

# Designing the MultiMap Library – Step 1

## Representation and Manipulation of Plane Vector Maps

Raquel Viaña Fernández, Enrico Puppo  
Department of Computer and Information Sciences (DISI),  
University of Genova, Via Dodecaneso 35, 16146 Genova - Italy

### Technical Report PDISI-01-14

#### Abstract

The purpose of the MultiMap library is to provide a data structure and a set operators to support the representation and manipulation in primary memory of 2D geographic maps in vector format and at multiple resolution.

This report contains fundamentals and specifications concerning the design of the first layer of the library, which consists of:

- A data structure to represent 2D geographic maps (plane maps) in vector format at a single resolution;
- A set of operators that permit to traverse, build and modify such a data structure;
- A set of spatial queries that can be implemented on top of such operators.

## 1 FUNDAMENTALS

### 1.1 Plane Maps

Several models have been proposed in the literature to formalize 2D geographic maps and operations acting on them. When trying to give a formal definition of a map there is no general agreement. The two main classes of models considered in the literature are: *raster models*, in which a map is represented by a regular subdivision of a domain into (small) regions called *raster elements*, or *pixels*, and each such region is treated as an atomic entity carrying synthetic information in the form of attributes; and *vector models*, in which a map is represented as an aggregate of geometric entities (points, lines and regions) of arbitrary shape, each entity being characterized by its own geometry and attributes. We will consider a model of the latter class in a quite general setting, defined as in [5, 18].

We consider spatial entities of three classes, namely points, lines, and regions embedded in the Euclidean plane (i.e., each entity corresponds to a 0-, 1- or 2-dimensional subset of  $\mathbb{R}^2$ ). A point is uniquely defined by its coordinates in a coordinate system of  $\mathbb{R}^2$ . With the usual topology, the border of a point is empty, while its interior (relative interior) coincides with the point itself. A (simple) open line  $l$  is a subset of  $\mathbb{R}^2$  that is homeomorphic to the standard open unit interval  $(0,1)$  on the real line. The (relative) interior of an open line is the line itself. By extending the homeomorphism by continuity to the closed interval  $[0,1]$ , we define  $l(0)$  and  $l(1)$  to be the endpoints of a line  $l$ . Such points define the border of  $l$ . The line keeps on being simple if it is injective on  $[0,1]$  or on both  $[0,1)$  and  $(0,1]$ . If  $l(0) = l(1)$ , line  $l$  is said to be closed. A closed line plus its endpoint is also called a loop. A (simple) chain is a sequence of lines  $c = (l_0, \dots, l_{k-1})$  such that  $\forall i = 1, \dots, k-1, l_{i-1}(1) = l_i(0)$  and  $\cup_{i=0}^{k-1} l_i$  is a (simple) line. A chain is said open or closed if it is open or closed as a line, respectively. With abuse of notation, points  $l_0(0)$  and  $l_{k-1}(1)$  are called endpoints of  $c$ ; all other endpoints of lines forming  $c$  are called joints of  $c$ . A region is characterised by a simple closed line (or chain)  $l_0$ , plus possibly a set  $\{l_1, \dots, l_k\}$  of simple closed lines (or chains), such that: (i) no two such lines intersect; (ii) for  $i = 1, \dots, k$  line  $l_i$  is contained in  $int(l_0)$  and in  $ext(l_j), \forall 0 \neq j \neq i$ . Region  $r$  is the subset of  $\mathbb{R}^2$  defined as  $\cap_{i=1}^k ext(l_i) \cap int(l_0)$ . Line  $l_0$  is called the outer border, lines  $l_1, \dots, l_k$  are called the inner

borders; the union, as point sets, of lines  $l_0, l_1, \dots, l_k$  defines the Euclidean border or  $r$  (i.e., the border of  $r$  regarded as a subset of  $\mathbb{R}^2$  with the standard topology).

A covering of a domain  $D$  of  $\mathbb{R}^2$  with a collection of spatial entities (a set of points  $P$ , a set of lines  $L$ , and a set of regions  $R$ ) is a *weakly disjoint covering* if and only if the interior of no region intersects the interior of a different region; the interior of no line intersects the interior of a different line; no point coincides with a different point, or intersects the relative interior of a line. A *plane map*  $M = (P, L, R)$  is defined as a weakly disjoint covering such that  $P \cup L \cup R$  is a regular set and regions in  $R$  correspond to interior of pointsets obtained by regularizing all maximal portions of the plane such that any two points in their interior can be connected by a line that does not intersect any line of  $L$ . The definition of a region in a map  $M$  also includes the unbounded portion of the plane surrounding the domain  $D$  covered by all other entities of  $M$ . Such region, called the infinite region (denoted  $r_\infty$ ) is such that  $r_\infty = \mathbb{R}^2 - D$ . It follows from the definition that a map  $M$  can be represented as a triple  $(P, L, R)$ , where sets  $P$ ,  $L$  and  $R$  represent the points, lines and regions of  $M$  respectively.

