed in constant time, or we design sophisticated implementation methods for them. The latter strategy is always needed while the former strategy is worthwhile to do.

Avoidance strategies for expensive predicate executions can be leveraged at the algebraic level and at

	simple point	simple line	simple region
simple point	2	3	3
simple line	3	33	19
simple region	3	19	8

(a)

	complex point	complex line	complex region
complex point	5	14	7
complex line	14	82	43
complex region	7	43	33

(b)

Table 1: Numbers of topological predicates between two simple spatial objects (a) and between two complex spatial objects (b)

the physical level. At the algebraic level, consistency checking procedures examine the collection of topological relationships contained in a query for topological consistency by employing topological reasoning techniques [39]. Optimization minimizes the number of computations needed [10] and aims at eliminating topological relationships that are implied uniquely by composition [20]. At the physical level, several methods pursue the concept of avoiding unnecessary or repetitive predicate evaluations in query access plans. Examples of these methods are predicate migration [30], predicate placement [29], disjunctive query optimization [5], and approximation-based evaluation [4]. Another important method is the deployment of spatial index structures [24] to identify those candidate pairs of spatial objects that could possibly fulfil the predicate of interest. This is done by a filtering test on the basis of minimum bounding rectangles.

At some point, avoidance strategies are of no help anymore, and the application of physical predicate execution algorithms is required. So far, there has been no published research on the efficient implementation of topological predicates, their optimization, and their connection to the underlying theory. All three aspects are objectives of our work. We leverage the well known plane sweep paradigm [3] and go far beyond our own initial attempts in [42, 43]. These attempts extend the concept of the eight topological predicates for two simple regions to a concept and implementation of these eight predicates for complex regions.

The spatial data management and analysis packages mentioned in Section 2.1 like the ESRI ArcSDE, the Informix Geodetic DataBlade, the Oracle Spatial Cartridge, and DB2's Spatial Extender offer limited sets of named topological predicates for simple and complex spatial objects. But their definitions are vague and their implementations unpublished. The open source JTS Topology Suite [32] conforms to the simple features specification [33] of the Open GIS Consortium and implements topological predicates through *topology graphs*. A topology graph stores topology explicitly and contains labeled nodes and edges corresponding to the endpoints and segments of a spatial object's geometry. For each node and edge of a spatial object, one determines whether it is located in the interior, in the exterior, or on the boundary of another spatial object. Computing the topology graphs and deriving the 9-intersection matrix from them require quadratic time and quadratic space in terms of the nodes and edges of the two operand objects; our solution requires linearithmic (loglinear) time and linear space.

3 Data Structures

Our approach to the verification (query type 1) as well as the determination (query type 2) of a topological relationship requires a two-phase algorithm consisting of the *exploration phase* and the *evaluation phase*. Only the exploration phase makes use of the two spatial input objects. Each input object can be of type *point2D*, *line2D*, or *region2D*. In this section, we describe their data structures. Our data structure design and implementation is part of a new type system (i.e., algebra) for handling two-dimensional spatial data. In Section 3.1 we give a brief overview of this algebra, explain its main components, and briefly discuss some main requirements of data structure design in a database context. The following three sections describe several data structures needed for a topological predicate implementation. Section 3.2 introduces the

primitive types *poi2D* and *seg2D*. Section 3.3 explains the geometric component data type *halfsegment2D* as the basis of the spatial data type implementation. Finally, the data structures for the spatial data types *point2D*, *line2D*, and *region2D* (i.e., their discrete model) are presented in Section 3.4.

3.1 Data Structure Design in the Type System SPAL2D

Our implementation of topological predicates is embedded into the framework of a new type system for two-dimensional spatial data, called *Spatial Algebra 2D* (*SPAL2D* for short). Some major design goals of this algebra are high-level and general, complex spatial data types, sophisticated spatial operations and predicates, numerically stable and topologically consistent algorithms, and usability in a database and query language context. The algebra consists of four constituents, shown in Figure 3.

A central problem for the design of spatial data types is usually the underlying number system that usually fails to ensure numerically stable and topologically consistent geometric algorithms. Numerical stability incorporates the three aspects of correctness, robustness, and efficiency. Correctness means that the implementation should always yield the right geometric or topological result compared to reality. Robustness implies that all cases occurring in a geometric operation should be handled correctly. Efficiency emphasizes the requirement that the possible advantages of a stable algorithm should not be neutralized by time-consuming numerical computations. Therefore, at the lowest level, our number system RATIO provides a numerically stable rational number implementation, which enables exact geometric computation. The arithmetic RATIO provides a data type *rat* for rational numbers whose value representations can be of arbitrary, finite length and are only limited by main memory. This enables exact computation but entails the price of possibly longer number representations. From an implementation standpoint, the idea is to leverage existing, fast computer number systems and the operations defined on them.

The next higher level incorporates two-dimensional *robust geometric primitives* (*RGP2D*) that are based on RATIO and that implement coordinates as rational numbers. These primitives serve as elementary building blocks for all higher-level structures and contribute significantly to their behavior, performance, and robustness. RGP2D offers a primitive data type *poi2D* for single points, a data type *seg2D* for segments, and a type *mbb2D* for minimum bounding boxes together with corresponding operations and predicates. Those concepts that are needed for our predicate executions are described in Section 3.2.

The third level provides the two-dimensional *geometric component data types* (*GCDT2D*) *face2D*, *polygon2D*, *block2D*, and *halfsegment2D*. Objects of these types rest on objects from RGP types and represent the components from which spatial objects at the highest level are constructed. The first three data types are pure user concepts. A *polygon* is a closed areal figure given by its linear boundary representation. A *face* consists of an *outer polygon* and possibly one or more edge-disjoint inner polygons called *holes*; it is a constituent part of a complex region. A *block* is a finite collection of connected segments and a constituent

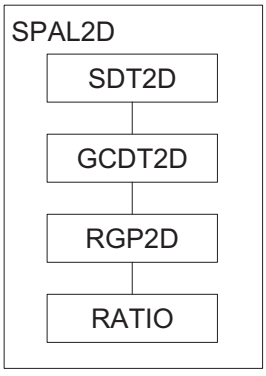


Figure 3: Architecture of the *Spatial Algebra 2D* (*SPAL2D*)

part of a complex line. The fourth data type is exclusively an implementation concept. For the exploration phase, we are only interested in this concept; we describe halfsegments in Section 3.3.

The highest level SDT2D offers the three *complex spatial data types point2D, line2D,* and *region2D* (see Section 2.1 for their intuitive description and [45] for their formal definition in the abstract model) together with a comprehensive collection of spatial operations and predicates. For the construction of complex spatial objects, SDT2D utilizes the three geometric component data types *face2D, polygon2D,* and *block2D*. The internal representation of complex spatial objects is based on the types *poi2D* and *halfsegment2D*. Since it is of particular interest for our predicate implementation, it is described in Section 3.4.

Since our goal is to plug our spatial algebra into a database system, the design and implementation of data structures is subject to other criteria than in other application contexts. In a database context, data structure design has to satisfy special requirements. A first aspect is that algorithms for different operations processing the same kind of data usually prefer different internal data representations in order to be as efficient as possible. In contrast to traditional work on algorithms, the focus here is not on finding the most efficient algorithm for each single problem (operation) together with a corresponding sophisticated data structure, but rather on considering the algebra as a whole and on reconciling the various requirements posed by different algorithms within a single data structure for each data type. Otherwise, the consequence would be considerable conversion costs between different data structures in main memory for the same data type. A second aspect is that the intended use in a database system implies that representations for data types should not employ pointer data structures in main memory but be embedded into compact storage areas (arrays), which can be efficiently transferred between main memory and disk. This avoids unnecessary and high costs for packing main memory data and unpacking external data.

In the following sections, we (only) introduce those data types and operations from RGP2D, GCDT2D, and SDT2D that are needed for the implementation of topological predicates.

3.2 Robust Geometric Primitive Types

All objects of robust geometric primitive types are stored in records. The type *poi2D* incorporates all single, two-dimensional points we can represent on the basis of our rational number system RATIO. That is, this type is defined as

$$poi2D = \{(x,y) \mid x,y \in rat\} \cup \{\varepsilon\}$$

The value ε represents *the empty object* and is an element of *all* data types. The empty object is needed to represent the case that an operation yields an "empty" result. Consider the case of a primitive operation that computes the intersection of two segments in a point. If both segments do not intersect at all or they intersect in a common segment, the result is empty and has to be represented.

Given two points $p,q \in poi2D$, we assume a predicate "=" ($p = q \Leftrightarrow p.x = q.x \ \wedge \ p.y = q.y$) and the lexicographic order relation "$<$" ($p < q \Leftrightarrow p.x < q.x \ \vee \ (p.x = q.x \ \wedge \ p.y < q.y)$).

The type *seg2D* includes all straight segments bounded by two endpoints. That is

$$seg2D = \{(p,q) \mid p,q \in poi2D, p < q\} \cup \{\varepsilon\}$$

The order defined on the endpoints normalizes segments and provides for a unique representation. This enables us to speak of a *left end point* and a *right end point* of a segment.

The predicates $on, in : poi2D \times seg2D \rightarrow bool$ check whether a point is located on a segment including and excluding its endpoints respectively. The predicates $poiIntersect, segIntersect : seg2D \times seg2D \rightarrow bool$ test whether two segments intersect in a point or a segment respectively. The predicates *collinear*, *equal*, *disjoint*, *meet* : $seg2D \times seg2D \rightarrow bool$ determine whether two segments lie on the same infinite line, are identical, do not share any point, and touch each other in exactly one common endpoint respectively. The

function $len : seg2D \rightarrow real$ computes the length of a segment. The type *real* is our own approximation type for the real numbers and implemented on the basis of type *rat*. The operation $poiIntersection : seg2D \times seg2D \rightarrow poi2D$ returns the intersection point of two segments.

The type *mbb2D* comprises all minimum bounding boxes, i.e., axis-parallel rectangles. It is defined as

$$mbb2D = \{(p,q) \mid p,q \in poi, p.x < q.x, p.y < q.y\} \cup \{\varepsilon\}$$

Here, the predicate $disjoint : mbb2D \times mbb2D \rightarrow bool$ checks whether two minimum bounding boxes are disjoint; otherwise, they interfere with each other.

3.3 The Geometric Component Data Type *halfsegment2D*

Halfsegments are the basic implementation components of objects of the spatial data types *line2D* and *region2D*. A *halfsegment*, which is stored in a record, is a hybrid between a point and segment. That is, it has features of both geometric structures; each feature can be inquired on demand. We define the set of all halfsegments as the component data type

$$halfsegment2D = \{(s,d) \mid s \in seg2D - \{\varepsilon\},\ d \in bool\}$$

For a halfsegment $h = (s,d)$, the Boolean flag d emphasizes one of the segment's end points, which is called the *dominating point* of h. If $d = true$ ($d = false$), the left (right) end point of s is the dominating point of h, and h is called *left* (*right*) *halfsegment*. Hence, each segment s is mapped to two halfsegments $(s, true)$ and $(s, false)$. Let dp be the function which yields the dominating point of a halfsegment.

The representation of *line2D* and *region2D* objects requires an order relation on halfsegments. For two distinct halfsegments h_1 and h_2 with a common end point p, let α be the enclosed angle such that $0° < \alpha \leq 180°$. Let a predicate *rot* be defined as follows: $rot(h_1, h_2)$ is *true* if, and only if, h_1 can be rotated around p through α to overlap h_2 in counterclockwise direction. This enables us now to define a complete order on halfsegments. For two halfsegments $h_1 = (s_1, d_1)$ and $h_2 = (s_2, d_2)$ we obtain:

$$\begin{aligned} h_1 < h_2 \quad \Leftrightarrow \quad & dp(h_1) < dp(h_2) \vee && (1)\\ & (dp(h_1) = dp(h_2) \wedge ((\neg d_1 \wedge d_2) \vee && (2a)\\ & \qquad (d_1 = d_2 \wedge rot(h_1, h_2)) \vee && (2b)\\ & \qquad (d_1 = d_2 \wedge collinear(s_1, s_2) \wedge len(s_1) < len(s_2)))) && (3) \end{aligned}$$

Examples of the order relation on halfsegments are given in Figure 4. Case 1 is exclusively based on the (x,y)-lexicographical order on dominating points. In the other cases the dominating points of h_1 and h_2 coincide. Case 2a deals with the situation that h_1 is a right halfsegment and h_2 is a left halfsegment. Case 2b handles the situation that h_1 and h_2 are either both left halfsegments or both right halfsegments so that the angle criterion is applied. Finally, case 3 treats the situation that h_1 and h_2 are collinear. Two halfsegments $h_1 = (s_1, d_1)$ and $h_2 = (s_2, d_2)$ are equal if, and only if, $s_1 = s_2$ and $d_1 = d_2$.

We will also need an order relation between a point $v \in poi2D$ and a halfsegment $h \in halfsegment2D$. We define $v < h \Leftrightarrow v < dp(h)$ and $v = h \Leftrightarrow v = dp(h)$. This shows the hybrid nature of halfsegments having point and segment features.

3.4 Spatial Data Types

The general, internal representation structure for *point2D*, *line2D*, and *region2D* objects of the discrete model is that of an *information part* of fixed length, followed by an *ordered sequence of elements* of variable length. Information part and sequence are stored in a compact storage area, i.e., an array. Depending on the type, the representations differ in the kind of elements and in the contents of the information parts.

