Efficient Evaluation Techniques for Topological Predicates on Complex Regions

Reasey Praing and Markus Schneider *

University of Florida, Department of Computer and Information Science & Engineering
{rpraing,mschneid}@cise.ufl.edu

Summary. Topological predicates between spatial objects have for a long time been a focus of intensive research in a number of diverse disciplines. In the context of spatial databases and geographical information systems, they support the construction of suitable query languages for spatial data retrieval and analysis. Whereas to a large extent conceptual aspects of topological predicates have been emphasized, the development of efficient evaluation 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 paper is to develop efficient implementation techniques for them within the framework of the spatial algebra SPAL2D.

Key words: topological predicates, spatial relationships, efficient evaluation, complex spatial objects, complex regions, predicate implementation

1 Introduction

Topological predicates between spatial objects have always been a main area of research in 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. The focus of this research has been on the conceptual design of and reasoning with these predicates. In contrast to this large amount of conceptual work, implementation issues for topological predicates have been widely neglected except for spatial index support as a pre-stage in query processing to identify candidate pairs of spatial objects that could possibly fulfill the predicate of interest. The main reason is probably the (simplifying) view that some plane-sweep algorithm is sufficient to implement topological predicates. Certainly a plane sweep plays an important role for the implementation of these predicates, but there are (at least) two aspects that make such

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

an implementation much more challenging. The first aspect refers to the details of the plane sweep itself. Issues are 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 [7]), how intersections are handled, and what kind of information the plane sweep must output so that this information can be leveraged for predicate determination. The second aspect deals with the kind of query posed. Given two spatial objects A and B, 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 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*. We will see that these two kinds of query benefit from two different evaluation procedures.

The goal of this paper is to present efficient implementation strategies for topological predicates on simple and complex regions within the framework of the spatial algebra SPAL2D. Section 2 discusses related work about spatial data types and available design and implementation concepts for topological predicates. For *predicate execution* we distinguish two phases: In an *exploration phase*, described in Section 3, a plane sweep scans a given configuration of two spatial objects and collects metadata that can help us later derive the topological relationship between both objects. In the next phase, called the *evaluation phase* and described in Section 4, these metadata are matched against characteristic properties of the topological predicates. This enables us to determine the Boolean result of a topological predicate (query 1) and the kind of topological predicate (query 2). Section 5 describes implementation results and performance analysis. Finally, Section 6 draws some conclusions.

2 Related Work

For the implementation of topological predicates we need two kinds of ingredients: a concept and implementation of spatial data types and a concept of topological predicates to be implemented. *Spatial data types* (see [7] for a survey) like *point*, *line*, and *region* have been accepted as fundamental abstractions for modeling the structure of geometric entities, their relationships, properties, and operations. Whereas in the past, spatial objects were only simple structures (single points, continuous lines, simple regions), the trend is now towards complex spatial objects allowing, e.g., multiple object components and holes in regions. The reason for this development lies in the need of applications and in the requirement of closure properties for spatial operations. Formal definitions of complex spatial data types can be found in [9]. Implementation descriptions of spatial data types are rare. Our implementation uses strategies found in [7].

The amount of conceptual work on *topological predicates* is large. The two most important approaches are the 9-*intersection model* [4] and the *Randell-Cui-Cohn model (RCC)* [3]. But since these approaches only deal with topological predicates for *simple* spatial objects, we have extended the approach in [4] to *complex* spatial

objects in [9], which is the basis of our implementation. As we move to complex spatial objects, the number of topological predicates increases significantly. This requires more sophisticated and efficient evaluation techniques. For two complex regions, one obtains 33 topological predicates whereas only 8 exist between two simple regions. The details about the determination process can be found in [9]. Table 1 shows the matrix representations of the 33 predicates.

Only a few papers have dealt with the execution and implementation of topological predicates. The paper in [2] uses an optimization technique similar to our matrix thinning technique in Section 4.3 to minimize the number of needed computations. Ad hoc implementations of topological predicates have been proposed in [8].

3 The Exploration Phase: Collecting Topological Information

In this section, we give an overview of a process of exploring topological information between two spatial objects (here: regions). The detail of this process is discussed in [8]. In this process, we scan a given configuration of the two objects in order to collect data that help us later to derive the topological relationship between both objects in the evaluation phase. The objective is to discover the topological information of each object represented by overlap numbers using the plane sweep paradigm. The concepts resemble those described in [6, 7] but are different in the sense that they are not realm-based and that they allow intersecting segments of the argument objects. Thus, we describe the general case. Section 3.1 depicts the data structure implemented for the *region* data type. Section 3.2 sketches some needed geometric concepts like parallel object traversal, overlap numbers, and plane sweep with an emphasis on special features as they are relevant to our context. Section 3.3 explains the algorithm combining these concepts and the output information provided for further analysis in the evaluation phase.

3.1 Data Structure for the *region* Data Type

In the implementation described in this paper, we employ a new rational arithmetic called RATIO. RATIO provides a data type *rational* whose value representations can be of arbitrary, finite length and are only limited by main memory. This ensures numerical robustness and topological consistency in our implementation. Conceptually, complex regions can be considered from a "structured" and a "flat" view. The structured view defines an object of type *region* as a set of edge-disjoint faces. A face is a simple polygon possibly containing a set of edge-disjoint holes. A hole is a simple polygon. Two simple polygons are *edge-disjoint* if their interiors are disjoint and they possibly share single boundary points but not boundary segments. The flat view defines an object of type *region* as a collection of line segments which altogether preserve the constraints and properties of the structured view. For a formal definition see [5].