The notions of combinatorial boundary and star of the entities in the map completely define the topological structure of the map. The *combinatorial boundary* of a region  $r$  is the collection of points and lines of  $M$  that form the topological border of  $r$  plus the point and lineal features. The *regularization*  $r^*$  is the closure of the interior of  $r$ . The elements of  $(P, L)$  belonging to the interior of  $r^*$  are called the *features* of  $r$ , while those belonging to the topological border of  $r^*$  form the *proper boundary* of  $r$ . The combinatorial boundary of  $r$  is formed by the union of its features and of its proper boundary. The combinatorial boundary of a line is formed by its endpoints, and the combinatorial boundary of a point is the empty set. The *star*, or *combinatorial co-boundary* of an entity is formed by the entity itself plus the set of all entities containing it in their topological boundaries.

## 1.2 Topological relations

The practical need of answering queries related to spatial concepts in a GIS leads to the study of spatial relations between the entities composing a map. Such relations are usually classified into *topological relations*, relations invariant under homeomorphism [8], *metric relations*, involving spatial proximity [5], and *order relations*, concerning the partial and total order of spatial objects [13]. In this report we will only deal with topological relations.

Topological spatial relationships between two geometric objects embedded in the real plane have been widely studied. The *Calculus Based Model* or *CBM* [2] introduces five relationships that can be applied to “simple” points, lines and areas (area entities are connected areas with no holes, line entities have no self-intersections and are either circular or with only two endpoints; and points entities may contain only one point). Two boundary operators are also defined to extract line and region boundaries. If we denote as  $a$  and  $b$  two simple entities topologically closed, the *CBM* relationships are:

- $a.disjoint(b) \Leftrightarrow a \cap b = \emptyset$ . It applies to every group of relations.
- $a.touches(b) \Leftrightarrow (I(a) \cap I(b) = \emptyset) \wedge (a \cap b \neq \emptyset)$ . It applies to area/area, line/line, line/area, point/area, point/line groups of relationships, but not to the point/point group.
- $a.crosses(b) \Leftrightarrow (dim(I(a) \cap I(b)) < max(dim(I(a)), dim(I(b)))) \wedge (a \cap b \neq a) \wedge (a \cap b \neq b)$ . It applies to line/line and line/area groups.
- $a.within(b) \Leftrightarrow (a \cap b = a) \wedge (I(a) \cap I(b) \neq \emptyset)$ . It applies to every group.
- $a.overlaps(b) \Leftrightarrow (dim(I(a)) = dim(I(b)) = dim(I(a) \cap I(b))) \wedge (a \cap b \neq a) \wedge (a \cap b \neq b)$ . It applies to area/area and line/line groups.

Such relations have also been extended to deal with complex regions -having both separations and holes-, complex lines -having separations-, and complex points -sets of points- [3].

The *4-intersections method* (*4IM* for short) [10] considers the boundary and interior of two homogeneous spatial objects of dimension 2 embedded in the plane. The topological invariant considered is the empty or non-empty intersection. With the resulting matrix we can say if two regions are related according to one of the following relations: *disjoint*, *meet*, *contains*, *inside*, *covers*, *coveredBy*, *equals* or *overlaps*. Considering

any pair of entities, also relations *bounds and boundedBy* are possible. In general, we have 16 combinations for each binary relation *PP, PL, PR, LL, LR, RR*. Some of them, however, make no geometrical sense. Other topological invariants, like the dimension of the non-empty intersection and the number of connected segments are able to distinguish more situations. The *Dimension Extended Method* or *DEM* [9, 4] is an extension of the *4IM*, and takes into account the dimension of the intersections. The *9-intersections method (9I)* [11] characterizes the topological relation between two sets through the intersection set of interiors, boundaries, and exterior of both sets. Considering as topological invariant the empty or non-empty intersection of such sets, 512 binary topological relations are possible. For two simple lines (non-branching, non self-intersections) embedded in the real plane, 33 different topological relations can be distinguished with the *9I* method, and for a line and a region, 19 different situations.

The *CBM* is more expressive than point-set methods and a combination of the *DEM* and the *9IM* must be produced to find an equivalent point-set method which have the same expressive power of the *CBM*. All the cases considered by the *CBM* can be represented in the *DE+9IM*. The previously defined *CBM* relations can be expressed in terms of the *DE-9IM* [2].

The OpenGIS 1.1 specification [16] considers the relations defined by the *CBM* and theirs equivalent *DE+9IM* for complex entities.

### 1.2.1 Topological relations in a map

In a given map, due to the constrained spatial relations allowed between entities, and to the consideration of topologically open entities, only the following situations, according to the *CBM* model, are possible:

- *Region-Region*. Two distinct regions are always *disjoint*. They might share a common boundary.
- *Line-Region*. Given a line  $l$  and a region  $r$ , either  $l$  and  $r$  are *disjoint*, or  $l$  is *within*  $r$ . Line  $l$  might bound  $r$ .
- *Point-Region*. Given a point  $p$  and a region  $r$ , it can only arise that  $p$  and  $r$  are *disjoint*, or  $p$  *within*  $r$ . Point  $p$  might bound  $r$ .
- *Line-Line*. Given two distinct lines, they are *disjoint*. They might share a common endpoint.
- *Line-Point*. Given a point  $p$  and a line  $l$ , they are *disjoint*. Point  $p$  might bound  $l$ .
- *Point-Point*. Two distinct points are always *disjoint*.