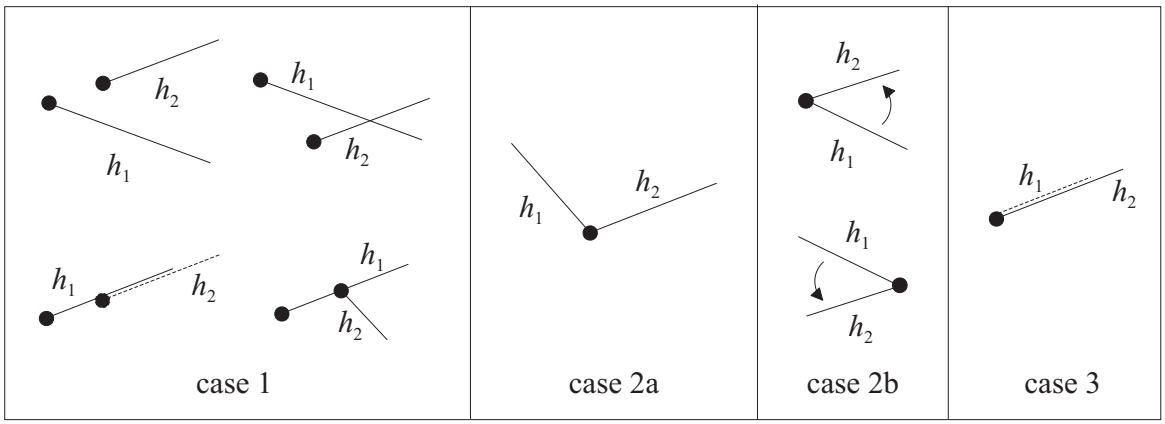


Figure 4: Examples of the order relation on halfsegments: $h_1 < h_2$

The element type is *poi2D* for *point2D* objects, *halfsegment2D* for *line2D* objects, and *attributed halfsegments* (see below) for *region2D* objects. Ordered sequences are selected as representation structures, since they directly and efficiently support parallel traversals (Section 4.1) and the plane sweep paradigm (see Section 4.3). The information part keeps the number of points for *point2D* objects, the number of halfsegments, the total length of all segments, etc. for *line2D* objects, and the number of halfsegments, the area, the perimeter, etc. for *region2D* objects. In general, the information part contains information that can be easily computed during the construction of a spatial object. Corresponding operations can later benefit from this information and answer in constant (instead of linear) time. In our context, the information part is not needed so that we neglect it in the following. Another simplification relates to the sequence structure of *line2D* and *region2D* objects. The sequence structures also support the block components of a *line2D* object and the face components of a *region2D* object. For this purpose, halfsegments belonging to the same block as well as attributed halfsegments belonging to the same face are linked with each other in the halfsegment sequence by array indices. We will also neglect this aspect, since we do not need block and face components.

We now have a closer look at the type definitions. Let $\mathbb{N}_0 := \mathbb{N} \cup \{0\}$. The spatial data type *point2D* is defined as

$$point2D = \{\langle p_1, \ldots, p_n \rangle \mid n \in \mathbb{N}_0, \forall\, 1 \leq i \leq n : p_i \in poi2D - \{\varepsilon\}, \forall\, 1 \leq i < n : p_i < p_{i+1}\}$$

Since $n = 0$ is allowed, the empty sequence $\langle\rangle$ represents the empty *point2D* object.

The spatial data type *line2D* is defined as

$$\begin{aligned} line2D = \{\langle h_1, \ldots, h_{2n} \rangle \mid\ & \text{(i)}\ \ n \in \mathbb{N}_0 \\ & \text{(ii)}\ \ \forall\, 1 \leq i \leq 2n : h_i \in halfsegment2D \\ & \text{(iii)}\ \ \forall\, h_i = (s_i, d_i) \in \{h_1, \ldots, h_{2n}\}\ \exists h_j = (s_j, d_j) \in \{h_1, \ldots, h_{2n}\}, \\ & \qquad 1 \leq i < j \leq 2n : equal(s_i, s_j)\ \wedge\ d_i = \neg d_j \\ & \text{(iv)}\ \ \forall\, 1 \leq i < 2n : h_i < h_{i+1} \\ & \text{(v)}\ \ \forall\, h_i = (s_i, d_i), h_j = (s_j, d_j) \in \{h_1, \ldots, h_{2n}\}, i \neq j : equal(s_i, s_j) \vee \\ & \qquad disjoint(s_i, s_j)\ \vee\ meet(s_i, s_j)\} \end{aligned}$$

The value n is equal to the number of segments of a *line2D* object. Since each segment is represented by a left and a right halfsegment (condition (iii)), a *line2D* object has $2n$ halfsegments. Since $n = 0$ is allowed (condition (i)), the empty sequence $\langle\rangle$ represents the empty *line2D* object. Condition (iv) expresses that a *line2D* object is given as an *ordered halfsegment sequence*. Condition (v) requires that the segments of two distinct halfsegments are either equal (this only holds for the left and right halfsegments of a segment), disjoint, or meet.

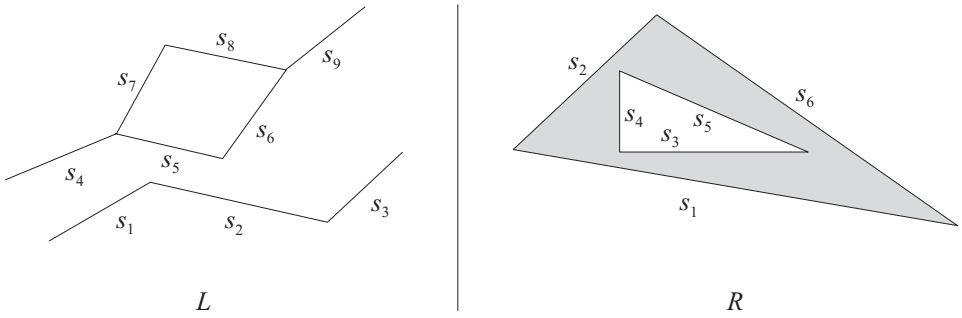


Figure 5: A *line2D* object L and a *region2D* object R

The internal representation of *region2D* objects is similar to that of *line2D* objects. But for each halfsegment, we also need the information whether the interior of the region is above/left or below/right of it. We denote such a halfsegment as *attributed halfsegment*. Each element of the sequence consists of a halfsegment that is augmented by a Boolean flag *ia* (for "interior above"). If $ia = true$ holds, the interior of the region is above or, for vertical segments, left of the segment. The spatial data type *region2D* is defined as

$$
\begin{aligned}
\mathit{region2D} = \{\langle (h_1, ia_1), \ldots, (h_{2n}, ia_{2n})\rangle \mid\ & \text{(i)} \quad n \in \mathbb{N}_0 \\
& \text{(ii)} \quad \forall\, 1 \le i \le 2n : h_i \in \mathit{halfsegment2D}, ia_i \in \mathit{bool} \\
& \text{(iii)} \quad \forall\, h_i = (s_i, d_i) \in \{h_1, \ldots, h_{2n}\} \\
& \qquad \exists\, h_j = (s_j, d_j) \in \{h_1, \ldots, h_{2n}\}, 1 \le i < j \le 2n : \\
& \qquad s_i = s_j \ \wedge\ d_i = \neg d_j \ \wedge\ ia_i = ia_j \\
& \text{(iv)} \quad \forall\, 1 \le i < 2n : h_i < h_{i+1} \\
& \text{(v)} \quad \text{"additional topological constraints"}\}
\end{aligned}
$$

Condition (v) refers to the additional topological constraints that all faces must be edge-disjoint from each other and that for each face its holes must be located inside its outer polygon, be edge-disjoint to the outer polygon, and be edge-disjoint among each other. These constraints are checked during the construction of a *region2D* object. We mention them for reasons of completeness and assume their satisfaction.

As an example, Figure 5 shows a *line2D* object L (with two blocks) and a *region2D* object R (with a single face containing a hole). Both objects are annotated with segment names s_i. We now determine the halfsegment sequences of L and R and let $h_i^l = (s_i, true)$ and $h_i^r = (s_i, false)$ denote the left halfsegment and right halfsegment of a segment s_i respectively. For L we obtain the ordered halfsegment sequence

$$L = \langle h_4^l, h_1^l, h_4^r, h_5^l, h_7^l, h_1^r, h_2^l, h_7^r, h_8^l, h_5^r, h_6^l, h_8^r, h_6^r, h_9^l, h_2^r, h_3^l, h_9^r, h_3^r\rangle$$

For R we obtain the following ordered sequence of attributed halfsegments ($t \equiv true$, $f \equiv false$):

$$R = \langle (h_1^l, t), (h_2^l, f), (h_3^l, f), (h_4^l, t), (h_4^r, t), (h_5^l, t), (h_2^r, f), (h_6^l, f), (h_5^r, t), (h_3^r, f), (h_6^r, f), (h_1^r, t)\rangle$$

Since inserting a halfsegment at an arbitrary position needs $O(n)$ time, in our implementation we use an AVL-tree embedded into an array whose elements are linked in halfsegment order. An insertion then requires $O(\log n)$ time.

If we take into account that the segments of a *line2D* object as well as the segments of a *region2D* object are not allowed to intersect each other or touch each other in their interiors according to their type specifications, the definition of the order relation on halfsegments (Section 3.3) seems to be too intricate. If, in Figure 4, we take away all subcases of case 1 except for the upper left subcase as well as case 3, the restricted order relation can already be leveraged for complex lines and complex regions. In case that all

spatial objects of an application space are defined over the same *realm*[1] [26, 41], the restricted order relation can also be applied for a parallel traversal of the sequences of two (or more) *realm-based line2D* or *region2D* objects. But only in the general case of intersecting spatial objects, the full order relation on halfsegments is needed for a parallel traversal of the objects' halfsegment sequences.

4 Basic Algorithmic Concepts

In this section, we describe three algorithmic concepts that serve as the foundation of the *exploration algorithms* in Section 5. These concepts are the *parallel object traversal* (Section 4.1), *overlap numbers* (Section 4.2), and the *plane sweep* paradigm (Section 4.3). Parallel object traversal and overlap numbers are employed during a plane sweep. We will not describe these concepts in full detail here, since they are well known methods in Computational Geometry [3] and spatial databases [28]. Instead, we will focus on the specialties of these concepts in our setting. This includes a smoothly integrated handling of general and realm-based spatial objects. An objective of this section is also to introduce a number of auxiliary operations and predicates that make the description of the exploration algorithms later much more comprehensible.

4.1 Parallel Object Traversal

For a plane sweep, the representation elements (points or segments) of the spatial operand objects have usually to be merged together and sorted afterwards according to some order relation (e.g., the order on *x*-coordinates). This initial merging and sorting is rather expensive and requires $O(n \log n)$ time, if n is the number of representation elements of both operand objects. Our approach avoids this initial sorting, since the representation elements of *point2D*, *line2D*, and *region2D* objects are already stored in the order we need (point order or halfsegment order). We also do not have to merge the object representations, since we can deploy a *parallel object traversal* that allows us to traverse the point or halfsegment sequences of both operand objects in parallel. Hence, by employing a cursor on both sequences, it is sufficient to check the point or halfsegment at the current cursor positions of both sequences and to take the lower one with respect to the point order or halfsegment order for further computation.

If the operand objects have already been intersected with each other, like in the realm case [26], the parallel object traversal has only to operate on two *static* point or halfsegment sequences. But in the general case, intersections between both objects can exist and are detected during the plane sweep. A purely static sequence structure is insufficient in this case, since detected intersections have to be stored and handled later during the plane sweep. In order to avoid a change of the original object representations, which would be very expensive and only temporarily needed, each object is associated with an additional and temporary *dynamic* sequence, which stores newly detected points or halfsegments of interest. Hence, our parallel object traversal has to handle a static and a dynamic sequence part for each operand object and thus four instead of two point or halfsegment sequences. It returns the smallest point or halfsegment from the four current cursor positions. We will give an example of the parallel object traversal when we discuss our plane sweep approach (Section 4.3).

To simplify the description of this parallel scan, two operations are provided. Let $O_1 \in \alpha$ and $O_2 \in \beta$ with $\alpha, \beta \in \{point2D, line2D, region2D\}$. The operation *select_first*(O_1, O_2, *object*, *status*) selects the first point or halfsegment of each of the operand objects O_1 and O_2 and positions a logical pointer on both of them. The parameter *object* with a possible value out of the set {*none*, *first*, *second*, *both*} indicates which of the two object representations contains the smaller point or halfsegment. If the value of *object* is *none*, no

[1] A *realm* provides a discrete geometric basis for the construction of spatial objects and consists of a finite set of points and *non-intersecting* segments, called *realm objects*. That is, a segment inserted into the realm is intersected and split according to a special strategy with all realm objects. All spatial objects like complex points, complex lines, and complex regions are then defined in terms of these realm objects. Hence, all spatial objects defined over the same realm become aquainted with each other beforehand.

point or halfsegment is selected, since O_1 and O_2 are empty. If the value is *first* (*second*), the smaller point or halfsegment belongs to O_1 (O_2). If it is *both*, the first point or halfsegment of O_1 and O_2 are identical. The parameter *status* with a possible value out of the set {*end_of_none*, *end_of_first*, *end_of_second*, *end_of_both*} describes the state of both object representations. If the value of *status* is *end_of_none*, both objects still have points or halfsegments. If it is *end_of_first* (*end_of_second*), O_1 (O_2) is exhausted. If it is *end_of_both*, both object representations are exhausted.

The operation *select_next*(O_1, O_2, *object*, *status*), which has the same parameters as *select_first*, searches for the next smallest point or halfsegment of O_1 and O_2. Two points (halfsegments) are compared with respect to the lexicographic (halfsegment) order. For the comparison between a point and a halfsegment, the dominating point of the halfsegment and hence the lexicographic order is used (see Section 3.3). If before this operation *object* was equal to *both*, *select_next* moves forward the logical pointers of both sequences; otherwise, if *object* was equal to *first* (*second*), it only moves forward the logical pointer of the first (second) sequence. In contrast to the first operation, which only has to consider the static sequence part of an object, this operation also has to check the dynamic sequence part of each object. Both operations together allow one to scan in linear time two object representations like one ordered sequence.