All coordinates are given as numbers of RATIO's data type *rational*. A value of type *point* is represented by a record $p = (x, y)$ where x and y are coordinates.

We assume the usual lexicographical order on points. A *region* object is given as an *ordered* sequence (array) of *halfsegments*. Note that we omit all components of a *region* object representation that do not play a role in this context. The idea of halfsegments is to store each segment twice. A single segment corresponds to two halfsegments: a left halfsegment and a right halfsegment. A left (right) halfsegment is defined by the left/smaller (right/larger) point of the segment as its *dominating point*. The order relation on these halfsegments is defined first by the order of their dominating points, type, angle, and length. Detail on the formal definition of this order relation is discussed in [8].

An ordered sequence of halfsegments is represented as an array $\langle (h_1, a_1), \ldots, (h_m, a_m) \rangle$ of m halfsegments $h_i \in H$ with $h_i < h_j$ for all $1 \leq i < j \leq m$. Since inserting a halfsegment at an arbitrary position needs $O(m)$ time, in our implementation we use an AVL-tree embedded into an array whose elements are linked in sequence order. An insertion then requires $O(\log m)$ time. Each *left* halfsegment $h_i = (s_i, left)$ has an attached set a_i of *attributes*. Attributes contain auxiliary information that is needed by geometric algorithms. It is usually unnecessary to attach attributes to *right* halfsegments since their existence in an ordered halfsegment sequence only indicates to plane sweep algorithms that the respective segment has to be removed from the *sweep line status*; in our implementation they are omitted. In the case for a region object r, the *left* halfsegments carry an associated attribute *InsideAbove* where the interior of r lies above or left of their respective segments.

3.2 Parallel Object Traversal, Overlap Numbers, Plane Sweep

Three main and well understood concepts are needed for the algorithm to be designed: parallel object traversal, the concept of overlap numbers, and the well-known plane sweep paradigm. *Parallel object traversal* allows us, during the plane sweep, to traverse the halfsegment sequences of both region operands in parallel since each sequence is already in halfsegment sequence order. This avoids expensive, initial sorting. Hence, by employing a cursor on both sequences, it is sufficient to check the halfsegments at the current cursor positions of both sequences and to take the lower one with respect to halfsegment sequence order for further computation. Below we will describe a slight extension of the parallel object traversal in the sense that it will traverse four instead of two halfsegment sequences and return the smallest halfsegment from the four current cursor positions.

Finding out the degree of overlapping of region parts is important for determining the topological relationship between two complex regions. For this purpose, we employ the concept of *overlap numbers* [6]. A point has overlap number k if k regions contain this point. For two regions F and G, a point p obtains the overlap number 2, iff $p \in F$ and $p \in G$. It obtains the overlap number 1, iff either $p \in F$ and $p \in G^-$, or $p \in F^-$ and $p \in G$. Otherwise, its overlap number is 0. Since a segment of a region separates space into two parts, an inner and an exterior one, during a plane sweep each (half)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. The lower (upper) overlap number indicates the number of overlapping *region* objects below

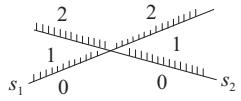

Fig. 1. Changing overlap numbers after an intersection.

(above) the segment. In this way, we obtain a *segment classification* of two *region* objects and speak about (m/n)-segments. Obviously, $m, n \leq 2$ holds.

Figure 1 shows the overlap numbers defined by 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 after the intersection point.

To preserve these benefits and to enable the use of overlap numbers also for arbitrary segments, in the case that two segments from different *region* objects 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 split at their common intersection point so that each of them is replaced by two segments (i.e., four halfsegments) (Figure 2a). 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 2b). 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 2c).

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

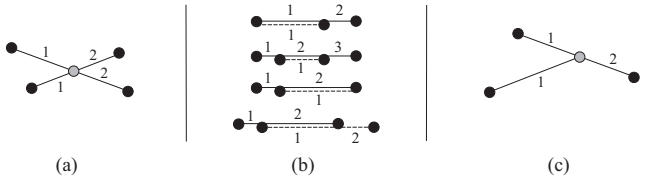

Fig. 2. Splitting of two intersecting segments (a), two partially coinciding segments (without symmetric counterparts) (b), and a segment whose interior is touched by another segment (c). Digits indicate part numbers of segments after splitting.

However, the splitting of segments entails some algorithmic effort. On the one hand, we want to keep the halfsegment sequences of the *region* objects unchanged since their update is expensive and is 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 *region* 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 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 additional $O(k)$ space, and to insert them requires $O(k \log k)$ time. After the plane sweep, this additional space is released.

3.3 Topological Exploration Algorithm for the Region/Region Case

Using the three aforementioned concepts, the main goal of the exploration algorithm is to determine the segment classification of each *region* object. Each object is assigned a *Boolean segment classification vector*. This vector contains a field for the segment classes $(0/1)$ and $(1/0)$, a field for $(0/2)$ and $(2/0)$, a field for $(1/2)$ and $(2/1)$, and a field for $(1/1)$. That is, symmetric segment classes are merged since only either their non-existence or the existence of at least one of the two classes is later relevant. Each field is initialized with *false*. An additional flag *point_in_common* indicates whether any two segments of both objects meet in a common endpoint.