### 1.2.2 Combinatorial and set-theoretic relations

In [5] topological relations are subclassified into two classes, topological -that we will rename *combinatorial relations* for not inducing any misunderstanding- and *set-theoretic relations*, characterized by different geometric concepts that reflect the different computational problems arising when answering the corresponding queries. While combinatorial relations only make sense when both entities belong to the same map, set-theoretic relations might take place between entities belonging to different maps.

- *Combinatorial relations* encode the combinatorial structure of a map, and are further broken down into adjacency, boundary, and co-boundary relations. These relations will be useful for answering queries involving computing the boundary or co-boundary of objects, or queries involving navigating the map through adjacency.

1. *Adjacency relations*. They are applied to pairs of entities belonging to the same class:

$$\begin{aligned}
 a, b \in P, a \stackrel{PP}{\sim} b &\Leftrightarrow \text{There exists a line bounded by } a \text{ and } b \\
 a, b \in L, a \stackrel{LL}{\sim} b &\Leftrightarrow a \text{ and } b \text{ share a common endpoint} \\
 a, b \in R, a \stackrel{RR}{\sim} b &\Leftrightarrow a \text{ and } b \text{ share some common boundary entity}
 \end{aligned}$$

2. *Boundary relations.* They apply to pairs of entities belonging to the classes  $LP, RP, RL$ :

$$a \in \{L, R\}, b \in \{P, L\} \text{ s.t. } \dim(a) > \dim(b), a \stackrel{LP, RP, RL}{\sim} b \Leftrightarrow b \text{ bounds } a$$

3. *Co-boundary relations.* They apply to pairs of entities belonging to the classes  $PL, LR, PR$ :

$$a \in \{P, L\}, b \in \{L, R\} \text{ s.t. } \dim(a) < \dim(b), a \stackrel{PL, LR, PR}{\sim} b \Leftrightarrow b \text{ is bounded by } a$$

- *Set-theoretic relations* encode spatial interference between entities, even in different maps. Points are considered the basic elements, and lines and regions are seen as infinite and continuous sets of points. They are subdivided into coincidence, element-of, containment and intersection relations. To compare spatial objects through set-theoretic operators they do not need be part of a given map. Hence, the following relations will be defined in general on the three spatial classes, instead of on the subsets of such classes composing a map.

1. *Coincidence relations.* They are unordered relations applied to pairs of entities belonging to the same class:

$$\equiv: (a, b) \in \{PP, LL, RR\}, a \stackrel{\equiv}{\sim} b \Leftrightarrow a \equiv b$$

2. *Element-of relations.* They are ordered relations applied to pairs of entities belonging to the PL, PR, LP or RP classes:

$$\begin{aligned} \in: a \in P, b \in \{L, R\}, a \stackrel{\in}{\sim} b &\Leftrightarrow a \in b \\ \ni: a \in \{L, R\}, b \in P, a \stackrel{\ni}{\sim} b &\Leftrightarrow b \in a \end{aligned}$$

3. *Containment relations.* They are ordered relations applied to pairs of entities belonging to the LL, LR, RL or RR classes:

$$\begin{aligned} \subset: (a \in L, b \in \{L, R\}) \cup (a \in R, b \in R), a \stackrel{\subset}{\sim} b &\Leftrightarrow a \subset b \\ \supset: (a \in \{R, L\}, b \in L) \cup (a \in R, b \in R), a \stackrel{\supset}{\sim} b &\Leftrightarrow b \subset a \end{aligned}$$

4. *Intersection relations.* They are unordered relations applied to pairs of entities belonging to the LL, LR, RL or RR classes:

$$\cap: a, b \in \{L, R\}, a \stackrel{\cap}{\sim} b \Leftrightarrow a \cap b \neq \emptyset$$

Let's see that this model is fully compatible with the *CBM* model. Let  $p, p_1, p_2$  be points,  $l, l_1, l_2$  be open lines, and  $r, r_1, r_2$  be regions belonging to some map (not necessarily the same map).

- *disjoint* relation can be applied to every pair of entities

$$\begin{aligned} p_1.\text{disjoint}(p_2) &\Leftrightarrow \neg(p_1 \text{ coincides with } p_2) \\ p.\text{disjoint}(c) &\Leftrightarrow \neg(p \text{ is element of } c), \text{ with } c \in \{L, R\} \\ c_1.\text{disjoint}(c_2) &\Leftrightarrow \neg(c_1 \text{ intersects } c_2), \text{ with } c_1, c_2 \in \{L, R\} \end{aligned}$$

- *touches* relation cannot arise between two entities belonging to the same or different maps because entities in maps are topologically open
- *crosses* relation applies to two entities belonging to different maps

$$\begin{aligned} l_1.\text{crosses}(l_2) &\Leftrightarrow (l_1 \text{ intersects } l_2) \wedge \neg(l_1(l_2) \text{ contained in } l_2(l_1)) \wedge \neg(l_1 \text{ coincides with } l_2) \\ l.\text{crosses}(r) &\Leftrightarrow (l \text{ intersects } r) \wedge \neg(l \text{ contained in } r) \end{aligned}$$