4.2 Overlap Numbers

The concept of *overlap numbers* is exclusively needed for the computation of the topological relationships between two *region2D* objects. The reason is that we have to find out the degree of overlapping of region parts. Overlap numbers serve as part of the "interface" between the discrete and abstract specifications of complex regions. For a complex region F, we have to distinguish its discrete specification F_d as a sequence of attributed halfsegments and its abstract specification F_a as an infinite point set. In the abstract model, a point in the plane has overlap number k if k regions contain this point. For two regions F_a and G_a, a point p obtains the overlap number 2, if, and only if, $p \in F_a$ and $p \in G_a$. It obtains the overlap number 1, if, and only if, either $p \in F_a$ and $p \in G_a^-$, or $p \in F_a^-$ and $p \in G_a$. Otherwise, its overlap number is 0. In the discrete model, where we cannot explicitly represent infinite point sets, we employ finite boundary representations, i.e., attributed halfsegments. Since a segment s_h of an attributed halfsegment $(h, ia_h) = ((s_h, d_h), ia_h)$ of a region F_d separates space into two parts, an interior and an exterior one, during a plane sweep each such segment is associated with a *segment class* which is a pair (m/n) of overlap numbers, a lower (or right) one m and an upper (or left) one n ($m, n \in \mathbb{N}_0$). The lower (upper) overlap number indicates the number of overlapping *region2D* objects below (above) the segment. In this way, we obtain a *segment classification* of two *region2D* objects and speak about (m/n)-segments. Obviously, $0 \leq m, n \leq 2$ holds. Of the nine possible combinations only seven describe valid segment classes. This is because a $(0/0)$-segment contradicts the definition of a complex *region2D* object, since then at least one of both regions would have two holes or an outer cycle and a hole with a common border segment. Similarly, $(2/2)$-segments cannot exist, since then at least one of the two regions would have a segment which is common to two outer cycles of the object. Hence, possible (m/n)-segments are $(0/1)$-, $(0/2)$-, $(1/0)$-, $(1/1)$-, $(1/2)$-, $(2/0)$-, and $(2/1)$-segments. Figure 6 gives an example.

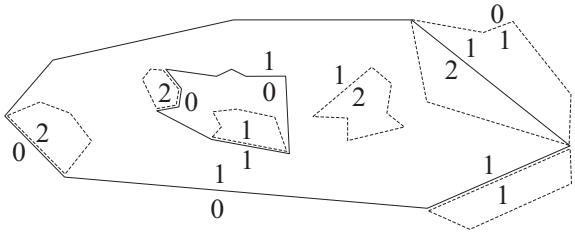


Figure 6: Example of the segment classification of two *region2D* objects

4.3 Plane Sweep

The plane sweep technique [3, 38] is a well known algorithmic scheme in Computational Geometry. Its central idea is to reduce a two-dimensional geometric problem to a simpler one-dimensional geometric problem. A vertical *sweep line* traversing the plane from left to right stops at special *event points* which are stored in a queue called *event point schedule*. The event point schedule must allow one to insert new event points discovered during processing; these are normally the initially unknown intersections of line segments. The state of the intersection of the sweep line with the geometric structure being swept at the current sweep line position is recorded in vertical order in a data structure called *sweep line status*. Whenever the sweep line reaches an event point, the sweep line status is updated. Event points which are passed by the sweep line are removed from the event point schedule. Note that, in general, an efficient and fully dynamic data structure is needed to represent the event point schedule and that, in many plane-sweep algorithms, an initial sorting step is needed to produce the sequence of event points in (x,y)-lexicographical order.

In our case, the event points are either the points of the static point sequences of *point2D* objects or the (attributed) halfsegments of the static halfsegment sequences of *line2D* (*region2D*) objects. This especially holds and is sufficient for the realm case. In addition, in the general case, new event points are determined during the plane sweep as intersections of line segments; they are stored as points or halfsegments in the dynamic sequence parts of the operand objects and are needed only temporarily for the plane sweep. As we have seen in Section 4.1, the concepts of point order, halfsegment order, and parallel object traversal avoid an expensive initial sorting at the beginning of the plane sweep. We use the operation *get_event* to provide the element to which the logical pointer of a point or halfsegment sequence is currently pointing. The Boolean predicate *look_ahead* tests whether the dominating points of a given halfsegment and the next halfsegment after the logical pointer of a given halfsegment sequence are equal.

Several operations are needed for managing the sweep line status. The operation *new_sweep* creates a new, empty sweep line status. If a left (right) halfsegment of a *line2D* or *region2D* object is reached during a plane-sweep, the operation *add_left* (*del_right*) stores (removes) its segment component into (from) the segment sequence of the sweep line status. The predicate *coincident* checks whether the just inserted segment partially coincides with a segment of the other object in the sweep line status. The operation *set_attr* (*get_attr*) sets (gets) an attribute for (from) a segment in the sweep line status. This attribute can be either a Boolean value indicating whether the interior of the region is above the segment or not (the "Interior Above" flag), or it can be an assigned segment classification. The operation *get_pred_attr* yields the attribute from the predecessor of a segment in the sweep line status. The operation *pred_exists* (*common_point_exists*) checks whether for a segment in the sweep line status a predecessor according to the vertical y-order (a neighbored segment of the *other* object with a common end point) exists. The operation *pred_of_p* searches the nearest segment below a given point in the sweep line status. The predicate *current_exists* tests whether such a segment exists. The predicate *poi_on_seg* (*poi_in_seg*) checks whether a given point lies *on* (*in*) any segment of the sweep line status.

Intersections of line segments stemming from two *lines2D* objects, two *region2D* objects, or a *line2D* object and a *region2D* object are of special interest, since they indicate topological changes. If two segments of two *line2D* objects intersect, this can, e.g., indicate a proper intersection, or a meeting situation between both segments, or an overlapping of both segments. If a segment of a *line2D* object intersects a segment of a *region2D* object, the former segment can, e.g., "enter" the region, "leave" the region, or overlap with the latter segment. Overlap numbers can be employed here to determine entering and leaving situations. If segments of two *region2D* objects intersect, this can, e.g., indicate that they share a common area and/or a common boundary. In this case, intersecting line segments have especially an effect on the overlap numbers of the segments of both *region2D* objects. In Section 4.2 we have tacitly assumed that any two segments from both *region2D* objects are either disjoint, or equal, or meet solely in a common end point. Only if these topological constraints are satisfied, we can use the concepts of overlap numbers and segment classes

for a plane sweep. But the general case in particular allows intersections. Figure 7 shows the problem of segment classes for two intersecting segments. The segment class of s_1 [s_2] left of the intersection point is $(0/1)$ [$(1/2)$]. The segment class of s_1 [s_2] right of the intersection point is $(1/2)$ [$(0/1)$]. That is, after the intersection point, seen from left to right, s_1 and s_2 exchange their segment classes. The reason is that the topology of both segments changes. Whereas, to the left of the intersection, s_1 (s_2) is outside (inside) the region to which s_2 (s_1) belongs, to the right of the intersection, s_1 (s_2) is inside (outside) the region to which s_2 (s_1) belongs.

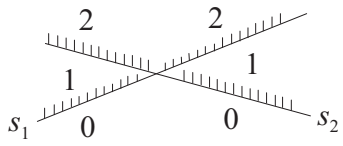


Figure 7: Changing overlap numbers after an intersection.

In order to be able to make the needed topological detections and to enable the use of overlap numbers for two general regions, in case that two segments from two different regions intersect, partially coincide, or touch each other within the interior of a segment, we pursue a splitting strategy that is executed during the plane sweep "on the fly". If segments intersect, they are temporarily split at their common intersection point so that each of them is replaced by two segments (i.e., four halfsegments) (Figure 8a). If two segments partially coincide, they are split each time the endpoint of one segment lies inside the interior of the other segment. Depending on the topological situations, which can be described by Allen's thirteen basic relations on intervals [1], each of the two segments either remains unchanged or is replaced by up to three segments (i.e., six halfsegments). From the thirteen possible relations, eight relations (four pairs of symmetric relations) are of interest here (Figure 8b). If an endpoint of one segment touches the interior of the other segment, the latter segment is split and replaced by two segments (i.e., four halfsegments) (Figure 8c).

This splitting strategy is numerically stable and thus feasible from an implementation standpoint, since RATIO ensures correctness, numerical robustness, and hence topological consistency of intersection operations. Intersecting and touching points can be *exactly* computed, lead to representable points with rational coordinates provided by RATIO, and are thus precisely located on the intersecting or touching segments.

However, as indicated before, the splitting of segments entails some algorithmic effort. On the one hand, we want to keep the halfsegment sequences of the *line2D* and *region2D* objects unchanged, since their update is expensive and only temporarily needed for the plane sweep. On the other hand, the splitting of halfsegments has an effect on these sequences. As a compromise, for each *line2D* or *region2D* object, we maintain its "static" representation, and the halfsegments obtained by the splitting process are stored in an additional "dynamic" halfsegment sequence. The dynamic part is also organized as an AVL tree which

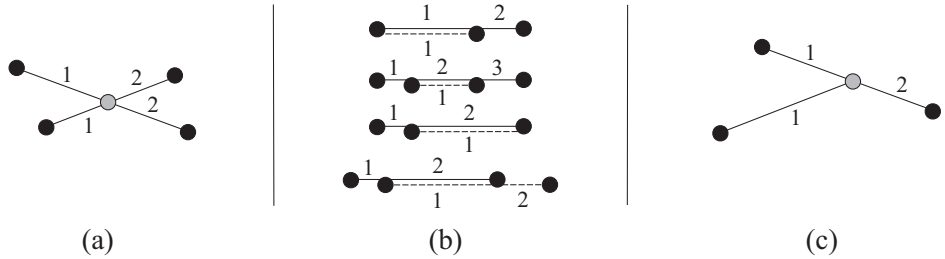


Figure 8: Splitting of two intersecting segments (a), two partially coinciding segments (without symmetric counterparts) (b), and a segment whose interior is touched by another segment (c). Digits indicate part numbers of segments after splitting.

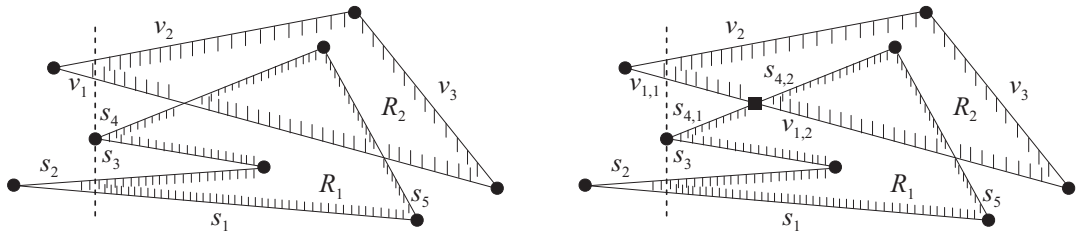


Figure 9: Sweep line status before the splitting (s_4 to be inserted) (a) and after the splitting (b). The vertical dashed line indicates the current position of the sweep line.

is embedded in an array and whose elements are linked in sequence order. Assuming that k splitting points are detected during the plane sweep, we need $O(k)$ additional space, and to insert them requires $O(k \log k)$ time. After the plane sweep, this additional space is released.

To illustrate the splitting process in more detail, we consider two *region2D* objects R_1 and R_2. In general, we have to deal with the three cases in Figure 8. We first consider the case that the plane sweep detects an intersection. This leads to a situation like in Figure 9a. The two static and the two dynamic halfsegment sequences of R_1 and R_2 are shown in Table 2. Together they form the *event point schedule* of the plane sweep and are processed by a parallel object traversal. Before the current position of the sweep line (indicated by the vertical dashed line in Figure 9), the parallel object traversal has already processed the attributed halfsegments $(h^l_{s_1}, t)$, $(h^l_{s_2}, f)$, $(h^l_{v_1}, t)$, and $(h^l_{v_2}, f)$ in this order. At the current position of the sweep line, the parallel object traversal encounters the halfsegments $(h^l_{s_3}, t)$ and $(h^l_{s_4}, f)$. For each left halfsegment visited, the corresponding segment is inserted into the sweep line status according to the y-coordinate of its dominating point and checked for intersections with its direct upper and lower neighbors. In our example, the insertion of s_4 leads to an intersection with its upper neighbor v_1. This requires *segment splitting*; we split v_1 into the two segments $v_{1,1}$ and $v_{1,2}$ and s_4 into the two segments $s_{4,1}$ and $s_{4,2}$. In the sweep line status, we have to replace v_1 by $v_{1,1}$ and s_4 by $s_{4,1}$ (Figure 9b). The new halfsegments $(h^r_{s_{4,1}}, f)$, $(h^l_{s_{4,2}}, f)$, and $(h^r_{s_{4,2}}, f)$ are inserted into the dynamic halfsegment sequence of R_1. Into the dynamic halfsegment sequence of R_2, we insert the halfsegments $(h^r_{v_{1,1}}, t)$, $(h^l_{v_{1,2}}, t)$, and $(h^r_{v_{1,2}}, t)$. We need not store the two halfsegments $(h^l_{s_{4,1}}, f)$ and $(h^l_{v_{1,1}}, t)$ since they refer to the "past" and have already been processed.

On purpose we have accepted a little inconsistency in this procedure, which can fortunately be easily controlled. Since, for the duration of the plane sweep, s_4 (v_1) has been replaced by $s_{4,1}$ ($v_{1,1}$) and $s_{4,2}$ ($v_{1,2}$), the problem is that the static sequence part of R_1 (R_2) still includes the now invalid halfsegment $(h^r_{s_4}, f)$ ($(h^r_{v_1}, t)$), which we may not delete (see Figure 9b). However, this is not a problem due to the following observation. If we find a right halfsegment in the dynamic sequence part of a *region2D* object, we know that it stems from splitting a longer, collinear, right halfsegment that is stored in the static sequence part of this object, has the same right end point, and has to be skipped during the parallel object traversal.