The segment classification is determined with the same sweep line status structure of the plane sweep that handles necessary segment splits. The pseudocode below presents the algorithm for its computation. When we encounter a right halfsegment in the event point schedule, its segment is looked up in the sweep line status and removed. For a left halfsegment, its segment is inserted into the sweep line status according to the y-coordinate of its left endpoint. Its lower overlap number is assigned the upper overlap number of its predecessor and its upper overlap number is assigned its incremented or decremented lower overlap number depending on whether the flag *InsideAbove* is true or false, respectively.

The algorithm uses some auxiliary operations which we briefly describe. The first two operations, *rr_select_first* and *rr_select_next*, support parallel object traversal. These operations check which of the two *region* objects F and G (i.e., the first, the second, or both) has the smaller halfsegment. If the status of the traversal is equal to *end_of_both*, both object representations have been traversed. The other needed operations refer to management of the sweep line status, which is initialized with *new_sweep*. If a left (right) halfsegment of a *region* 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 operation *pred_exists* (*common_point_exists*) checks whether, for the segment currently considered in the sweep line status, a predecessor (a neighbored segment immediately below the current segment) exists. The operation *set_attr* (*get_pred_attr*) sets (gets) a set of attributes for (from the predecessor of) the segment currently considered in the sweep

line status. Finally, the operation *get_attr* yields the attributes associated with a halfsegment.

algorithm *SegmentClassification*
input: *region* objects F and G, initialized segment classification vectors v_F and v_G
output: updated vectors v_F and v_G
begin
  S := *new_sweep*(); *point_in_common* := *false*; *rr_select_first*(F, G, *object*, *status*);
  while (*status* $\neq$ *end_of_both*) **do**
    if (*object* = *first*) **or** (*object* = *both*) **then** h := *get_hs*(F) /* Let $h = (s, d)$. */
      else h := *get_hs*(G) **endif**; /* Let $h = (s, d)$. */
    if d = *right* **then** *del_right*(S, s) **else** *add_left*(S, s);
      point_in_common := *point_in_common* **or** *common_point_exists*(S);
      if not *pred_exists*(S) **then** $(m_p/n_p) := (*/0)$
        else $\{(m_p/n_p)\}$:= *get_pred_attr*(S) **endif**;
      $m_s := n_p$; $n_s := n_p$;
      if ((*object* = *first*) **or** (*object* = *both*)) **and** (*InsideAbove* $\in$ *get_attr*(F))
        then $n_s := n_s + 1$ **else** $n_s := n_s - 1$ **endif**;
      if ((*object* = *second*) **or** (*object* = *both*)) **and** (*InsideAbove* $\in$ *get_attr*(G))
        then $n_s := n_s + 1$ **else** $n_s := n_s - 1$ **endif**;
      S := *set_attr*(S, $\{(m_s/n_s)\}$);
      if (*object* = *first*) **then** $v_F[(m_s/n_s)]$:= *true*
      else if (*object* = *second*) **then** $v_G[(m_s/n_s)]$:= *true*
      else if (*object* = *both*) **then** $v_F[(m_s/n_s)]$:= *true*; $v_G[(m_s/n_s)]$:= *true*; **endif**
    endif;
    rr_select_next(F, G, *object*, *status*);
  endwhile
end *SegmentClassification*.

If F has l and G has m halfsegments, the while-loop is executed at most $n = l + m$ times, because each time a new halfsegment is visited. The most expensive operations within the loop are the insertion and the removal of a segment into and from the sweep line status. We implement the status structure by an AVL tree which realizes each of the two update operations in time $O(\log n)$ and the other operations in constant time. Since at most n elements can be contained in the sweep line status, the worst time complexity of the algorithm is $O(n \log n)$.

4 The Evaluation Phase: Matching Topological Relationships

So far, we are able to compute for two given complex regions F and G their segment classification vectors v_F and v_G. The values of both vectors depend on the relative positions of F and G to each other. The existence or non-existence of a segment class in v_F and v_G, respectively, conveys a topological information. For example, if v_F indicates the existence of $(0/1)$ segments in F, we can conclude that F contains segments that are located outside of G. But this does not enable us to derive the topological predicate between both objects, since it can be a disjoint-like, overlap-like, or covers-like relationship. To be able to decide this, we need to consider more

information in both vectors. Before we get to the actual evaluation process, we give a definition for segment classification to better understand the semantic behind these classifications.

Definition 1. *The possible segment classes for a segment s of a complex region F with respect to another complex region G are given by a function* SC *as follows:*

$$SC(s,F;G) = \begin{cases} (0/1) \text{ iff } s \in \partial F \ \wedge \ IA(s,F) \ \wedge \ s \in G^- \\ (1/0) \text{ iff } s \in \partial F \ \wedge \ \neg IA(s,F) \ \wedge \ s \in G^- \\ (1/2) \text{ iff } s \in \partial F \ \wedge \ IA(s,F) \ \wedge \ s \in G^\circ \\ (2/1) \text{ iff } s \in \partial F \ \wedge \ \neg IA(s,F) \ \wedge \ s \in G^\circ \\ (0/2) \text{ iff } s \in \partial F \ \wedge \ IA(s,F) \ \wedge \ s \in \partial G \ \wedge \ IA(s,G) \\ (2/0) \text{ iff } s \in \partial F \ \wedge \ \neg IA(s,F) \ \wedge \ s \in \partial G \ \wedge \ \neg IA(s,G) \\ (1/1) \text{ iff } s \in \partial F \ \wedge \ s \in \partial G \ \wedge \ ((IA(s,F) \ \wedge \ \neg IA(s,G)) \vee \\ \qquad (\neg IA(s,F) \ \wedge \ IA(s,G))) \end{cases}$$

Of the nine possible combinations only seven describe valid segment classes. This is because a $(0/0)$-segment contradicts the definition of a complex *region* object, since then at least one of both regions would have two holes or an outer cycle and a hole with a common border. 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.