- *within*

$p_1.within(p_2) \Leftrightarrow p_1 \text{ coincides with } p_2$ ,  $p_1$  and  $p_2$  belonging to the same or different maps  
 $p.within(c) \Leftrightarrow p \text{ is element of } c$ , with  $c \in \{L, R\}$ ,  $p$  and  $r$  in any situation,  $p$  and  $l$  in different maps  
 $l.within(c) \Leftrightarrow c \text{ contains } l$ , with  $c \in \{L, R\}$ ,  $l$  and  $r$  in any situation,  $l_1$  and  $l_2$  in different maps  
 $r_1.within(r_2) \Leftrightarrow r_2 \text{ contains } r_1$ ,  $r_1$  and  $r_2$  in different maps

- *overlaps* relation applies to entities belonging to different maps

$c_1.overlaps(c_2) \Leftrightarrow (c_1 \text{ intersects } c_2) \wedge \neg(c_1 \text{ crosses } c_2) \wedge \neg(c_1 \text{ within } c_2)$ , with  $c_1, c_2 \in \{l, r\}$

### 1.3 Separating topology from geometry and semantics

A map and the entities composing it can be characterized in terms of three distinct kinds of information:

- **Geometry**, which concerns the shape, extension and location of entities in space;
- **Topology**, which concerns the mutual positions of different entities in a map, hence their spatial relations;
- **Semantics**, which concern the meaning of entities in a map, thus heterogeneous information on objects they represent in the real world.

Geometry and semantics can be modeled at the level of single entities, while topology involves relations among different entities. For this reason, geometry and semantics can be represented as *attributes* attached to single entities, while topology needs to be encoded in more complex structures. In the following, we just outline basic concepts for modeling geometry and semantics, and we study in more detail how to model the topology of a map. Once a topological model has been obtained, geometry and semantics can be added to it in a modular way.

#### 1.3.1 Geometry

The geometry of a point in a map is completely defined by its coordinates in a 2D Euclidean reference frame.

The geometry of a line can be represented mathematically in various ways, e.g., either as a spline curve, or simply as a polyline. In the following, we will adopt a representation based on polylines. However, we will follow a modular approach in designing our data structures and algorithms, which allows for later extensions with different representations, such as splines.

The geometry of a region is induced by its proper boundary, since the region is intended as the portion of space that lies inside its external loop and outside all its internal loops.

Thus, in order to encode the geometry of a plane map  $M = (P, L, R)$ , it is sufficient to provide: a list of coordinates for all points of  $P$ ; and mathematical representations for all lines of  $L$ .

#### 1.3.2 Semantics

Entities in a map may have very different kinds of semantics, depending on the kind of map and on the specific objects represented in it.

The simplest and most common kind of map in GIS is the *choropleth map* which simply gives a partition of different areas (or more generally, different entities) into a finite set of classes. In this case, semantics for entities can be obtained by simply assigning each entity an attribute in a finite enumeration type.

More complex maps can extend attributes to be qualitative, descriptive, etc., and an arbitrarily high level of richness and complexity can be achieved in describing a single entity, depending on the representation needs in the context of a given map.

For our concern, the only important thing is that a description of semantics can be given in *tabular form* associated to each single entity in the map.

Further semantics can concern groups of entities that have a given attribute in common and, e.g., form a connected set in space (an example is a sequence of lines that form a given road or river). However, grouping information do not need to be stored explicitly in a map model, since they can be inferred on the basis of basic semantics and topology.

### 1.3.3 Topology

From now on, we will regard only the topological aspects of maps, by considering geometric aspects as attributes attached to points and lines.

As we have already said, the combinatorial boundary and star of the entities in the map completely define its topological structure. Therefore, when disregarding geometry, maps can be modeled as pure combinatorial entities in terms of *Abstract Cell Complexes (ACC)*, taking advantage of the properties of this well known topological structure [14, 15].

Let  $C$  be a finite set, called the *set of cells*. An *abstract cell complex (ACC)*  $\Gamma = (C, \preceq, \dim)$  with cells in  $C$  is defined as follows:

- $\preceq$  is a partial ordering on the elements of  $C$  (i.e.,  $\preceq$  is a reflexive, antisymmetric, and transitive binary relation) called the *bounding relation*;
- $\dim : C \rightarrow \mathbb{N}$ , called the *dimension function*, is such that

$$\gamma' \preceq \gamma'' \wedge \gamma' \neq \gamma'' \Rightarrow \dim(\gamma') < \dim(\gamma'').$$

The *boundary* of a cell  $\gamma$  of  $\Gamma$  is defined as  $\partial\gamma = \{\xi \in C \mid \xi \preceq \gamma\} \setminus \{\gamma\}$ .

The *star* of a cell  $\gamma$  of  $\Gamma$  is defined as  $*\gamma = \{\xi \in C \mid \gamma \preceq \xi\}$ .

A map  $M = (P, L, R)$  can be effectively described as an ACC as follows:

- Define the set of atomic cells  $C = P \cup L \cup R$ ;
- Define the following bounding relation on  $C$ :  $x \preceq y \Leftrightarrow x \subseteq \bar{y}$ , where the symbol  $\subseteq$  denotes containment between sets in  $\mathbb{R}^2$ .
- Define the dimension function *dim* as follows:

$$\dim(x) = \begin{cases} 0 & \text{if } x \in P \\ 1 & \text{if } x \in L \\ 2 & \text{if } x \in R \end{cases}$$

The triple  $\Gamma = (C, \preceq, \dim)$  is an abstract cell complex. Moreover, the concepts of boundary and star in a map and in its corresponding ACC coincide. Thus, the ACC  $\Gamma$  gives a complete description of the *structure* of map  $M$ , while its geometry and semantics are given by adding attributes to cells of an ACC.