R_1 *dynamic* sequence part	$(h^r_{s_{4,1}}, f)$	$(h^l_{s_{4,2}}, f)$	$(h^r_{s_{4,2}}, f)$		
R_1 *static* sequence part	$(h^l_{s_1}, t)$	$(h^l_{s_2}, f)$	$(h^l_{s_3}, t)$	$(h^l_{s_4}, f)$	$(h^r_{s_3}, t)$
	$(h^r_{s_2}, f)$	$(h^r_{s_4}, f)$	$(h^l_{s_5}, f)$	$(h^r_{s_5}, f)$	$(h^r_{s_1}, t)$
R_2 *static* sequence part	$(h^l_{v_1}, t)$	$(h^l_{v_2}, f)$	$(h^r_{v_2}, f)$	$(h^l_{v_3}, f)$	$(h^r_{v_3}, f)$
	$(h^r_{v_1}, t)$				
R_2 *dynamic* sequence part	$(h^r_{v_{1,1}}, t)$	$(h^l_{v_{1,2}}, t)$	$(h^r_{v_{1,2}}, t)$		

Table 2: Static and dynamic halfsegment sequences of the regions R_1 and R_2 in Figure 9.

For the second and third case in Figure 8, the procedure is the same but more splits can occur. In case of overlapping, collinear segments, we obtain up to six new halfsegments. In case of a touching situation, the segment whose interior is touched is split. In all cases considered, each right halfsegment of a split segment contained in the static sequence part gets a new counterpart in the dynamic sequence part pointing to it.

5 The Exploration Phase for Collecting Topological Information

For a given scene of two spatial objects, the goal of the exploration phase is to discover appropriate topological information that is characteristic and unique for this scene and that is suitable both for verification queries (query type 1) and determination queries (query type 2). Our approach is to scan such a scene from left to right by a plane sweep and to collect topological data during this traversal that later in the evaluation phase helps us confirm, deny, or derive the topological relationship between both objects. From both phases, the exploration phase is the computationally expensive one since topological information has to be explicitly derived by geometric computation.

Our research shows that it is unfavorable to aim at designing a *universal* exploration algorithm that covers all combinations of spatial data types. This has three main reasons. First, each of the data types *point2D*, *line2D*, and *region2D* has very type-specific, well known properties that are different from each other (like different dimensionality). Second, for each combination of spatial data types, the topological information we have to collect is very specific and especially different from all other type combinations. Third, the topological information we collect about each spatial data type is different in different type combinations.

Therefore, using the basic algorithmic concepts of Section 4, in this section, we present exploration algorithms for all combinations of complex spatial data types. Between two objects of types *point2D*, *line2D*, or *region2D*, we have to distinguish six different cases, if we assume that the first operand has an equal or lower dimension than the second operand[2]. All algorithms except for the *point2D*/*point2D* case require the plane sweep technique. Depending on the types of spatial objects involved, a boolean vector v_F consisting of a special set of "topological flags" is assigned to *each* object F. We call it a *topological feature vector*. Its flags are all initialized to *false*. Once certain topological information about an object has been discovered, the corresponding flag of its topological feature vector is set to *true*. For all type combinations, we aim at minimizing the number of topological flags of both spatial argument objects. In symmetric cases, only the first object gets the flag. The topological feature vectors are later used in the evaluation phase for predicate matching. Hence, the selection of topological flags is highly motivated by the requirements of the evaluation phase (Section 6, [37]).

Let $P(F)$ be the *set* of all points of a *point2D* object F, $H(F)$ be the *set* of all (attributed) halfsegments [including those resulting from our splitting strategy] of a *line2D* (*region2D*) object F, and $B(F)$ be the set of all boundary points of a *line2D* object F. For $f \in H(F)$, let $f.s$ denote its segment component, and, if F is a *region2D* object, let $f.ia$ denote its attribute component. The definitions in the next subsections make use of the operations on robust geometric primitives (Section 3.2) and halfsegments (Section 3.3).

5.1 The Exploration Algorithm for the *point2D*/*point2D* Case

The first and simplest case considers the exploration of topological information for two *point2D* objects F and G. Here, the topological facts of interest are whether (i) both objects have a point in common and (ii) F (G) contains a point that is not part of G (F). Hence, both topological feature vectors v_F and v_G get the flag *poi_disjoint*. But only v_F in addition gets the flag *poi_shared* since the sharing of a point is symmetric. We obtain (the symbol ":⇔" means "equivalent by definition"):

[2] If, in the determination query case, a predicate $p(A,B)$ has to be processed, for which the dimension of object A is higher than the dimension of object B, we process the converse predicate $p^{conv}(B,A)$ where p^{conv} has the transpose of the 9-intersection matrix (see Figure 2) of p.

```
01 algorithm ExplorePoint2DPoint2D
02 input:   point2D objects F and G, topological feature
03          vectors v_F and v_G initialized with false
04 output:  updated vectors v_F and v_G
05 begin
06   select_first(F, G, object, status);
07   while status = end_of_none and not (v_F[poi_disjoint]
08         and v_G[poi_disjoint] and v_F[poi_shared]) do
09     if object = first then v_F[poi_disjoint] := true
10     else if object = second then v_G[poi_disjoint] := true
11     else /* object = both */
12       v_F[poi_shared] := true;
13     endif
14     select_next(F, G, object, status);
15   endwhile;
16   if status = end_of_first then v_G[poi_disjoint] := true
17   else if status = end_of_second then
18     v_F[poi_disjoint] := true
19   endif
20 end ExplorePoint2DPoint2D.
```

Figure 10: Algorithm for computing the topological feature vectors for two *point2D* objects

Definition 1 Let $F, G \in$ *point2D*, and let v_F and v_G be their topological feature vectors. Then

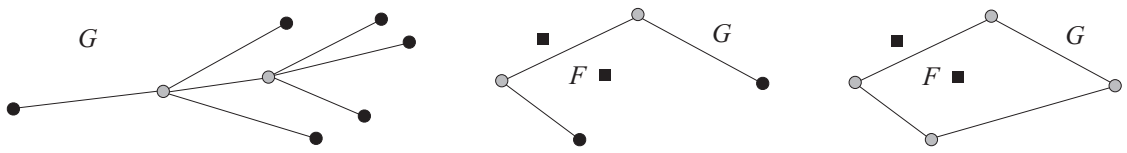
$$\begin{array}{lrcl} \text{(i)} & v_F[\mathit{poi\_shared}] & :\Leftrightarrow & \exists\, f \in P(F)\ \exists g \in P(G) : f = g \\ \text{(ii)} & v_F[\mathit{poi\_disjoint}] & :\Leftrightarrow & \exists\, f \in P(F)\ \forall g \in P(G) : f \neq g \\ \text{(iii)} & v_G[\mathit{poi\_disjoint}] & :\Leftrightarrow & \exists g \in P(G)\ \forall\, f \in P(F) : f \neq g \end{array}$$

For the computation of the topological feature vectors, a plane sweep is not needed in this case; a parallel traversal suffices, as the algorithm in Figure 10 shows. The while-loop terminates if either the end of one of the objects has been reached or all topological flags have been set to *true* (lines 7 and 8). In the worst case, the loop has to be traversed $l+m$ times where l (m) is the number of points of the first (second) *point2D* object. Since the body of the while-loop requires constant time, the overall time complexity is $O(l+m)$.

5.2 The Exploration Algorithm for the *point2D*/*line2D* Case

In case of a *point2D* object F and a *line2D* object G, at the lowest level of detail, we are interested in the possible relative positions between the individual points of F and the halfsegments of G. This requires a precise understanding of the definition of the boundary of a *line2D* object, as it has been given in [45]. It follows from this definition that each boundary point of G is an endpoint of a (half)segment of G and that this does not necessarily hold vice versa, as Figure 11(a) indicates. The black segment endpoints belong to the boundary of G, since exactly one segment emanates from each of them. Intuitively, they "bound" G. In contrast, the grey segment endpoints belong to the interior of G, since several segments emanate from each of them. Intuitively, they are "connector points" between different segments of G.

The following argumentation leads to the needed topological flags for F and G. Seen from the perspective of F, we can distinguish three cases, since the boundary of F is empty [45] and the interior of F can interact with the exterior, interior, or boundary of G. First, (the interior of) a point f of F can be disjoint from G (flag *poi_disjoint*). Second, a point f can lie in the interior of a segment of G (flag *poi_on_interior*).



(a) (b) (c)

Figure 11: Boundary points (in black) and connector points (in grey) of a *line2D* object (a), a scenario where a boundary point of a *line2D* object exists that is unequal to all points of a *point2D* object (b), a scenario where this is not the case (c).

```
01 algorithm ExplorePoint2DLine2D
02 input:   point2D object F and line2D object G,
03          topological feature vectors v_F and v_G
04          initialized with false
05 output:  updated vectors v_F and v_G
06 begin
07   S := new_sweep(); last_dp := ε;
08   select_first(F, G, object, status);
09   while status ≠ end_of_second and status ≠ end_of_both and
10         not (v_F[poi_disjoint] and v_F[poi_on_interior] and
11         v_F[poi_on_bound] and v_G[bound_poi_disjoint]) do
12     if object = first then p := get_event(F);
13       if poi_in_seg(S, p) then v_F[poi_on_interior] := true
14       else v_F[poi_disjoint] := true endif
15     else if object = second then
16       h := get_event(G); /* h = (s,d) */
17       if d then add_left(S, s) else del_right(S, s) endif;
18       if dp(h) ≠ last_dp then last_dp := dp(h);
19         if not look_ahead(h,G) then
20           v_G[bound_poi_disjoint] := true
21         endif
22       endif
23     else /* object = both */
24       h := get_event(G); /* h = (s,d) */
25       if d then add_left(S, s) else del_right(S, s) endif;
26       last_dp := dp(h);
27       if look_ahead(h,G) then
28         v_F[poi_on_interior] := true
29       else v_F[poi_on_bound] := true endif
30     endif
31     select_next(F, G, object, status);
32   endwhile;
33   if status = end_of_second then
34     v_F[poi_disjoint] := true
35   endif
36 end ExplorePoint2DLine2D.
```

Figure 12: Algorithm for computing the topological feature vectors for a *point2D* object and a *line2D* object

This includes an endpoint of such a segment, if the endpoint is a connector point of G. Third, a point f can be equal to a boundary point of G (flag *poi_on_bound*). Seen from the perspective of G, we can distinguish four cases, since the boundary and the interior of G can interact with the interior and exterior of F. First, G can contain a boundary point that is unequal to all points in F (flag *bound_poi_disjoint*). Second, G can have a boundary point that is equal to a point in F. But the flag *poi_on_bound* already takes care of this situation. Third, the interior of a segment of G (including connector points) can comprehend a point of F. This situation is already covered by the flag *poi_on_interior*. Fourth, the interior of a segment of G can be part of the exterior of F. This is always true since a segment of G represents an infinite point set that cannot be covered by the finite number of points in F. Hence, we need not handle this as a special situation. More formally, we define the semantics of the topological flags as follows:

Definition 2 Let $F \in$ *point2D*, $G \in$ *line2D*, and v_F and v_G be their topological feature vectors. Then

$$
\begin{array}{rlll}
\text{(i)} & v_F[\mathit{poi\_disjoint}] & :\Leftrightarrow & \exists f \in P(F)\ \forall g \in H(G) : \neg \mathit{on}(f, g.s) \\
\text{(ii)} & v_F[\mathit{poi\_on\_interior}] & :\Leftrightarrow & \exists f \in P(F)\ \exists g \in H(G)\ \forall b \in B(G) : \mathit{on}(f, g.s) \ \wedge\ f \neq b \\
\text{(iii)} & v_F[\mathit{poi\_on\_bound}] & :\Leftrightarrow & \exists f \in P(F)\ \exists g \in B(G) : f = g \\
\text{(iv)} & v_G[\mathit{bound\_poi\_disjoint}] & :\Leftrightarrow & \exists g \in B(G)\ \forall f \in P(F) : f \neq g
\end{array}
$$

Our algorithm for computing the topological information for this case is shown in Figure 12. The while-loop is executed until the end of the *line2D* object (line 9) and as long as not all topological flags have been set to *true* (lines 10 to 11). The operations *select_first* and *select_next* compare a point and a halfsegment according to the order relation defined in Section 3.3 in order to determine the next element(s) to be processed. If only a point has to be processed (line 12), we know that it does not coincide with an endpoint of a segment of G and hence not with a boundary point of G. But we have to check whether the point lies in the interior of a segment in the sweep line status structure S. This is done by the search operation *poi_in_seg* on S (line 13). If this is not the case, the point must be located outside the segment (line 14). If only a halfsegment h has to be processed (line 15), its segment component is inserted into (deleted from) S if h is a left (right) halfsegment (line 17). We also have to test if the dominating point of h, say v, is a boundary point of G. This is the case if v is unequal to the previous dominating point stored in the variable *last_dp* (line 18) and if the operation *look_ahead* finds out that v does also not coincide with the dominating point of the next halfsegment (lines 19 to 20). In case that a point v of F is equal to a dominating point of

```
01 algorithm ExplorePoint2DRegion2D
02 input:  point2D object F and region2D object G,
03         topological feature vectors v_F and v_G
04         initialized with false
05 output: updated vectors v_F and v_G
06 begin
07   S := new_sweep();
08   select_first(F, G, object, status);
09   while status = end_of_none and not (v_F[poi_inside]
10         and v_F[poi_on_bound] and v_F[poi_outside]) do
11     if object = first then p := get_event(F);
12       if poi_on_seg(S, p) then v_F[poi_on_bound] := true
13       else pred_of_p(S, p);
14         if current_exists(S) then ia := get_attr(S);
15           if ia then v_F[poi_inside] := true
16             else v_F[poi_outside] := true endif
17           else v_F[poi_outside] := true
18           endif
19       endif
20     else h := get_event(G); ia := get_attr(G); /* h = (s,d) */
21       if d then add_left(S, s); set_attr(S, ia)
22       else del_right(S, s) endif;
23       if object = both then v_F[poi_on_bound] := true endif
24     endif
25     select_next(F, G, object, status);
26   endwhile;
27   if status = end_of_second then
28     v_F[poi_outside] := true
29   endif
30 end ExplorePoint2DRegion2D.
```

Figure 13: Algorithm for computing the topological feature vectors for a *point2D* object and a *region2D* object

a halfsegment h in G (line 23), we know that v has never been visited before and that it is an end point of the segment component of h. Besides the update of S (line 25), it remains to decide whether v is an interior point (line 28) or a boundary point (line 29) of h. For this, we look ahead (line 27) to see whether the next halfsegment's dominating point is equal to v or not.