With this definition, we can establish a connection to the topological predicates by constructing an evaluation technique called 9-*intersection matrix* (9-*IM*) *characterization* (Section 4.1). This technique is used to answer both verification and determination queries. Section 4.2 describes a fine-tuned approach called *minimum cost decision tree* for predicate determination. Section 4.3 delineates a sophisticated approach called *matrix thinning* for predicate verification.

4.1 9-Intersection Matrix Characterization

Instead of characterizing each topological predicate directly, the idea of this approach is to uniquely characterize each element of the 3×3-matrix of the 9-intersection model [4]. Such an element is a predicate that checks one of the nine intersections between the boundary ∂F, interior F°, or exterior F^- of a spatial object F with the boundary ∂G, interior G°, or exterior G^- of another spatial object G for inequality to the empty set (see the left sides of the equivalence of Theorem 1). We call such an element *matrix predicate*. For each topological predicate, its specification is then given as the conjunction of the characterizations of the nine matrix predicates. In the region/region case, a *matrix predicate characterization* is again performed on the basis of a segment classification and finally on the regions' segment classification vectors. The goal of the following lemmas is to lead us to a unique characterization of all matrix predicates by means of segment classes. In all lemmas, let $H(F)$ and $H(G)$ be the set of all halfsegments (including any split halfsegment) in F and G respectively. The first lemma provides a translation of each segment class into a matrix predicate. Due to space limitations, detailed proofs are omitted.

Lemma 1. *Let F and G be two complex regions. Then we can infer the following implications and partial equivalences between segment classes and matrix predicates:*

$$
\begin{array}{rll}
\text{(i)} & \exists\, h \in H(F) : SC(h, F; G) \in \{(0/1), (1/0)\} & \Leftrightarrow \partial F \cap G^{-} \neq \varnothing \\
\text{(ii)} & \exists\, g \in H(G) : SC(g, G; F) \in \{(0/1), (1/0)\} & \Leftrightarrow F^{-} \cap \partial G \neq \varnothing \\
\text{(iii)} & \exists\, h \in H(F) : SC(h, F; G) \in \{(1/2), (2/1)\} & \Leftrightarrow \partial F \cap G^{\circ} \neq \varnothing \\
\text{(iv)} & \exists\, g \in H(G) : SC(g, G; F) \in \{(1/2), (2/1)\} & \Leftrightarrow F^{\circ} \cap \partial G \neq \varnothing \\
\text{(v)} & \exists\, h \in H(F) : SC(h, F; G) \in \{(0/2), (2/0)\} & \Rightarrow \partial F \cap \partial G \neq \varnothing \ \wedge\ F^{\circ} \cap G^{\circ} \neq \varnothing \\
\text{(vi)} & \exists\, g \in H(G) : SC(g, G; F) \in \{(0/2), (2/0)\} & \Leftrightarrow \exists\, h \in H(F) : \\
 & & \quad SC(h, F; G) \in \{(0/2), (2/0)\} \\
\text{(vii)} & \exists\, h \in H(F) : SC(h, F; G) \in \{(1/1)\} & \Rightarrow \partial F \cap \partial G \neq \varnothing \ \wedge\ F^{\circ} \cap G^{-} \neq \varnothing \\
 & & \quad \wedge\, F^{-} \cap G^{\circ} \neq \varnothing \\
\text{(viii)} & \exists\, g \in H(G) : SC(g, G; F) \in \{(1/1)\} & \Leftrightarrow \exists\, h \in H(F) : \\
 & & \quad SC(h, F; G) \in \{(1/1)\}
\end{array}
$$

The proof for this lemma follows directly from the segment classification definition (Definition 1) and the definition of complex region in [9]. The second lemma provides a translation of some matrix predicates into segment classes.

Lemma 2. *Let F and G be two complex regions. Then we can infer the following implications between matrix predicates and segment classes:*

$$
\begin{array}{rl}
\text{(i)} & F^{\circ} \cap G^{\circ} \neq \varnothing \ \Rightarrow \exists\, h \in H(F) : SC(h, F; G) \in \{(0/2), (2/0), (1/2), (2/1)\} \vee \\
 & \qquad \exists\, g \in H(G) : SC(g, G; F) \in \{(0/2), (2/0), (1/2), (2/1)\} \\
\text{(ii)} & F^{\circ} \cap G^{-} \neq \varnothing \Rightarrow \exists\, h \in H(F) : SC(h, F; G) \in \{(0/1), (1/0), (1/1)\} \vee \\
 & \qquad \exists\, g \in H(G) : SC(g, G; F) \in \{(1/2), (2/1), (1/1)\} \\
\text{(iii)} & F^{-} \cap G^{\circ} \neq \varnothing \Rightarrow \exists\, h \in H(F) : SC(h, F; G) \in \{(1/2), (2/1), (1/1)\} \vee \\
 & \qquad \exists\, g \in H(G) : SC(g, G; F) \in \{(0/1), (1/0), (1/1)\}
\end{array}
$$

This lemma is proved using the ovelap number concept in conjunction with complex region definition. The third lemma states some implications between matrix predicates.