## 2 A data structure for maps

The definition of map can be naturally formulated in terms of partitions of a plane domain. We allow such partitions to be induced from any planar graph with piecewise linear edges, and possibly containing isolated points and/or dangling chains, i.e., a *Planar Euclidean Graph (PEG)*. We represent our map  $M = (P, L, R)$  through a PEG  $G = (V, E, F)$ , where:

- $V$ , the set of vertices, is a set of points in the plane;
- $E$ , the set of edges, is a set of polygonal chains whose endpoints belong to  $V$  and such that two edges of  $E$  never cross;
- $F$ , the set of faces, is a set of maximal regions  $f$  bounded by chains of edges in  $E$ , such that for every two points in the interior of  $f$  there is a curve in the plane that joins them without intersecting any edge of  $E$ .

In section 4, we will study algorithms for solving interference queries that manipulate PEGs. As such algorithms always deal with straight-line segments, and regions bounded by such kind of lines, we introduce here the *expanded graph* of a PEG. The idea is simply to divide each edge (polygonal chain) of the PEG into the straight-line segments forming it, and to add these segments to the set of edges, and their endpoints (corresponding to joints) to the set of vertices. The faces of a PEG and of its expanded graph are coincident.

The data structure chosen to efficiently implement the *PEG* is obtained as an extension of the well known *DCEL* data structure [17], widely used to encode simple plane subdivisions. It is an edge-oriented data structure, so it codifies explicitly relations between every edge and its adjacent entities (edges with a common endpoint), boundary (endpoints) or star (faces containing the edge).

Topological relations associated to vertex, edges and faces of a generalized plane subdivision are:

- (a) Relations associated to a vertex  $v$ 
  - *vertex – vertex*: represents the set of vertex  $w$  such that the edges of endpoints  $v$  and  $w$  belong to the subdivision
  - *vertex – edge*: represents the set of edges incident in  $v$ , i.e., the set of edges that have an endpoint equal to  $v$
  - *vertex – face*: represents the set of faces for which  $v$  is a vertex of its proper boundary, or a point feature or an endpoint of a linear feature
- (b) Relations associated to an edge  $e$ 
  - *edge – vertex*: represents the endpoints of  $e$
  - *edge – edge*: represents the four edges incident to the endpoints of  $e$  that are first found when, from one point interior to  $e$ , one rotates clockwise and counterclockwise
  - *edge – face*: represents the two faces belonging to the subdivision for which  $e$  is an edge of the proper boundary, or the face that contains it in case the edge is a linear feature
- (c) Relations associated to a face  $f$ 
  - *face – vertex*: represents the set of vertices belonging to the proper boundary of  $f$
  - *face – edge*: represents the set of edges belonging to the proper boundary of  $f$
  - *face – face*: represents the set of faces adjacent to  $f$
  - *f – face – vertex*: represents the set of point features of  $f$
  - *f – face – edge*: represents the set of lineal features of  $f$

The data structure encodes the entities composing a PEG, together with their geometry and attributes, and a subset of the topological relations listed above. The remaining relations can be computed efficiently through simple algorithms working on such data structure. A specification of the data structure follows:

- For each vertex:
  - its two coordinates;
  - if the vertex is isolated, a pointer to the face containing it, otherwise, a pointer to one of the edges incident into it;
- For each edge:
  - a pointer to its geometry (a chain of joints);
  - two pointers to its endpoints (vertices);
  - two pointers to its incident faces (in case the edge is a lineal feature, the two faces will be coincident);
  - two pointers to the first adjacent edges met by rotating counterclockwise about its endpoints.

- For each face:
  - a pointer to a list of edges, containing one edge for each connected component of its boundary;
  - a pointer to a list of edges, containing one edge for each connected component of its contained edges (lineal features);
  - a pointer to a list of isolated points contained in the face.

The list of vertices can be maintained sorted (e.g., lexicographically), and stored into a balanced tree; also, similar lists of edges and vertices could be maintained.

This data structure is intended for main memory; disk storage, instead, requires partitioning information among different disk pages. It is not immediately clear, and it is not our subject here, how to distribute entities such that links between them can be retrieved efficiently.

### 3 Primitive operations

To create the graph representing the map through the data structure previously explained as well as to answer queries and to manipulate maps through combinatorial consistent transformations [1] the following primitive operations have been developed.

**A** Operators to build the topological structure. These operators support the creation of a map by adding elements to a simpler one.