If l is the number of points of F and m is the number of halfsegments of G, the while-loop is executed at most $l+m$ times. The insertion of a left halfsegment into and the removal of a right halfsegment from the sweep line status needs $O(\log m)$ time. The check whether a point lies within or outside a segment (predicate *poi_in_seg*) also requires $O(\log m)$ time. Altogether, the worst time complexity is $O((l+m)\log m)$. The while-loop has to be executed at least m times for processing the entire *line2D* object in order to find out whether a boundary point exists that is unequal to all points of the *point2D* object (Figures 11(b) and (c)).

5.3 The Exploration Algorithm for the *point2D*/*region2D* Case

In case of a *point2D* object F and a *region2D* object G, the situation is simpler than in the previous case. Seen from the perspective of F, we can again distinguish three cases between (the interior of) a point of F and the exterior, interior, or boundary of G. First, a point of F lies either inside G (flag *poi_inside*), on the boundary of G (flag *poi_on_bound*), or outside of region G (flag *poi_outside*). Seen from the perspective of G, we can distinguish four cases between the boundary and the interior of G with the interior and exterior of F. The intersection of the boundary (interior) of G with the interior of F implies that a point of F is located on the boundary (inside) of G. This situation is already covered by the flag *poi_on_bound* (*poi_inside*). The intersection of the boundary (interior) of G with the exterior of F is always true, since F as a finite point set cannot cover G's boundary segments (interior) representing an infinite point set. More formally, we define the semantics of the topological flags as follows (we assume *poiInRegion* to be a predicate which checks whether a point lies inside a *region2D* object):

Definition 3 Let $F \in$ *point2D*, $G \in$ *region2D*, and v_F and v_G be their topological feature vectors. Then

$$\begin{array}{lll} \text{(i)} & v_F[\mathit{poi\_inside}] & :\Leftrightarrow \quad \exists\, f \in P(F) : \mathit{poiInRegion}(f, G) \\ \text{(ii)} & v_F[\mathit{poi\_on\_bound}] & :\Leftrightarrow \quad \exists\, f \in P(F)\ \exists g \in H(G) : \mathit{on}(f, g.s) \\ \text{(iii)} & v_F[\mathit{poi\_outside}] & :\Leftrightarrow \quad \exists\, f \in P(F)\ \forall g \in H(G) : \neg \mathit{poiInRegion}(f, G) \wedge \neg \mathit{on}(f, g.s) \end{array}$$

We see that v_G is not needed. The algorithm for this case is shown in Figure 13. The while-loop is executed as long as none of the two objects has been processed (line 9) and as long as not all topological

flags have been set to *true* (lines 9 to 10). If only a point has to be processed (line 11), we must check its location. The first case is that it lies on a boundary segment; this is checked by the sweep line status predicate *poi_on_seg* (line 12). Otherwise, it must be located inside or outside of G. We use the operation *pred_of_p* (line 13) to determine the nearest segment in the sweep line status whose intersection point with the sweep line has a lower y-coordinate than the y-coordinate of the point. The predicate *current_exists* checks whether such a segment exists (line 14). If this is not the case, the point must be outside of G (line 17). Otherwise, we ask for the information whether the interior of G is above the segment (line 14). We can then derive whether the point is inside or outside the region (lines 15 to 16). If only a halfsegment h has to be processed or a point v of F is equal to a dominating point of a halfsegment h in G (line 20), h's segment component is inserted into (deleted from) S if h is a left (right) halfsegment (line 20 to 21). In case of a left halfsegment, in addition, the information whether the interior of G is above the segment is stored in the sweep line status (line 21). If v and the dominating point of h coincide, we know that the point is on the boundary of G (line 23).

If l is the number of points of F and m is the number of halfsegments of G, the while-loop is executed at most $l+m$ times. Each of the sweep line status operations *add_left*, *del_right*, *poi_on_seg*, *pred_of_p*, *current_exists*, *get_attr*, and *set_attr* needs $O(\log m)$ time. The total worst time complexity is $O((l+m)\log m)$.

5.4 The Exploration Algorithm for the *line2D*/*line2D* Case

We now consider the exploration algorithm for two *line2D* objects F and G. Seen from the perspective of F, we can differentiate six cases between the interior and boundary of F and the interior, boundary, and exterior of G. First, the interiors of two segments of F and G can partially or completely coincide (flag *seg_shared*). Second, if a segment of F does not partially or completely coincide with any segment of G, we register this in the flag *seg_unshared*. Third, we set the flag *interior_poi_shared* if two segments intersect in a single point that does not belong to the boundaries of F or G. Fourth, a boundary endpoint of a segment of F can be located in the interior of a segment (including connector points) of G (flag *bound_on_interior*). Fifth, both objects F and G can share a boundary point (flag *bound_shared*). Sixth, if a boundary endpoint of a segment of F lies outside of all segments of G, we set the flag *bound_disjoint*. Seen from the perspective of G, we can identify the same cases. But due to the symmetry of three of the six topological cases, we do not need all flags for G. For example, if a segment of F partially coincides with a segment of G, this also holds vice versa. Hence, it is sufficient to introduce the flags *seg_unshared*, *bound_on_interior*, and *bound_disjoint* for G. More formally, we define the semantics of the topological flags as follows:

Definition 4 Let $F, G \in$ *line2D*, and let v_F and v_G be their topological feature vectors. Then

$$
\begin{array}{lrll}
\text{(i)} & v_F[\textit{seg\_shared}] & :\Leftrightarrow & \exists\, f \in H(F)\ \exists g \in H(G) : \textit{segIntersect}(f.s, g.s) \\
\text{(ii)} & v_F[\textit{interior\_poi\_shared}] & :\Leftrightarrow & \exists\, f \in H(F)\ \exists g \in H(G)\ \forall\, p \in B(F) \cup B(G) : \\
 & & & \textit{poiIntersect}(f.s, g.s) \ \wedge\ \textit{poiIntersection}(f.s, g.s) \neq p \\
\text{(iii)} & v_F[\textit{seg\_unshared}] & :\Leftrightarrow & \exists\, f \in H(F)\ \forall g \in H(G) : \neg\textit{segIntersect}(f.s, g.s) \\
\text{(iv)} & v_F[\textit{bound\_on\_interior}] & :\Leftrightarrow & \exists\, f \in H(F)\ \exists g \in H(G)\ \exists\, p \in B(F) \setminus B(G) : \\
 & & & \textit{poiIntersection}(f.s, g.s) = p \\
\text{(v)} & v_F[\textit{bound\_shared}] & :\Leftrightarrow & \exists\, p \in B(F)\ \exists q \in B(G) : p = q \\
\text{(vi)} & v_F[\textit{bound\_disjoint}] & :\Leftrightarrow & \exists\, p \in B(F)\ \forall g \in H(G) : \neg\textit{on}(p, g.s) \\
\text{(vii)} & v_G[\textit{seg\_unshared}] & :\Leftrightarrow & \exists g \in H(G)\ \forall\, f \in H(F) : \neg\textit{segIntersect}(f.s, g.s) \\
\text{(viii)} & v_G[\textit{bound\_on\_interior}] & :\Leftrightarrow & \exists\, f \in H(F)\ \exists g \in H(G)\ \exists\, p \in B(G) \setminus B(F) : \\
 & & & \textit{poiIntersection}(f.s, g.s) = p \\
\text{(ix)} & v_G[\textit{bound\_disjoint}] & :\Leftrightarrow & \exists q \in B(G)\ \forall\, f \in H(F) : \neg\textit{on}(q, f.s)
\end{array}
$$

The exploration algorithm for this case is given in Figure 14. The while-loop is executed until both objects have been processed (line 9) and as long as not all topological flags have been set to *true* (lines 9 to

```
01 algorithm ExploreLine2DLine2D
02 input:  line2D objects F and G, topological feature
03         vectors v_F and v_G initialized with false
04 output: updated vectors v_F and v_G
05 begin
06   S := new_sweep(); last_dp_in_F := ε; last_dp_in_G := ε;
07   last_bound_in_F := ε; last_bound_in_G := ε;
08   select_first(F, G, object, status);
09   while status ≠ end_of_both and not (v_F[seg_shared]
10         and v_F[interior_poi_shared] and v_F[seg_unshared]
11         and v_F[bound_on_interior] and v_F[bound_shared]
12         and v_F[bound_disjoint] and v_G[bound_disjoint]
13         and v_G[bound_on_interior] and v_G[seg_unshared]) do
14     if object = first then h := get_event(F); /* h = (s,d) */
15       if d then add_left(S, s)
16       else del_right(S, s); v_F[seg_unshared] := true endif;
17       if dp(h) ≠ last_dp_in_F then last_dp_in_F := dp(h);
18         if not look_ahead(h,F) then
19           last_bound_in_F := dp(h);
20           if last_bound_in_F = last_bound_in_G then
21             v_F[bound_shared] := true
22           else if last_bound_in_F = last_dp_in_G then
23             v_F[bound_on_interior] := true
24           else if not look_ahead(h,G) then
25             v_F[bound_disjoint] := true
26           endif
27         endif
28       endif;
29       if dp(h) ≠ last_bound_in_F then
30         if dp(h) = last_bound_in_G then
31           v_G[bound_on_interior] := true
32         else if dp(h) = last_dp_in_G then
33           v_F[interior_poi_shared] := true
34         endif
35       endif
36     else if object = second then h := get_event(G);
37       ... /* like lines 15 to 35 with F and G swapped */
38     else /* object = both */
39       h := get_event(F); v_F[seg_shared] := true;
40       if d then add_left(S, s) else del_right(S, s) endif;
41       if dp(h) ≠ last_dp_in_F then last_dp_in_F := dp(h);
42         if not look_ahead(h,F) then
43           last_bound_in_F := dp(h)
44         endif
45       endif;
46       if dp(h) ≠ last_dp_in_G then last_dp_in_G := dp(h);
47         if not look_ahead(h,G) then
48           last_bound_in_G := dp(h)
49         endif
50       endif;
51       if last_bound_in_F = last_bound_in_G then
52         v_F[bound_shared] := true
53       else
54         if last_bound_in_F = last_dp_in_G then
55           v_F[bound_on_interior] := true
56         endif;
57         if last_bound_in_G = last_dp_in_F then
58           v_G[bound_on_interior] := true
59         endif
60       endif
61     endif;
62     if status = end_of_first then
63       v_G[seg_unshared] := true;
64     else if status = end_of_second then
65       v_F[seg_unshared] := true;
66     endif
67     select_next(F, G, object, status);
68   endwhile;
69 end ExploreLine2DLine2D.
```

Figure 14: Algorithm for computing the topological feature vectors for two *line2D* objects