Lemma 3. *Let F and G be two complex regions. Then we can infer the following implications between matrix predicates:*

$$
\begin{array}{rll}
\text{(i)} & point\_in\_common & \Rightarrow \partial F \cap \partial G \neq \varnothing \\
\text{(ii)} & \partial F \cap G^{-} \neq \varnothing & \Rightarrow F^{\circ} \cap G^{-} \neq \varnothing \ \wedge\ F^{-} \cap G^{-} \neq \varnothing \\
\text{(iii)} & F^{-} \cap \partial G \neq \varnothing & \Rightarrow F^{-} \cap G^{\circ} \neq \varnothing \ \wedge\ F^{-} \cap G^{-} \neq \varnothing \\
\text{(iv)} & \partial F \cap G^{\circ} \neq \varnothing & \Rightarrow F^{\circ} \cap G^{\circ} \neq \varnothing \ \wedge\ F^{-} \cap G^{\circ} \neq \varnothing \\
\text{(v)} & F^{\circ} \cap \partial G \neq \varnothing & \Rightarrow F^{\circ} \cap G^{\circ} \neq \varnothing \ \wedge\ F^{\circ} \cap G^{-} \neq \varnothing
\end{array}
$$

The proof for this lemma is based on the definition of $point\_in\_common$ and point set topological concepts found in [9]. The following theorem collects the results we have obtained so far and proves the lacking parts of the nine matrix predicate characterizations.

Theorem 1. *Let F and G be two complex regions. Let $H(F)$ and $H(G)$ be the set of possibly split halfsegments of F and G. Then the matrix predicates of the 9-intersection matrix are equivalent to the following segment class characterizations:*

$$
\begin{aligned}
&\text{(i)}\ F^\circ \cap G^\circ \neq \varnothing \ \Leftrightarrow\ & &\exists\, h \in H(F) : SC(h, F; G) \in \{(0/2), (2/0), (1/2), (2/1)\} \vee \\
& & &\exists\, g \in H(G) : SC(g, G; F) \in \{(0/2), (2/0), (1/2), (2/1)\} \\
&\text{(ii)}\ F^\circ \cap \partial G \neq \varnothing \ \Leftrightarrow\ & &\exists\, g \in H(G) : SC(g, G; F) \in \{(1/2), (2/1)\} \\
&\text{(iii)}\ F^\circ \cap G^- \neq \varnothing \ \Leftrightarrow\ & &\exists\, h \in H(F) : SC(h, F; G) \in \{(0/1), (1/0), (1/1)\} \vee \\
& & &\exists\, g \in H(G) : SC(g, G; F) \in \{(1/2), (2/1), (1/1)\} \\
&\text{(iv)}\ \partial F \cap G^\circ \neq \varnothing \ \Leftrightarrow\ & &\exists\, h \in H(F) : SC(h, F; G) \in \{(1/2), (2/1)\} \\
&\text{(v)}\ \partial F \cap \partial G \neq \varnothing \ \Leftrightarrow\ & &\exists\, h \in H(F) : SC(h, F; G) \in \{(0/2), (2/0), (1/1)\} \vee \\
& & &\exists\, g \in H(G) : SC(g, G; F) \in \{(0/2), (2/0), (1/1)\} \vee \\
& & &point\_in\_common \\
&\text{(vi)}\ \partial F \cap G^- \neq \varnothing \ \Leftrightarrow\ & &\exists\, h \in H(F) : SC(h, F; G) \in \{(0/1), (1/0)\} \\
&\text{(vii)}\ F^- \cap G^\circ \neq \varnothing \ \Leftrightarrow\ & &\exists\, h \in H(F) : SC(h, F; G) \in \{(1/2), (2/1), (1/1)\} \vee \\
& & &\exists\, g \in H(G) : SC(g, G; F) \in \{(0/1), (1/0), (1/1)\} \\
&\text{(viii)}\ F^- \cap \partial G \neq \varnothing \ \Leftrightarrow\ & &\exists\, g \in H(G) : SC(g, G; F) \in \{(0/1), (1/0)\} \\
&\text{(ix)}\ F^- \cap G^- \neq \varnothing \ \Leftrightarrow\ & &\textit{true}
\end{aligned}
$$

Theorem 1 provides us with a unique characterization of each individual matrix predicate of the 9-intersection matrix. This approach has several benefits. First, it is systematic and has a formal and sound foundation. Hence, we can be sure about the correctness of segment classes assigned to matrix predicates, and vice versa. Second, this evaluation method is independent of the number of topological predicates and only requires a constant number of evaluations for matrix predicate characterizations. Instead of nine, even only eight matrix predicates have to be checked since $F^- \cap G^- \neq \varnothing$ is always true (Theorem1(ix)). Third, we have proved the correctness of our corresponding implementation.

Based on this result, we can perform the predicate verification for a topological predicate p on the basis of p's 9-intersection matrix (see Table 1). In the case of a value 1 (*true*) for a matrix predicate, we take its equivalent, assigned segment classification on the right side in Theorem 1 and match it with the segment classification vectors v_F and v_G computed in the exploration phase. If there is a match, we proceed with the next value and matrix predicate in the 9-intersection matrix; otherwise p is *false*. In the case of a value 0 (*false*) for a matrix predicate, we pursue the same strategy but have to negate the assigned segment classification on the right side in Theorem 1 first.