1. *bool add\_infinite\_face* (*face f*) : inserts the infinite face in the subdivision and assigns it to *f*.
2. *bool add\_point\_feature* (*face f; vertex v*) : inserts vertex *v* among the ones associated to the *f – face – vertex* relation of the face indicated by *f*.
3. *bool add\_line\_feature* (*vertex v1, v2; edge e, e1, e2*) : inserts in the subdivision the new oriented edge whose endpoints are vertices *v1* and *v2* and assigns it to *e*. The edge is inserted so that it results associated to the *edge – edge* relation of *e1* applied in *v1* and of *e2* applied in *v2*.
4. *bool expand\_point\_to\_line* (*vertex v, v1, v2; edge e, e1, e2*) : expands a vertex to an edge; *v* represents the vertex to expand. It will be removed from the subdivision and replaced by two new vertices *v1* and *v2*. The edge is inserted so that it results associated to the *edge – edge* relation of *e1* applied in *v1* and of *e2* applied in *v2*. Every edge incident in *v* and placed (clockwise) among *e1* (included) and *e2* (included) will have as new vertex *v1*, the others *v2*.
5. *bool expand\_point\_to\_region* (*vertex v; edge e, e1; face f, f1*) : expands vertex *v* in an oriented loop counterclockwise *f* of boundary edge *e*; *e1* is the edge that determines the loop position. The loop is inserted in the section limited by *e1* and by the edge obtained of the *edge – edge* relation of *e1* applied in vertex *v*. Face *f1* is the face the loop is inserted in.
6. *bool expand\_line\_to\_region* (*edge e, e2; face f*) : expands edge *e* to a region. To do this, it creates the new edge *e2*, that has the same vertices and the same orientation as *e*. Then the new region *f* of proper boundary edges *e* and *e2* is created so that it is to the left of oriented edge *e*.
7. *bool split\_line* (*edge e0, e; vertex v0*) : splits in two edge *e0* by creating the new edge *e*. Vertex *v0* shows the point where the edge must be split. If *e0* represents the oriented edge (*v1, v2*), two new edges (*v1, v0*) and (*v0, v2*) are created.
8. *bool split\_region* (*face f0, f; vertex v1, v2; edge e*) : splits in two the region indicated by *f* joining the two vertex indicated by *v1* and *v2* with edge *e*. The oriented edge from *v1* to *v2* is assigned to *e*, the left region to *f0* and the right region to *f*.

**B** Operators to perform combinatorial consistent transformations.

1. *bool point\_abstraction* (*vertex v*) : removes vertex *v* from the subdivision
2. *bool line\_abstraction* (*edge e*) : removes edge *e* of the subdivision.
3. *bool line\_to\_point\_contraction* (*edge e; vertex v*) : contracts edge *e* to vertex *v*.

4. **bool region\_to\_point\_contraction** (*face f*) : contracts the loop indicated by *f* in the only vertex that *f* has in its boundary.
5. **bool region\_to\_line\_contraction** (*face f; edge e*) : contracts the region indicated by *f* in edge *e*.
6. **bool line\_merge** (*vertex v, edge e*) : merges the two edges incident in vertex *v*; *e* is assigned to the edge representing the merging.
7. **bool region\_merge** (*edge e, face f*) : merges the two regions associated to the *edge – face* relation of *e* and assigns the obtained region to *f*. The merging is obtained removing *e* from the outer boundary of the two regions and reassigning as edge feature of the new region the eventually other edges in common to the two original regions.

### C Functions for calculating total topological relations

1. **void vertex\_vertex** (*vertex v; list of vertices lv*) : assigns to *lv* a list of vertices associated to the *vertex – vertex* total relation of vertex *v*.
2. **void vertex\_edge** (*vertex v; list of edges le*) : assigns to *le* a list of the edges associated to the *vertex – edge* total relation of vertex *v*.
3. **void vertex\_face** (*vertex v; list of faces lf*) : assigns to *lf* a list of faces associated to the *vertex – face* total relation of vertex *v*.
4. **void edge\_vertex** (*edge e; vertex v0, v1*) : assigns to *v0* and *v1* the vertices associated to the *edge – vertex* total relation of edge *e*.
5. **void edge\_edge** (*edge e, e00, e01, e10, e11*) : assigns to *e00, e01, e10, e11* the edges associated to the *edge – edge* total relation of edge *e*.
6. **void edge\_face** (*edge e; face f0, f1*) : assigns to *f0* and *f1* the faces associated to the *edge – face* total relation of edge *e*.
7. **void face\_vertex** (*face f; list of lists of vertices llv*) : *llv* represents a list of lists of vertices and is assigned with the list of the vertices associated to the *face – vertex* total relation of face *f* (one for the outer boundary and one for each inner boundary).
8. **void face\_edge** (*face f; list of lists of edges lle*) : *lle* represents a list of lists of edges and is assigned with the list of edges associated to the *face – edge* total relation of face *f* (one for the outer boundary and one for each inner boundary).
9. **void face\_face** (*face f; list of lists of faces llf*) : *llf* represents a list of lists of faces and is assigned with the list of faces associated to the *face – face* total relation of face *f* (one for the outer boundary and one for each inner boundary).
10. **void f\_face\_vertex** (*face f; list of vertices lv*) : assigns to *lv* the list of vertices associated to the *f – face – vertex* total relation of face *f*.
11. **void f\_face\_edge** (*face f; list of edges lle*) : *le* represents a list of lists of edges and is assigned with the list of edges associated to the *f – face – edge* total relation of face *f* (one list for each connected component associated to the correspondent partial relation).