13). If a single left (right) halfsegment of F (line 14) has to be processed (the same for G (line 36)), we insert it into (delete it from) the sweep line status (lines 15 and 16). The deletion of a single right halfsegment further indicates that it is not shared by G (line 16). If the current dominating point, say v, is unequal to the previous dominating point of F (line 17) and if the operation look_ahead finds out that v is also unequal to the dominating point of the next halfsegment of F (line 18), v must be a boundary point of F (line 19). In this case, we perform three checks. First, if v coincides with the current boundary point in G, both objects share a part of their boundary (lines 20 to 21). Second, otherwise, if v is equal to the current dominating point, say w, in G, w must be an interior point of G, and the boundary of F and the interior of G share a point (lines 22 to 23). Third, otherwise, if v is different from the dominating point of the next halfsegment in G, F contains a boundary point that is disjoint from G (lines 24 to 25). If v has not been identified as a boundary point in the previous step (line 29), it must be an interior point of F. In this case, we check whether it coincides with the current boundary point in G (lines 30 to 31) or whether it is also an interior point in G (lines 32 to 33). If a halfsegment belongs to both objects (line 38), we can conclude that it is shared by them (line 39). Depending on whether it is a left or right halfsegment, it is inserted into or deleted from the sweepline status (line 40). Lines 41 to 45 (46 to 50) test whether the dominating point v of the halfsegment

```
01 algorithm ExploreLine2DRegion2D
02 input:  line2D object F and region2D object G,
03         topological feature vectors v_F and v_G
04         initialized with false
05 output: updated vectors v_F and v_G
06 begin
07   S := new_sweep(); last_dp_in_F := ε; last_dp_in_G := ε;
08   last_bound_in_F := ε;
09   select_first(F, G, object, status);
10   while status ≠ end_of_first and status ≠ end_of_both
11         and not (v_F[seg_inside] and v_F[seg_shared]
12         and v_F[seg_outside] and v_F[poi_shared]
13         and v_F[bound_inside] and v_F[bound_shared]
14         and v_G[bound_disjoint] and v_G[seg_unshared]) do
15       if object = first then h := get_event(F); /* h = (s,d) */
16         if d then add_left(S, s)
17         else
18           if pred_exists(S,s) then
19             (m_p/n_p) := get_pred_attr(S, s);
20             if n_p = 1 then v_F[seg_inside] := true
21             else v_F[seg_outside] := true endif
22           else v_F[seg_outside] := true endif;
23           del_right(S, s);
24         endif;
25         if dp(h) ≠ last_dp_in_F then last_dp_in_F := dp(h);
26           if not look_ahead(h,F) then
27             last_bound_in_F := dp(h);
28             if last_bound_in_F = last_dp_in_G
29               or look_ahead(h,G) then
30               v_F[bound_shared] := true
31             else
32               if pred_exists(S,s) then
33                 (m_p/n_p) := get_pred_attr(S, s);
34                 if n_p = 1 then v_F[bound_inside] := true
35                 else v_F[bound_outside] := true endif
36               else v_F[bound_outside] := true endif
37             endif
38           endif
39         endif;
40         if dp(h) ≠ last_bound_in_F and
41           (dp(h) = last_dp_in_G or look_ahead(h,G)) then
42           v_F[poi_shared] := true
43         endif
44       else if object = second then
45         h := get_event(G); ia := get_attr(G);
46         if d then add_left(S, s); set_attr(S, ia)
47         else del_right(S, s); v_G[seg_unshared] := true endif;
48         if dp(h) ≠ last_dp_in_G then
49           last_dp_in_G := dp(h) endif;
50       else /* object = both */ v_F[seg_shared] := true;
51         h := get_event(G); ia := get_attr(G);
52         if d then add_left(S, s); set_attr(S, ia)
53         else del_right(S, s) endif;
54         if dp(h) ≠ last_dp_in_F then last_dp_in_F := dp(h);
55           if not look_ahead(h,F) then
56             v_F[bound_shared] := true
57           else v_F[poi_shared] := true endif
58         endif;
59         if dp(h) ≠ last_dp_in_G then
60           last_dp_in_G := dp(h) endif;
61       endif;
62       if status = end_of_second then
63         v_F[seg_outside] := true endif;
64       select_next(F, G, object, status);
65   endwhile;
66   if status = end_of_first then
67     v_G[seg_unshared] := true endif
68 end ExploreLine2DRegion2D.
```

Figure 15: Algorithm for computing the topological feature vectors for a *line2D* object and a *region2D* object

is a boundary point of F (G). Afterwards, we check whether v is a boundary point of both objects (lines 51 to 52). If this is not the case, we examine whether one of them is a boundary point and the other one is an interior point (lines 54 to 59). Lines 62 to 66 handle the case that exactly one of the two halfsegment sequences is exhausted.

Let l (m) be the number of halfsegments of F (G). Segments of both objects can intersect or partially coincide (Figure 8), and we handle these topological situations with the splitting strategy described in Section 4.3. If, due to splitting, k is the total number of additional halfsegments stored in the dynamic halfsegment sequences of both objects, the while-loop is executed at most $l+m+k$ times. The only operations needed on the sweep line status are *add_left* and *del_right* for inserting and deleting halfsegments; they require $O(\log(l+m+k))$ time each. No special predicates have to be deployed for discovering topological information. Due to the splitting strategy, all dominating end points either are already endpoints of existing segments or become endpoints of newly created segments. The operation *look_ahead* needs constant time. In total, the algorithm requires $O((l+m+k)\log(l+m+k))$ time and $O(l+m+k)$ space.

5.5 The Exploration Algorithm for the *line2D*/*region2D* Case

Next, we describe the exploration algorithm for a *line2D* object F and a *region2D* object G. Seen from the perspective of F, we can distinguish six cases between the interior and boundary of F and the interior, boundary, and exterior of G. First, the intersection of the interiors of F and G means that a segment of F lies in G (flag *seg_inside*). Second, the interior of a segment of F intersects with a boundary segment of G if either both segments partially or fully coincide (flag *seg_shared*), or if they properly intersect in a single point (flag *poi_shared*). Third, the interior of a segment of F intersects with the exterior of G if the segment is disjoint from G (flag *seg_outside*). Fourth, a boundary point of F intersects the interior of G if the boundary point lies inside of G (flag *bound_inside*). Fifth, if it lies on the boundary of G, we set the flag *bound_shared*. Sixth, if it lies outside of G, we set the flag *bound_disjoint*.

Seen from the perspective of G, we can differentiate the same six cases as before and obtain most of the topological flags as before. First, if the interiors of G and F intersect, a segment of F must partially or totally lie in G (already covered by flag *seg_inside*). Second, if the interior of G and the boundary of F intersect, the boundary point of a segment of F must be located in G (already covered by flag *bound_inside*). Third, the case that the interior of G intersects the exterior of F is always true due to the different dimensionality of both objects; hence, we do not need a flag. Fourth, if the boundary of G intersects the interior of F, a segment of F must partially or fully coincide with a boundary segment of G (already covered by flag *seg_shared*). Fifth, if the boundary of G intersects the boundary of F, a boundary point of a segment of F must lie on a boundary segment of G (already covered by flag *bound_shared*). Sixth, if the boundary of G intersects the exterior of F, a boundary segment of G must be disjoint from F (new flag *seg_unshared*). More formally, we define the semantics of the topological flags as follows:

Definition 5 Let $F \in$ *line2D*, $G \in$ *region2D*, and v_F and v_G be their topological feature vectors. Then

$$
\begin{array}{lrll}
\text{(i)} & v_F[\textit{seg\_inside}] & :\Leftrightarrow & \exists f \in H(F)\ \forall g \in H(G): \\
 & & & \neg \textit{segIntersect}(f.s, g.s)\ \wedge\ \textit{segInRegion}(f.s, G) \\
\text{(ii)} & v_F[\textit{seg\_shared}] & :\Leftrightarrow & \exists f \in H(F)\ \exists g \in H(G): \textit{segIntersect}(f.s, g.s) \\
\text{(iii)} & v_F[\textit{seg\_outside}] & :\Leftrightarrow & \exists f \in H(F)\ \forall g \in H(G): \\
 & & & \neg \textit{segIntersect}(f.s, g.s)\ \wedge\ \neg \textit{segInRegion}(f.s, G) \\
\text{(iv)} & v_F[\textit{poi\_shared}] & :\Leftrightarrow & \exists f \in H(F)\ \exists g \in H(G): \textit{poiIntersect}(f.s, g.s)\ \wedge \\
 & & & \textit{poiIntersection}(f.s, g.s) \notin B(F) \\
\text{(v)} & v_F[\textit{bound\_inside}] & :\Leftrightarrow & \exists f \in H(F): \textit{poiInRegion}(dp(f), G)\ \wedge\ dp(f) \in B(F) \\
\text{(vi)} & v_F[\textit{bound\_shared}] & :\Leftrightarrow & \exists f \in H(F)\ \exists g \in H(G): \textit{poiIntersect}(f.s, g.s)\ \wedge \\
 & & & \textit{poiIntersection}(f.s, g.s) \in B(F) \\
\text{(vii)} & v_F[\textit{bound\_disjoint}] & :\Leftrightarrow & \exists f \in H(F)\ \forall g \in H(G): \neg \textit{poiInRegion}(dp(f), G)\ \wedge \\
 & & & dp(f) \in B(F)\ \wedge\ \neg \textit{on}(dp(f), g.s) \\
\text{(viii)} & v_G[\textit{seg\_unshared}] & :\Leftrightarrow & \exists g \in H(G)\ \forall f \in H(F): \neg \textit{segIntersect}(f.s, g.s)
\end{array}
$$

The operation *segInRegion* is assumed to check whether a segment is located inside a region; it is an imaginary predicate and not implemented as a robust geometric primitive.

The exploration algorithm for this case is given in Figure 15. The while-loop is executed until at least the first object has been processed (line 10) and as long as not all topological flags have been set to *true* (lines 11 to 14). In case that we only encounter a halfsegment h of F (line 15), we insert its segment component s into the sweep line status if it is a left halfsegment (line 16). If it is a right halfsegment, we find out whether h is located inside or outside of G (lines 18 to 23). We know that it cannot coincide with a boundary segment of G, since this is another case. The predicate *pred_exists* checks whether s has a predecessor in the sweep line status (line 18); it ignores segments in the sweep line status that stem from F. If this is not the case (line 22), s must lie outside of G. Otherwise, we check the upper overlap number of s's predecessor (line 19).

The overlap number 1 indicates that s lies inside G (line 20); otherwise, it is outside of G (line 21). After this check, we remove s from the sweep line status (line 23). Next we test whether the dominating point of h is a boundary point of F (line 26) by using the predicate *look_ahead*. If this is the case, we determine whether this point is shared by G (lines 28 to 30) or whether this point is located inside or outside of G (lines 31 to 38). Last, if the dominating point turns out not to be a boundary point of F, we check whether it is an interior point that shares a boundary point with G (lines 40 to 43). In case that we only obtain a halfsegment h of G (line 44), we insert its segment component s into the sweep line status and attach the Boolean flag *ia* indicating whether the interior of G is above s or not (line 46). Otherwise, we delete a right halfsegment h from the sweep line status and know that it is not shared by F (line 47). In case that both F and G share a halfsegment, we know that they also share their segment components (line 50). The sweep line status is then modified depending on the status of h (lines 52 to 53). If we encounter a new dominating point of F, we have to check whether F shares a boundary point (lines 55 to 56) or an interior point (line 57) with the boundary of G. If the halfsegment sequence of G should be exhaused (line 62), we know that F must have a segment whose interior is outside of G (line 63). If after the while-loop only F is exhausted but not G (line 66), G must have a boundary segment that is disjoint from F (line 67).

Let l be the number of halfsegments of F, m be the number of attributed halfsegments of G, and k be the total number of new halfsegments created due to our splitting strategy. The while-loop is then executed at most $l+m+k$ times. All operations needed on the sweep line status require $O(\log(l+m+k))$ time each. Due to the splitting strategy, all dominating end points are already endpoints of existing segments or become endpoints of newly created segments. The operation *look_ahead* needs constant time. In total, the algorithm requires $O((l+m+k)\log(l+m+k))$ time and $O(l+m+k)$ space.

5.6 The Exploration Algorithm for the *region2D*/*region2D* Case

The exploration algorithm for the *region2D*/*region2D* case is quite different from the preceding five cases, since it has to take into account the areal extent of both objects. The indices of the vector fields, with one exception described below, are not flags as before but segment classes. The fields of the vectors again contain Boolean values that are initialized with *false*. The main goal of the exploration algorithm is to determine the existing segment classes in each *region2D* object. Hence, the topological feature vector for each object is a *segment classification vector*. Each vector contains a field for the segment classes $(0/1)$, $(1/0)$, $(0/2)$, $(2/0)$, $(1/2)$, $(2/1)$, and $(1/1)$. The following definition makes a connection between representational concepts and point set topological concepts as it is later needed in the evaluation phase. For a segment $s=(p,q) \in$ *seg2D*, the function *pts* yields the infinite point set of s as $pts(s)=\{r \in \mathbb{R}^2 \,|\, r=p+\lambda(q-p), \lambda \in \mathbb{R}, 0 \leq \lambda \leq 1\}$. Further, for $F \in$ *region2D*, we define $\partial F = \bigcup_{f \in H(F)} pts(f.s)$, $F^\circ = \{p \in \mathbb{R}^2 \,|\, poiInRegion(p,F)\}$, and $F^- = \mathbb{R}^2 - \partial F - F^\circ$. We can now define the semantics of this vector as follows:

Definition 6 Let $F, G \in$ *region2D* and v_F be the segment classification vector of F. Then

$$
\begin{array}{rlll}
\text{(i)} & v_F[(0/1)] & :\Leftrightarrow & \exists\, f \in H(F) : f.ia \wedge pts(f.s) \subset G^- \\
\text{(ii)} & v_F[(1/0)] & :\Leftrightarrow & \exists\, f \in H(F) : \neg f.ia \wedge pts(f.s) \subset G^- \\
\text{(iii)} & v_F[(1/2)] & :\Leftrightarrow & \exists\, f \in H(F) : f.ia \wedge pts(f.s) \subset G^\circ \\
\text{(iv)} & v_F[(2/1)] & :\Leftrightarrow & \exists\, f \in H(F) : \neg f.ia \wedge pts(f.s) \subset G^\circ \\
\text{(v)} & v_F[(0/2)] & :\Leftrightarrow & \exists\, f \in H(F)\ \exists g \in H(G) : f.s = g.s \wedge f.ia \wedge g.ia \\
\text{(vi)} & v_F[(2/0)] & :\Leftrightarrow & \exists\, f \in H(F)\ \exists g \in H(G) : f.s = g.s \wedge \neg f.ia \wedge \neg g.ia \\
\text{(vii)} & v_F[(1/1)] & :\Leftrightarrow & \exists\, f \in H(F)\ \exists g \in H(G) : f.s = g.s \wedge \\
 & & & ((f.ia \wedge \neg g.ia) \vee (\neg f.ia \wedge g.ia)) \\
\text{(viii)} & v_F[bound\_poi\_shared] & :\Leftrightarrow & \exists\, f \in H(F)\ \exists g \in H(G) : f.s \neq g.s \wedge dp(f) = dp(g)
\end{array}
$$

The segment classification vector v_G of G includes the cases (i) to (iv) with F and G swapped; we omit the flags for the cases (v) to (viii) due to their symmetry (or equivalence) to flags of F. The flag

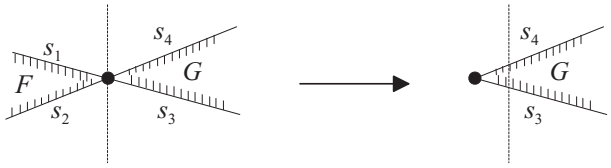


Figure 16: Special case of the plane sweep.

bound_poi_shared indicates whether any two unequal boundary segments of both objects share a common point. Before the splitting, such a point may have been a segment endpoint or a proper segment intersection point for each object. The determination of the boolean value of this flag also includes the treatment of a special case illustrated in Figure 16. If two regions F and G meet in a point like in the example, such a topological meeting situation cannot be detected by a usual plane sweep. The reason is that the plane sweep forgets completely about the already visited segments (right halfsegments) left of the sweep line. In our example, after s_1 and s_2 have been removed from the sweep line status, any information about them is lost. When s_3 is inserted into the status sweep line, its meeting with s_2 cannot be detected. Our solution is to look ahead in object G for a next halfsegment with the same dominating point before s_2 is removed from the sweep line status.