For a predicate determination we take the following approach: For the first topological predicate p, we begin with the segment class characterization of the first matrix predicate on the right side in Theorem 1 and match it with v_F and v_G. For a value 1 for the matrix predicate, this means that v_F and v_G must satisfy the segment class characterization on the right side of the matrix predicate. For a value 0 of the matrix predicate, v_F and v_G must satisfy the negated segment class characterization. If they match, we know that this matrix predicate is satisfied and we continue with the segment class characterization of the next matrix predicate. Otherwise, p is *false* and we perform the whole procedure with the next topological predicate. In the worst case, this requires 33 tests of topological predicates.

4.2 The *MinCostDecisionTree* Algorithm

In this and the next section we fine-tune the approach of Section 4.1. A first observation is that for predicate determination we have to test all 33 topological predicates in the worst case. We propose a concept called *minimum cost decision tree* (MCDT) in this section which avoids this drawback and is similar to a technique introduced in [2] for topological predicates for simple regions. The idea is to construct a binary decision tree whose inner nodes are matrix predicates and whose leaf nodes are the 33 topological predicates. The tree partitions the search space at each node and progressively excludes more and more other topological predicates. In the best case, at each node of the decision tree the search space is partitioned into two halves. This requires $\log s$ computations where s is the number of topological predicates. For $s = 33$ the height of the tree is at least 6.

The pseudocode below shows our recursive algorithm for computing a minimum cost decision tree. Assuming that all topological relationships occur with equal probability, our cost model is to sum up all the path lengths from each topological predicate to the root. The algorithm takes as input the list of intersection matrices of the topological predicates (Table 1). These matrices later become the leaves of the decision tree. In addition, a node list is required such that the algorithm may skip those decision branch elements that already appeared in the node path. This node list is empty at the start of the program and updated for every recursive call. The algorithm constructs the best tree by traversing through each valid decision branch using depth first search as it makes recursive calls. The recursion stack ends once a leaf is found, and at this point we have a sub-tree for which we can calculate the total cost. Then the recursion returns and recursively finds the next leaf. By comparing each alternative minimum cost sub-tree from each decision branch at a level, we obtain the minimum cost sub-trees at the parent level of the current level. This comparison takes place from the bottom up until the complete minimum cost tree is constructed and the root is chosen.

algorithm *MinCostDecisionTree*
input: A matrix list mat[] and a node list nodeList[]
output: The root node of a minimum cost decision tree.
begin
  $element := firstElement(); bestNode := newNode();$
  while ($element.isValid$) **do** $node := newNode(element);$
    if ($node.isUnary$) **then** $continue;$
    else if ($node.isLeaf$) **then** $bestNode.id := mat[0].id$; $bestNode.cost := 0$; $break;$
    else $nodeList.add(node)$; $node.lChild :=$ *MinCostTree*$(node.lChildren, nodeList);$
      $node.rChild :=$ *MinCostTree*$(node.rChildren, nodeList);$
      $node.cost := node.lChild.cost + node.rChild.cost + node.lChildren.length +$
        $node.rChildren.length;$
      if ($node.cost < bestNode.cost$) **then** $bestNode := node;$ **endif**;
    endif; $element := nextElement();$
  endwhile;
  return $bestNode$;
end *MinCostDecisionTree*

Several trees exist with the minimum total path length (minimum cost). For our case, we choose the first tree found. Due to space limitations, we cannot show the tree. It has height 6, and the total cost (path length) for all topological predicates is 170. Compared with the cost of $8 \cdot 33 = 264$ for the solution of Section 4.1, this reduces the cost for predicate determination to 64%.

4.3 Matrix Thinning

A second observation is that for predicate verification not all matrix predicates have to be evaluated. For example, for predicate 1 in Table 1 the combination that $F^\circ \cap G^\circ = \varnothing \ \wedge \ \partial F \cap \partial G = \varnothing$ holds (indicated by two 0's) is unique among the 33 predicates. Hence, only these two matrix predicates have to be tested in order to decide about *true* or *false* of the predicate. The question arises how the matrices can be "thinned out" and nevertheless remain unique among the 33 predicates. We have implemented a brute-force algorithm which for each intersection matrix checks all combinations of matrix predicate values for uniqueness among the 32 other intersection matrices. The algorithm begins with single matrix predicate values and then proceeds with pairs, triples, quadruples, quintuples, etc. Table 1 shows the result. The '*' elements correspond to "don't care" elements whereas other elements are the essential elements. We have found 6 matrices with 2 matrix predicates that have to be checked, 6 matrices with 3 matrix predicates to be checked, 10 matrices with 4 matrix predicates to be checked, and 11 matrices with 5 matrix predicates to be checked. The total cost is $6 \cdot 2 + 6 \cdot 3 + 10 \cdot 4 + 11 \cdot 5 = 125$. Compared with the cost of 8 per topological predicate for the solution of Section 4.1, this reduces the average cost for predicate verification to 3.8 $(= 47\%)$ per topological predicate. [2] uses a similar approach which is based on a greedy heuristic. In contrast to this work, we also provide an implementation concept.

5 Implementation and Performance Analysis

The aforementioned techniques have been tested and verified through an implementation of the topological predicates as part of the SPAL2D package. The implementation makes use of a complex spatial data type system (SDT) which in turn is built on top of the rational number system (RATIO). This design framework ensures system-wide numerical robustness and topological consistency. Since performance is one of the goals for this implementation, we choose a popular compiled language *C++* for the development.