## 4 Interference queries

A generic query asks for all entities in a map that are in a certain relation with a given entity [5]. We are now interested in the situation arising when the query entity does not belong to the query map, as in any other case the model encoding our map implemented through the data structure described in 2 allows efficient retrieval of the entities forming the answer of such query. We will refer to queries induced by such set-theoretic relations, when the query object does not belong to the query map, as *interference queries*. Suitable algorithms must be found that are able to retrieve the entities in a map that spatially interfere with a given entity foreign to the map, once such entity is ‘plunged’ into the query map. Computational Geometry offers algorithms to search efficiently planar subdivisions, and to intersect sets of segments. All interference queries can be answered through such techniques. However, CG techniques require huge preprocessing time, a cost

one cannot afford when dealing with a map of big size. We will try to avoid additional storage, preprocessing time and the superimposition of any spatial index by exploiting the topological structure encoding the query map, at the cost of higher query answer time. The establishment of a tradeoff between naive methods, that do not require any additional structure at a cost of high search time, and the computational geometry elaborated methods which need additional storage and preprocessing costs, give rise to the so called *walking methods*, which do neither require preprocessing nor additional storage and improve the naives one, although a complexity study of such methods does not actually exist.

Interference queries induced by element-of, containment and intersection relations can be all answered efficiently by solving the point location, line intersection and region intersection problems. An additional procedure to walk along a given direction on the map must also be available.

#### 4.1 Walking along a line on an expanded PEG

We will study every possible situation arising when walking on an Expanded Plane Euclidean Graph  $G^*$  along a line. Given a point  $A$  lying on some element of  $G^*$ , we must be able to find the element of  $G^*$  first found when  $A$  suffers an infinitesimal displacement along a given direction. If  $A$  is inside a face, whatever direction we take we will advance through the same face (Figure 1.a). In case  $A$  is placed at the interior of an edge it might continue walking along that edge (Figure 1.b.1) or advancing through one of the faces incident to the edge (Figure 1.b.2). If the edge is a feature (Figure 1.b.3) there will be only one bounding face. And finally, if  $A$  is placed at a vertex it might continue along one of the edges incident to it (Figure 1.c.1) or through one of the faces incident to it (Figure 1.c.2). If it belongs to a feature there will be only one such faces (Figure 1.c.3). If the vertex is isolated (Figure 1.c.4) we are in the first considered case,  $A$  inside a face. Let's see a specification of this simple auxiliary procedure called `FIND_NEXT_TRAVERSED_ENTITY`.

```
FIND_NEXT_TRAVERSED_ENTITY (point B, face in_FACE, edge in_EDGE, vertex in_VERTEX;
face out_FACE, edge out_EDGE);
/* We are placed at a point A, different from B, located inside face in_FACE, or at the interior of edge
in_EDGE, or at vertex in_VERTEX. These three possibilities are mutually excluding, so exactly two of these
three input parameters are NULL. Point B is such that oriented segment AB defines the direction to follow.
The procedure finds face or edge of  $G^*$  first cut by segment AB */
1. if A lies inside face in_FACE //in_FACE != NULL //1.a
2.   then report that face // out_FACE := in_FACE
3.   else if A is placed at the interior of edge in_EDGE//in_EDGE != NULL
4.     then if AB continues along in_EDGE //1.b.1
5.       then report in_EDGE //out_EDGE := in_EDGE
6.       else if in_EDGE is a feature edge //1.b.2
7.         then report the face bounding in_EDGE
8.         else if AB continues to the right side of in_EDGE//1.b.3
9.           then report the face to the right of in_EDGE
10.          else report the face to the left of in_EDGE
11.   else /* A is placed at vertex in_VERTEX != NULL*/
12.     ENTITY_INCIDENT_VERTEX(in_VERTEX B;face out_FACE, edge out_EDGE); //1.c
```

Most lines in this procedure are written in natural language, so they deserve no more explanation. Nevertheless, to implement line 12 a primitive must be developed. In fact, line 12 is the only one in this procedure that, in case of having to be evaluated, does not require constant time. We have called the new primitive `ENTITY_INCIDENT_VERTEX`. It takes as input a vertex  $V$  of  $G^*$ , where point  $A$  is located, and point  $B$  specifying the direction to follow. If the vertex is isolated (Figure 1.c.4), its containing face is reported. In any other case, the procedure does a binary search through the list of edges incident to vertex  $V$ .

```
ENTITY_INCIDENT_VERTEX (vertex V, point B; face out_FACE, edge out_EDGE)
1. if vertex V is isolated // 1.c.4
```