The segment classification is computed by the algorithm in Figure 17. The while-loop is executed as long as none of the two objects has been processed (line 8) and as long as not all topological flags have been set to *true* (lines 8 to 12). Then, according to the halfsegment order, the next halfsegment h is obtained, which belongs to one or both objects, and the variables for the last considered dominating points in F and/or G are updated (lines 13 to 20). Next, we check for a possible common boundary point in F and G (lines 21 to 25). This is the case if the last dominating points of F and G are equal, or the last dominating point in F (G) coincides with the next dominating point in G (F). The latter algorithmic step, in particular, helps us solve the special situation in Figure 16. If h is a right halfsegment (line 26), we update the topological feature vectors of F and/or G correspondingly (lines 27 to 32) and remove its segment component s from the sweep line status (line 33). In case that h is a left halfsegment, we insert its segment component s into the sweep line status (line 34) according to the y-order of its dominating point and the y-coordinates of the intersection points of the current sweep line with the segments momentarily in the sweep line status. If h's segment component s either belongs to F or to G, but not to both objects, and partially coincides with a segment from the other object in the sweep lines status, our splitting strategy is applied. Its effect is that the segment we currently consider suddenly belongs to *both* objects. Therefore, we modify the *object* variable in line 35 correspondingly. Next, we compute the segment class of s. For this purpose, we determine the lower and upper overlap numbers m_p and n_p of the predecessor p of s (lines 36 to 37). If there is no predecessor, m_p gets the 'undefined' value '$*$'. The segment classification of the predecessor p is important since the lower overlap number m_s of s is assigned the upper overlap number n_p of p, and the upper overlap number n_s of s is assigned its incremented or decremented lower overlap number, depending on whether the Boolean flag

n_s	1	2	1	2	1	2	0	1	2	0	1	2	0	1	2	0	1	0	1
m_s	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	2	2	2	2
n_p	$*$	$*$	0	0	0	0	1	1	1	1	1	1	1	1	1	2	2	2	2
m_p	$*$	$*$	1	1	2	2	0	0	0	1	1	1	2	2	2	0	0	1	1

Table 3: Possible segment class constellations between two consecutive segments in the sweep line status.

```
01 algorithm ExploreRegion2DRegion2D
02 input:   region2D objects F and G, topological feature
03          vectors v_F and v_G initialized with false
04 output:  updated vectors v_F and v_G
05 begin
06   S := new_sweep(); last_dp_in_F := ε; last_dp_in_G := ε;
07   select_first(F, G, object, status);
08   while status = end_of_none and not (v_F[(0/1)] and v_F[(1/0)]
09         and v_F[(1/2)] and v_F[(2/1)] and v_F[(0/2)] and
10         v_F[(2/0)] and v_F[(1/1)] and v_F[bound_poi_shared]
11         and v_G[(0/1)] and v_G[(1/0)] and v_G[(1/2)] and
12         v_G[(2/1)]) do
13     if object = first then /* h = (s,d) */
14       h := get_event(F); last_dp_in_F := dp(h)
15     else if object = second then
16       h := get_event(G); last_dp_in_G := dp(h)
17     else /* object = both */
18       h := get_event(F);
19       last_dp_in_F := dp(h); last_dp_in_G := dp(h)
20     endif;
21     if last_dp_in_F = last_dp_in_G
22       or last_dp_in_F = look_ahead(h, G)
23       or last_dp_in_G = look_ahead(h, F) then
24       v_F[bound_poi_shared] := true
25     endif;
26     if d = right then
27       {(m_s/n_s)} := get_attr(S);
28       if object = first then v_F[(m_s/n_s)] := true
29       else if object = second then v_G[(m_s/n_s)] := true
30       else if object = both then
31         v_F[(m_s/n_s)] := true; v_G[(m_s/n_s)] := true
32       endif;
33       del_right(S, s)
34     else add_left(S, s);
35       if coincident(S, s) then object := both endif;
36       if not pred_exists(S, s) then (m_p/n_p) := (*/0)
37       else {(m_p/n_p)} := get_pred_attr(S) endif;
38       m_s := n_p; n_s := n_p;
39       if object = first or object = both then
40         if get_attr(F) then n_s := n_s + 1
41         else n_s := n_s - 1 endif
42       endif;
43       if object = second or object = both then
44         if get_attr(G) then n_s := n_s + 1
45         else n_s := n_s - 1 endif
46       endif;
47       set_attr(S, {(m_s/n_s)});
48     endif;
49     select_next(F, G, object, status);
50   endwhile;
51   if status = end_of_first then
52     v_G[(0/1)] := true; v_G[(1/0)] := true
53   else if status = end_of_second then
54     v_F[(0/1)] := true; v_F[(1/0)] := true
55   endif
56 end ExploreRegion2DRegion2D.
```

Figure 17: Algorithm for computing the topological feature vectors for two *region2D* objects

'Interior Above' obtained by the predicate *get_attr* is *true* or *false* respectively (lines 39 to 46). The newly computed segment classification is then attached to s (line 47). The possible 19 segment class constellations between two consecutive segments in the sweep line status are shown in Table 3. The table shows which segment classes (m_s/n_s) a new segment s just inserted into the sweep line status can get, given a certain segment class (m_p/n_p) of a predecessor segment p. The first two columns show the special case that at the beginning the sweep line status is empty and the first segment is inserted. This segment can either be shared by both region objects ($(0/2)$-segment) or stems from one of them ($(0/1)$-segment). In all these cases (except the first two cases), $n_p = m_s$ must hold.

Let l be the number of attributed halfsegments of F, m be the number of attributed halfsegments of G, and k be the total number of new halfsegments created due to our splitting strategy. The while-loop is then executed at most $l+m+k$ times. All operations needed on the sweep line status require at most $O(\log(l+m+k))$ time each. The operations on the halfsegment sequences of F and G need constant time. In total, the algorithm requires $O((l+m+k)\log(l+m+k))$ time and $O(l+m+k)$ space.

6 Direct Predicate Characterization as a Simple Evaluation Method

In the previous section we have developed sophisticated and efficient algorithms for the computation of the topological feature vectors v_F and v_G of two complex spatial objects $F, G \in \{point2D, line2D, region2D\}$. The vectors v_F and v_G contain specific topological feature flags for each type combination. The flags capture all topological situations between F and G and are different for different type combinations. Consequently, each type combination has led to a different exploration algorithm.

$$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} \qquad \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix}$$

(a) (b)

Figure 18: The 9-intersection matrix number 8 for the predicate *meet* between two *line2D* objects (a) and the 9-intersection matrix number 7 for the predicate *inside* between two *region2D* objects (b)

In this section, we present a first method for the *evaluation phase* whose objective it is to uniquely characterize the topological relationship of two given spatial objects of any type combination. Our general evaluation strategy is to leverage the information kept in the topological feature vectors and accommodate the objects' mutual topological features with an existing topological predicate for both predicate verification and predicate determination. The method presented here provides a *direct predicate characterization* of all s topological predicates of each type combination (see Table 1(b) for the different values of s) that is based on the feature values of v_F and v_G of the two spatial argument objects F and G. For example, for the *line2D*/*line2D* case, we have to determine which topological flags of v_F and v_G must be turned on and which flags must be turned off so that a given topological predicate (verification query) or a predicate to be found (determination query) is fulfilled. For the *region2D*/*region2D* case, the central question is to which segment classes the segments of both objects must belong so that a given topological predicate or a predicate to be found is satisfied. The direct predicate characterization gives an answer for each individual predicate of each individual type combination. This means that we obtain 184 individual predicate characterizations without converse predicates and 248 individual predicate characterizations with converse predicates. In general, each characterization is a Boolean expression in conjunctive normal form and expressed in terms of the topological feature vectors v_F and v_G.

We give two examples of direct predicate characterizations. As a first example, we consider the topological predicate number 8 (*meet*) between two *line2D* objects F and G (Figure 18(a) and [45]) and see how the flags of the topological feature vectors (Definition 4) are used.

$$\begin{aligned} p_8(F,G) \quad :\Leftrightarrow \quad & \neg v_F[\textit{seg\_shared}] \wedge \neg v_F[\textit{interior\_poi\_shared}] \wedge v_F[\textit{seg\_unshared}] \wedge \\ & \neg v_F[\textit{bound\_on\_interior}] \wedge v_F[\textit{bound\_shared}] \wedge v_F[\textit{bound\_disjoint}] \wedge \\ & v_G[\textit{seg\_unshared}] \wedge \neg v_G[\textit{bound\_on\_interior}] \wedge v_G[\textit{bound\_disjoint}] \end{aligned}$$

If we take into account the semantics of the topological feature flags, the right side of the equivalence means that both objects may only and must share boundary parts. More precisely and by considering the matrix in Figure 18(a), intersections between both interiors ($\neg v_F[\textit{seg\_shared}]$, $\neg v_F[\textit{interior\_poi\_shared}]$) as well as between the boundary of one object and the interior of the other object ($\neg v_F[\textit{bound\_on\_interior}]$, $\neg v_G[\textit{bound\_on\_interior}]$) are not allowed; besides intersections between both boundaries ($v_F[\textit{bound\_shared}$), each component of one object must interact with the exterior of the other object ($v_F[\textit{seg\_unshared}]$, $v_G[\textit{seg\_unshared}]$, $v_F[\textit{bound\_disjoint}]$, $v_G[\textit{bound\_disjoint}]$).

Next, we view the topological predicate number 7 (*inside*) between two *region2D* objects F and G (Figure 18(b) and [45]) and see how the segment classes kept in the topological feature vectors (Definition 6) are used.

$$\begin{aligned} p_7(F,G) \quad :\Leftrightarrow \quad & \neg v_F[(0/1)] \wedge \neg v_F[(1/0)] \wedge \neg v_F[(0/2)] \wedge \neg v_F[(2/0)] \wedge \neg v_F[(1/1)] \wedge \\ & \neg v_F[\textit{bound\_poi\_shared}] \wedge (v_F[(1/2)] \vee v_F[(2/1)]) \wedge \\ & \neg v_G[(1/2)] \wedge \neg v_G[(2/1)] \wedge (v_G[(0/1)] \vee v_G[(1/0)]) \end{aligned}$$

For the *inside* predicate, the segments of F must be located inside of G since the interior and boundary of F must be located in the interior of G; hence they must all have the segment classes $(1/2)$ or $(2/1)$. This

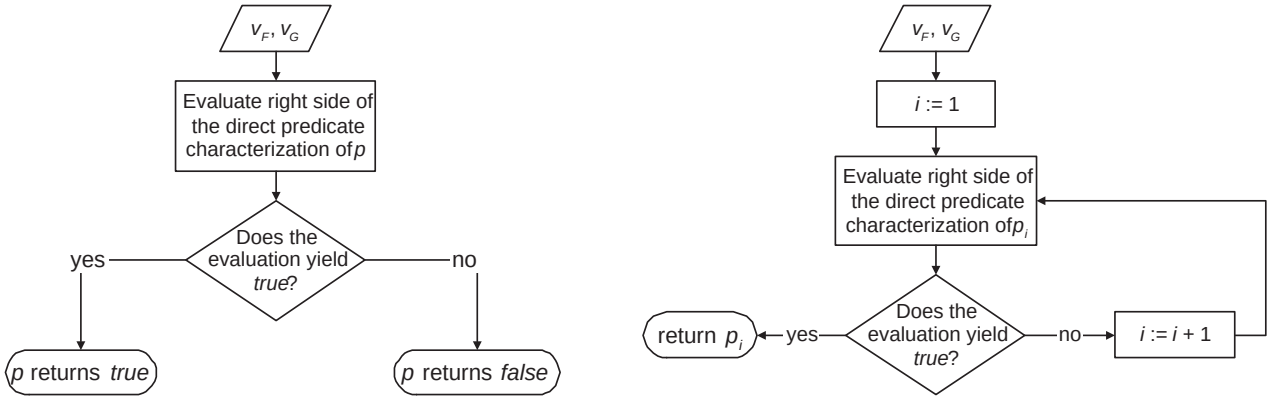


Figure 19: Predicate verification (a) and predicate determination (b) based on the direct predicate characterization method

"for all" quantification is tested by checking whether $v_F[(1/2)]$ or $v_F[(2/1)]$ are *true* and whether *all* other vector fields are *false*. The fact that all other vector fields are *false* means that the interior and boundary of F do not interact with the boundary and exterior of G. That is, the segments of G must be situated outside of F, and thus they *all* must have the segment classes $(0/1)$ or $(1/0)$; other segment classes are forbidden for G. Further, we must ensure that no segment of F shares a common point with any segment of G ($\neg v_F[bound\_poi\_shared]$).

The predicate characterizations can be read in both directions. If we are interested in *predicate verification*, i.e., in evaluating a specific topological predicate, we look from left to right and check the respective right side of the predicate's direct characterization (Figure 19(a)). This corresponds to an explicit implementation of each individual predicate. If we are interested in *predicate determination*, i.e., in deriving the topological relationship from a given spatial configuration of two spatial objects, we have to look from right to left. That is, consecutively we evaluate the right sides of the predicate characterizations by applying them to the given topological feature vectors v_F and v_G. For the characterization that matches we look on its left side to obtain the name or number of the predicate (Figure 19(b)).