As far as topological predicate implementation is concerned, the 9-intersection matrix characterization technique makes it possible to process a single matrix predicate at a time. This gives rise to the possibility of using decision tree or thin matrices depending on the query type. An empirical study has been performed to verify both correctness and performance improvement in using these evaluation methods. Figure 3 illustrates the result for predicate determination.

$$1:\begin{pmatrix}0|0 & 0|* & 1|*\\ 0|* & 0|0 & 1|*\\ 1|* & 1|* & 1|*\end{pmatrix}\quad 2:\begin{pmatrix}0|0 & 0|* & 1|*\\ 0|* & 1|* & 0|0\\ 1|* & 1|* & 1|*\end{pmatrix}\quad 3:\begin{pmatrix}0|0 & 0|* & 1|*\\ 0|* & 1|* & 1|*\\ 1|* & 0|0 & 1|*\end{pmatrix}\quad 4:\begin{pmatrix}0|0 & 0|* & 1|*\\ 0|* & 1|1 & 1|1\\ 1|* & 1|1 & 1|1\end{pmatrix}\quad 5:\begin{pmatrix}1|* & 0|* & 0|0\\ 0|* & 1|* & 0|*\\ 0|0 & 0|* & 1|*\end{pmatrix}$$

$$6:\begin{pmatrix}1|* & 0|* & 0|0\\ 0|0 & 1|* & 0|*\\ 1|1 & 1|* & 1|*\end{pmatrix}\quad 7:\begin{pmatrix}1|* & 0|* & 0|0\\ 1|* & 0|0 & 0|*\\ 1|* & 1|* & 1|*\end{pmatrix}\quad 8:\begin{pmatrix}1|* & 0|* & 0|0\\ 1|1 & 1|* & 0|*\\ 1|* & 0|0 & 1|*\end{pmatrix}\quad 9:\begin{pmatrix}1|* & 0|* & 0|0\\ 1|1 & 1|1 & 0|*\\ 1|* & 1|1 & 1|*\end{pmatrix}\quad 10:\begin{pmatrix}1|1 & 0|0 & 1|1\\ 0|0 & 1|* & 0|0\\ 1|* & 1|* & 1|*\end{pmatrix}$$

$$11:\begin{pmatrix}1|* & 0|0 & 1|1\\ 0|* & 1|* & 1|*\\ 0|0 & 0|* & 1|*\end{pmatrix}\quad 12:\begin{pmatrix}1|1 & 0|0 & 1|*\\ 0|0 & 1|* & 1|*\\ 1|1 & 0|0 & 1|*\end{pmatrix}\quad 13:\begin{pmatrix}1|1 & 0|0 & 1|*\\ 0|0 & 1|* & 1|1\\ 1|* & 1|1 & 1|*\end{pmatrix}\quad 14:\begin{pmatrix}1|1 & 0|0 & 1|1\\ 1|* & 0|0 & 1|*\\ 1|* & 1|* & 1|*\end{pmatrix}\quad 15:\begin{pmatrix}1|* & 0|0 & 1|1\\ 1|* & 1|* & 0|0\\ 1|* & 0|0 & 1|*\end{pmatrix}$$

$$16:\begin{pmatrix}1|* & 0|0 & 1|1\\ 1|1 & 1|* & 0|0\\ 1|* & 1|1 & 1|*\end{pmatrix}\quad 17:\begin{pmatrix}1|* & 0|0 & 1|*\\ 1|1 & 1|* & 1|1\\ 1|* & 0|0 & 1|*\end{pmatrix}\quad 18:\begin{pmatrix}1|* & 0|0 & 1|*\\ 1|1 & 1|1 & 1|1\\ 1|* & 1|1 & 1|*\end{pmatrix}\quad 19:\begin{pmatrix}1|* & 1|* & 1|*\\ 0|* & 0|0 & 1|*\\ 0|0 & 0|* & 1|*\end{pmatrix}\quad 20:\begin{pmatrix}1|1 & 1|* & 1|*\\ 0|0 & 0|0 & 1|*\\ 1|1 & 1|* & 1|*\end{pmatrix}$$

$$21:\begin{pmatrix}1|* & 1|1 & 1|*\\ 0|* & 1|* & 0|0\\ 0|0 & 0|* & 1|*\end{pmatrix}\quad 22:\begin{pmatrix}1|* & 1|* & 1|*\\ 0|0 & 1|* & 0|0\\ 1|1 & 0|0 & 1|*\end{pmatrix}\quad 23:\begin{pmatrix}1|* & 1|1 & 1|*\\ 0|0 & 1|* & 0|0\\ 1|* & 1|1 & 1|*\end{pmatrix}\quad 24:\begin{pmatrix}1|* & 1|1 & 1|*\\ 0|* & 1|1 & 1|1\\ 0|0 & 0|* & 1|*\end{pmatrix}\quad 25:\begin{pmatrix}1|* & 1|1 & 1|*\\ 0|0 & 1|* & 1|1\\ 1|1 & 0|0 & 1|*\end{pmatrix}$$

$$26:\begin{pmatrix}1|* & 1|1 & 1|*\\ 0|0 & 1|1 & 1|1\\ 1|* & 1|1 & 1|*\end{pmatrix}\quad 27:\begin{pmatrix}1|* & 1|1 & 1|*\\ 1|* & 0|0 & 0|0\\ 1|* & 1|* & 1|*\end{pmatrix}\quad 28:\begin{pmatrix}1|* & 1|* & 1|*\\ 1|1 & 0|0 & 1|*\\ 1|* & 0|0 & 1|*\end{pmatrix}\quad 29:\begin{pmatrix}1|* & 1|1 & 1|*\\ 1|1 & 0|0 & 1|1\\ 1|* & 1|1 & 1|*\end{pmatrix}\quad 30:\begin{pmatrix}1|* & 1|1 & 1|*\\ 1|1 & 1|* & 0|1\\ 1|* & 0|0 & 1|*\end{pmatrix}$$