The direct predicate characterization demonstrates how we can leverage the concept of topological feature vectors. However, this particular evaluation method has three main drawbacks. First, the method depends on the number of topological predicates. That is, each of the 184 (248) topological predicates between complex spatial objects requires an own specification. Second, in the worst case, all direct predicate characterizations with respect to a particular type combination have to be checked for predicate determination. Third, the direct predicate characterization is error-prone. It is difficult to ensure that each predicate characterization is correct and unique and that all predicate characterizations together are mutually exclusive and cover all topological predicates. From this standpoint, this solution is an *ad hoc* approach. In [37] we present a sophisticated approach that avoids these shortcomings and has a formal foundation.

7 Implementation and Testing of the Approach

To verify the feasibility, practicality, and correctness of the concepts presented, we have implemented and tested our approach. The system implementation of the algebra package SPAL2D (Section 3.1) for handling two-dimensional spatial data includes the implementation of all six exploration algorithms from Section 5. These algorithms are implemented at the highest level SDT2D (Figure 3). The implementation makes use

of the complex spatial data types *point2D*, *line2D*, and *region2D*, as they have been specified in Section 3.4. Since performance is one of the goals of this implementation, C++ is our employed programming language.

A universal interface method *TopPredExploration* is provided to explore the topological events of two interacting spatial objects. This interface is overloaded to accept two spatial objects of any type combination as input. The output consists of two topological feature vectors which hold the topological information for both argument objects. This information is then used to identify the corresponding topological predicate. Since the direct predicate characterization method has proved to be an ad hoc, error-prone, and inefficient solution for this purpose, we have not fully implemented this method (i.e., all 184 characterizations). Instead, an efficient, well-founded, and systematical method described in [37] has been implemented for this purpose.

Our implementation incorporating the data structures for the spatial data types and the exploration algorithms underwent various tests in order to verify the correctness of the concepts. We use a mixture of *black-box*[3] and *white-box*[4] testing techniques known as *gray-box*[5] testing. The black-box testing part arranges for well defined input and output objects. Each input object is a correct element of one of our spatial data types. Each output is guaranteed to be a topological feature vector and is tailored to the respective type combination. This enables us to test the functional behavior of our implementation by designing a collection of test cases to cover all type combinations of spatial objects as input and all possible values (*true* and *false*) of all topological feature flags as output. The white-box testing part considers every single execution path and guarantees that each statement is executed at least once. This ensures that all special cases that are specified and handled by the algorithms are properly tested. Our collection of test cases consists of 184 different scenes corresponding to the total number of topological predicates between spatial objects. They have been successfully tested and verified and indicate the correctness of our concepts and the ability of our algorithms to correctly discover the needed topological information from any given scene.

A special test case generation technique has been leveraged to check the functionality of the exploration algorithms and the correctness of the resulting values of the topological feature vectors. The vector values have to be independent of the location of the two spatial objects involved. In order to check this, this technique is able to generate arbitrarily many different orientations of a topologically identical scene of two spatial objects with respect to the same sweep line and coordinate system. The idea is to rotate such a scene by a random, rational angle around a central reference point. Special test cases including vertical segments or a meeting situation like in Figure 16 are considered too. For this rotation scenario, we especially take care of maintaining the robustness of geometric computation and preserving the topological consistency of a scene. We ensure that segment end points are rotated to end points with rational coordinates (and not floating point coordinates) by involving the Newton Method for approximating the results of numerical computations by rational numbers from our special rational number system RATIO (Section 3.1). Intersecting segments are broken up into four segments in advance.

Starting with a set of 184 explicitly constructed base cases (one for each topological predicate) which cover the special test cases if possible, we have generated at least 20,000 test cases for the topological predicates of each of the six type combinations by our random scene rotation technique. In total, more than 120,000 test cases have been successfully generated, tested, and checked in the exploration phase. All test cases have been checked for predicate verification and predicate determination.

[3]Black box testing takes an external perspective of the test object to derive test cases and does not consider the internal structure of the test object. The test designer selects valid and invalid input, determines the correct output, and checks the correctness of the functional behavior of the test object.

[4]White box testing takes an internal perspective of the test object and designs test cases based on its internal structure. The tester chooses test case inputs to exercise all paths through the software and determines the appropriate outputs. It requires programming skills to identify all paths through the software.

[5]Gray box testing maintains the advantages of black-box and white-box testing and is not impeded by the limitations of each particular one.

8 Conclusions and Future Work

While, from a conceptual perspective, topological predicates have been investigated to a large extent, the design of efficient implementation methods for them has been widely neglected. Especially due to the introduction of complex spatial data types, the resulting increase of topological predicates between them asks for efficient and correct implementation concepts. Our approach has been implemented as part of our SPAL2D software library which is a sophisticated spatial type system currently under development and determined for an integration into extensible databases. We propose a two-phase approach which consists of an exploration phase and an evaluation phase and which can be applied to both predicate verification and predicate determination. The goal of the exploration phase is to traverse a given scene of two objects in space, collect any topological information of importance, and store it in topological feature vectors. The goal of the evaluation phase is to interpret the gained topological information and match it against the topological predicates. In this article, we have focused on the exploration phase and in detail developed exploration algorithms for all type combinations.

Future work refers to the design of efficient and correct evaluation methods that interpret and match the information provided by the topological feature vectors against the topological predicates. In this article, we have presented the direct characterization method as a first solution. However, we have also pointed out its shortcomings. In [37] we propose evaluation methods that are robust, correct, and independent of the number of topological predicates of a particular type combination and that have a formal basis.

References

[1] J. F. Allen. Maintaining Knowledge about Temporal Intervals. *Communications of the ACM*, 26:832–843, 1983.

[2] T. Behr and M. Schneider. Topological Relationships of Complex Points and Complex Regions. *Int. Conf. on Conceptual Modeling*, pp. 56–69, 2001.

[3] M. de Berg, M. van Krefeld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 2nd edition, 2000.

[4] A. Brodsky and X. S. Wang. On Approximation-based Query Evaluation, Expensive Predicates and Constraint Objects. *Int. Workshop on Constraints, Databases, and Logic Programming*, 1995.

[5] J. Claussen, A. Kemper, G. Moerkotte, K. Peithner, and M. Steinbrunn. Optimization and Evaluation of Disjunctive Queries. *IEEE Trans. on Knowledge and Data Engineering*, 12(2):238–260, 2000.

[6] E. Clementini and P. Di Felice. A Model for Representing Topological Relationships between Complex Geometric Features in Spatial Databases. *Information Systems*, 90(1-4):121–136, 1996.

[7] E. Clementini and P. Di Felice. Topological Invariants for Lines. *IEEE Trans. on Knowledge and Data Engineering*, 10(1), 1998.

[8] E. Clementini, P. Di Felice, and G. Califano. Composite Regions in Topological Queries. *Information Systems*, 20(7):579–594, 1995.

[9] E. Clementini, P. Di Felice, and P. van Oosterom. A Small Set of Formal Topological Relationships Suitable for End-User Interaction. *3rd Int. Symp. on Advances in Spatial Databases*, LNCS 692, pp. 277–295, 1993.

[10] E. Clementini, J. Sharma, and M.J. Egenhofer. Modeling Topological Spatial Relations: Strategies for Query Processing. *Computers and Graphics*, 18(6):815–822, 1994.

[11] Z. Cui, A. G. Cohn, and D. A. Randell. Qualitative and Topological Relationships. *3rd Int. Symp. on Advances in Spatial Databases*, LNCS 692, pp. 296–315, 1993.

[12] J.R. Davis. IBM's DB2 Spatial Extender: Managing Geo-Spatial Information within the DBMS. Technical report, IBM Corporation, 1998.

[13] M. J. Egenhofer. A Formal Definition of Binary Topological Relationships. *3rd Int. Conf. on Foundations of Data Organization and Algorithms*, LNCS 367, pp. 457–472. Springer-Verlag, 1989.

[14] M. J. Egenhofer. Definitions of Line-Line Relations for Geographic Databases. *16th Int. Conf. on Data Engineering*, pp. 40–46, 1993.

[15] M. J. Egenhofer. Spatial SQL: A Query and Presentation Language. *IEEE Trans. on Knowledge and Data Engineering*, 6(1):86–94, 1994.

[16] M. J. Egenhofer and R. D. Franzosa. Point-Set Topological Spatial Relations. *Int. Journal of Geographical Information Systems*, 5(2):161–174, 1991.

[17] M. J. Egenhofer and J. Herring. A Mathematical Framework for the Definition of Topological Relationships. *4th Int. Symp. on Spatial Data Handling*, pp. 803–813, 1990.

[18] M. J. Egenhofer and J. Herring. Categorizing Binary Topological Relations Between Regions, Lines, and Points in Geographic Databases. Technical Report 90-12, National Center for Geographic Information and Analysis, University of California, Santa Barbara, 1990.

[19] M. J. Egenhofer and D. Mark. Modeling Conceptual Neighborhoods of Topological Line-Region Relations. *Int. Journal of Geographical Information Systems*, 9(5):555–565, 1995.

[20] M.J. Egenhofer. Deriving the Composition of Binary Topological Relations. *Journal of Visual Languages and Computing*, 2(2):133–149, 1994.

[21] M.J. Egenhofer, E. Clementini, and P. Di Felice. Topological Relations between Regions with Holes. *Int. Journal of Geographical Information Systems*, 8(2):128–142, 1994.

[22] ESRI Spatial Database Engine (SDE). Environmental Systems Research Institute, Inc., 1995.

[23] S. Gaal. *Point Set Topology*. Academic Press, 1964.

[24] V. Gaede and O. Günther. Multidimensional Access Methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[25] R. H. Güting. Geo-Relational Algebra: A Model and Query Language for Geometric Database Systems. *Int. Conf. on Extending Database Technology*, pp. 506–527, 1988.

[26] R. H. Güting and M. Schneider. Realms: A Foundation for Spatial Data Types in Database Systems. *3rd Int. Symp. on Advances in Spatial Databases*, LNCS 692, pp. 14–35. Springer-Verlag, 1993.

[27] R. H. Güting and M. Schneider. Realm-Based Spatial Data Types: The ROSE Algebra. *VLDB Journal*, 4:100–143, 1995.

[28] R.H. Güting, T. de Ridder, and M. Schneider. Implementation of the ROSE Algebra: Efficient Algorithms for Realm-Based Spatial Data Types. *Int. Symp. on Advances in Spatial Databases*, 1995.

[29] J. M. Hellerstein. Practical Predicate Placement. *ACM SIGMOD Int. Conf. on Management of Data*, pp. 325–335.

[30] J. M. Hellerstein and M. Stonebraker. Predicate Migration: Optimizing Queries with Expensive Predicates. *ACM SIGMOD Int. Conf. on Management of Data*, pp. 267–276, 1993.

[31] Informix Geodetic DataBlade Module: User's Guide. Informix Press, 1997.

[32] JTS Topology Suite. Vivid Solutions. URL: http://www.vividsolutions.com/JTS/JTSHome.htm.

[33] OGC Abstract Specification. OpenGIS Consortium (OGC), 1999. URL: http://www.opengis.org/techno/specs.htm.

[34] OGC Geography Markup Language (GML) 2.0. OpenGIS Consortium (OGC), 2001. URL: http://www.opengis.net/gml/01-029/GML2.html.

[35] Oracle8: Spatial Cartridge. An Oracle Technical White Paper. Oracle Corporation, 1997.

[36] J. A. Orenstein and F. A. Manola. PROBE Spatial Data Modeling and Query Processing in an Image Database Application. *IEEE Trans. on Software Engineering*, 14:611–629, 1988.

[37] R. Praing and M. Schneider. Efficient Implementation Techniques for Topological Predicates on Complex Spatial Objects: The Evaluation Phase. Technical report, University of Florida, Department of Computer & Information Science & Engineering, 2006.

[38] F. P. Preparata and M. I. Shamos. *Computational Geometry*. Springer Verlag, 1985.

[39] M. A. Rodriguez, M. J. Egenhofer, and A. D.Blaser. Query Pre-Processing of Topological Constraints: Comparing a Composition-Based with Neighborhood-Based Approach. *Int. Symp. on Spatial and Temporal Databases*, LNCS 2750, pp. 362–379. Springer-Verlag, 2003.

[40] N. Roussopoulos, C. Faloutsos, and T. Sellis. An Efficient Pictorial Database System for PSQL. *IEEE Trans. on Software Engineering*, 14:639–650, 1988.

[41] M. Schneider. *Spatial Data Types for Database Systems - Finite Resolution Geometry for Geographic Information Systems*, volume LNCS 1288. Springer-Verlag, Berlin Heidelberg, 1997.

[42] M. Schneider. Implementing Topological Predicates for Complex Regions. *Int. Symp. on Spatial Data Handling*, pp. 313–328, 2002.

[43] M. Schneider. Computing the Topological Relationship of Complex Regions. *15th Int. Conf. on Database and Expert Systems Applications*, pp. 844–853, 2004.

[44] M. Schneider and T. Behr. Topological Relationships between Complex Lines and Complex Regions. *Int. Conf. on Conceptual Modeling*, 2005.

[45] M. Schneider and T. Behr. Topological Relationships between Complex Spatial Objects. *ACM Trans. on Database Systems*, 31(1):39–81, 2006.

[46] M.F. Worboys and P. Bofakos. A Canonical Model for a Class of Areal Spatial Objects. *3rd Int. Symp. on Advances in Spatial Databases*, LNCS 692, pp. 36–52. Springer-Verlag, 1993.