$$31:\begin{pmatrix}1|* & 1|1 & 1|*\\ 1|1 & 1|1 & 0|0\\ 1|* & 1|1 & 1|*\end{pmatrix}\quad 32:\begin{pmatrix}1|* & 1|1 & 1|*\\ 1|1 & 1|1 & 1|1\\ 1|* & 0|0 & 1|*\end{pmatrix}\quad 33:\begin{pmatrix}1|* & 1|1 & 1|*\\ 1|1 & 1|1 & 1|1\\ 1|* & 1|1 & 1|*\end{pmatrix}$$

Table 1. Topological matrices | thinning matrices for the 33 topological predicates between two complex regions.

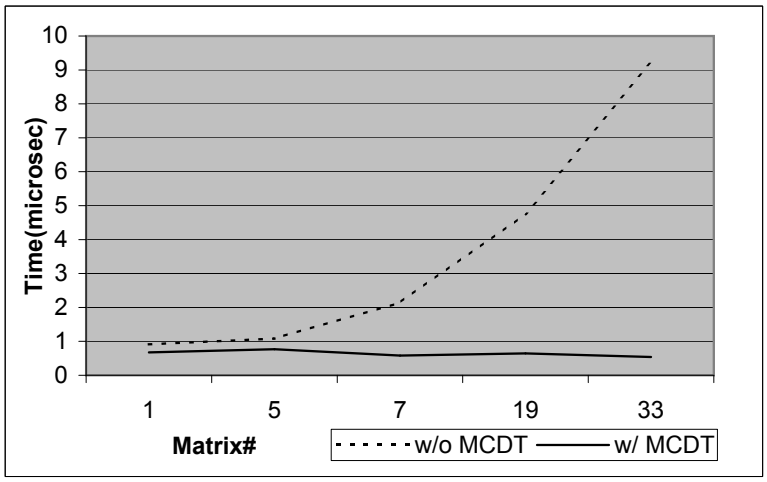

Fig. 3. Predicate Determination Comparison

Without using the decision tree, one would have to perform a linear search through all 33 predicates. Logarithmic search is not possible since there is no comparison relation between these predicates. Hence, the time function is a monotonely increasing function with the best case requiring 8 matrix predicate comparisons and worst case requiring upto $8 \cdot 33 = 264$ comparisons. In contrast, by using the decision tree, each predicate determination is limited to at most 6 matrix predicate comparisons. Our test cases are designed to cover all predicates for complex regions. Using

a 1.8 GHz 64-bit processor, the average predicate evaluation time with decision tree is 0.6416 micro seconds as opposed to 3.6228 micro seconds without decision tree. This reduces the cost to 18% which indicates an improvement of 82%.

For predicate verification, evaluation without thin matrices requires 8 matrix predicate comparisons, whereas at most 5 comparisons are required with thin matrices. Furthermore, by using thin matrices, we can reject evaluation as soon as there is a mismatch of matrix predicates. For our test cases, the verification requires an average of 3.24 matrix predicate comparisons. This reduces the cost to 40.5% which indicates an improvement of 59.5%.

6 Conclusions and Future Work

This paper presents research results on the evaluation and implementation of topological predicates on complex regions. It considers the two main problems of topological predicate verification and determination. The main idea is to characterize each matrix predicate of the 9-intersection matrix by a unique set of segment classes that have to be checked. This characterization allows the use of minimum cost decision tree and matrix thinning, which significantly speed up the predicate determination and verification processes. The approach has been implemented in the SPAL2D software library which is currently under development and determined for an integration into extensible databases. In the future, we plan to consider the evaluation of topological predicates for all type combinations that also include the spatial data types *point* and *lines*.

References

1. J. F. Allen. Maintaining Knowledge about Temporal Intervals. *Communications of the ACM*, 26:832–843, 1983.
2. E. Clementini, J. Sharma, and M.J. Egenhofer. Modeling Topological Spatial Relations: Strategies for Query Processing. 18(6):815–822, 1994.
3. Z. Cui, A. G. Cohn, and D. A. Randell. Qualitative and Topological Relationships. *3rd*, LNCS 692, pp. 296–315, 1993.
4. M. J. Egenhofer and R. D. Franzosa. Point-Set Topological Spatial Relations. 5(2):161–174, 1991.
5. R. H. Güting and M. Schneider. Realm-Based Spatial Data Types: The ROSE Algebra. 4:100–143, 1995.
6. R.H. Güting, T. de Ridder, and M. Schneider. Implementation of the ROSE Algebra: Efficient Algorithms for Realm-Based Spatial Data Types. pp. 216–239, 1995.
7. M. Schneider. *Spatial Data Types for Database Systems - Finite Resolution Geometry for Geographic Information Systems*, volume LNCS 1288. Springer-Verlag, Berlin Heidelberg, 1997.
8. M. Schneider. Implementing Topological Predicates for Complex Regions. pp. 313–328, 2002.
9. M. Schneider and T. Behr. Topological Relationships between Complex Spatial Objects. *ACM Transactions on Database Systems*, 2006. Accepted for publication.