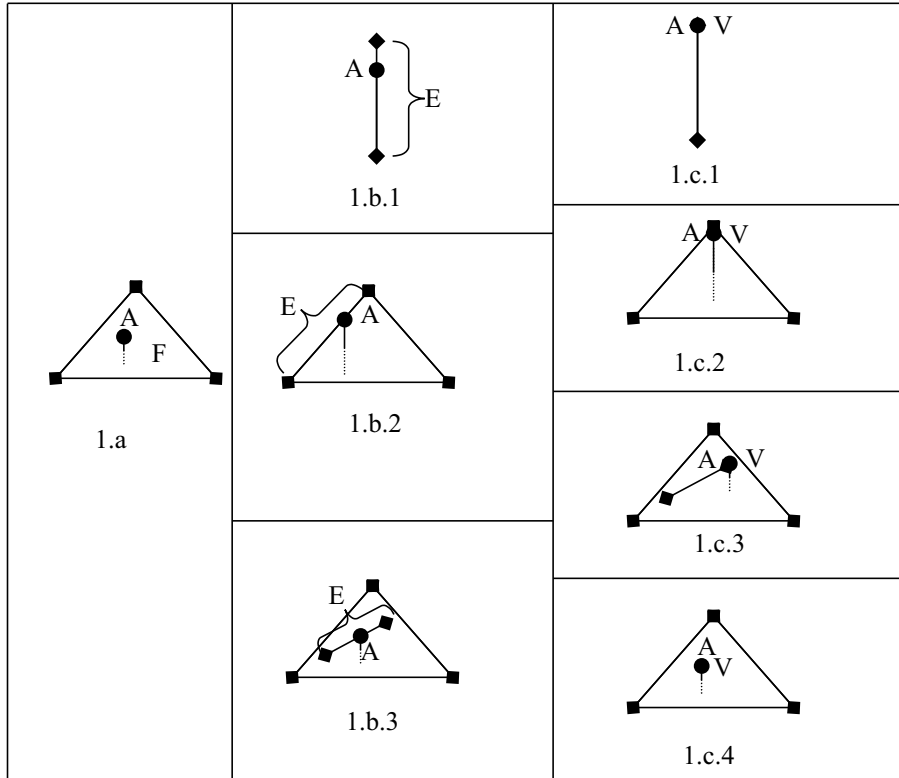


Figure 1: Situations considered by procedure FIND\_NEXT\_TRAVERSED\_ENTITY. Point A might be located at the interior of a face (1.a), at the interior of an edge (1.b), or at a vertex (1.c). In 1.b.1, it continues advancing along an edge, either belonging to a feature or to the proper boundary of any face. In 1.b.2, it continues walking through one of the faces incident to the proper boundary edge, and in 1.b.3 through the face containing the feature edge. In 1.c.1, A is located on a vertex and continues walking along an edge. In 1.c.2, it continues through one of the faces incident to the proper boundary vertex. In 1.c.3, A is located on an endpoint of a feature edge and continues walking through the containing face, and in 1.c.4, A is located on a feature vertex, hence it continues walking through the containing face.

```

2.  then report the face containing it
3.  else let E_L be the list of n edges incident to V (counterclockwise). Add the first element of E_L at
      the end of the list. Let E be the edge at the head of the list;
5.    if AB is totally or partially coincident with E
6.      then first_position := last_position := 1; advance := 0;
7.      else first_position := 1; last_position := n+1; advance := 1;
8.    while (advance > 0) do
9.      advance := int [ (last_position - first_position) / 2 ];
10.     let E be the edge in position first_position + advance in list E_L;
11.     if AB is totally or partially coincident with E
12.       then first_position := first_position + advance; last_position := first_position; advance := 0;
13.       else if AB is found counterclockwise between edge in first_position and E
14.         then last_position := first_position + advance;
15.         else first_position := first_position + advance;
16. if first_position = last_position
17.   then out_EDGE = edge in first_position in E_L // 1.c.1
18.   else out_FACE = face found rotating counterclockwise from edge in first_position around V //1.c.2

```

## 4.2 Point Location

The planar point location problem is a relevant topic of research in Computational Geometry. It consists on finding the face of a general plane subdivision that contains a given point. Using a *brute force* algorithm, a search time of  $O(n)$  would be required to find such location, being  $n$  the number of non-isolated vertices of the subdivision. If point location operation has to be accomplished a certain number of times, then a more efficient search structure would be desirable. In this case, the improvement in query time efficiency compensates the associated extra storage and time costs. Among these more efficient strategies we can mention the *trapezoidal decomposition method*, the *chain method*, the *triangulation refinement method* or the *persistence method*, all of which achieve  $O(\log(n))$  search time and  $O(n)$  storage solutions. Preprocessing time never improves  $O(n \log n)$  [6].

We are interested in finding the entity of a map  $M$ , represented by a expanded Plane Euclidean Graph  $G^*$ , where a point  $P$  not belonging to the map is located, taking advantage of the data structure encoding the map. The method we have chosen to solve this problem is based on the well-known *ray-shooting strategy* consistent in finding the proper boundary edge immediately above the point. It starts by considering the infinity face as departure entity. Then, going downwards along the vertical half-line with source endpoint  $P$ , another entity closer to  $P$  will be reached, and so on until point  $P$  is located. However, the concept of closeness, when dealing with non necessarily convex faces, must be clarified. Let's take a general oriented segment  $S$ , of source endpoint  $A$  and target endpoint  $B$  (Figure 2). A vertical half-line can be regarded as a degenerate case of segment with one of its endpoints at infinity. For entities  $e1$  and  $e2$  intersecting  $S$  being compared with respect to their closeness to  $A$ , each of them must have some point intersecting  $S$ . We say  $e1$  is previous to  $e2$  (or equivalently  $e1$  is closer to  $A$  than  $e2$  or  $e2$  is posterior to  $e1$ ) if the intersection point of  $e1$  with  $S$  closest to  $A$  lies between  $A$  and the intersection point of  $e2$  with  $S$  closest to  $A$ . We will also say  $e1$  is immediately previous (resp. posterior) to  $e2$  if there is no entity  $e3$  such that  $e1$  is previous to  $e3$  and  $e3$  is previous to  $e2$ .

Now we are defining some remarkable points belonging to open vertical half-line  $SL$ , which will be called *stop points* (see Figure 3). First stop point will be the point of  $SL$  closest to  $P$  that intersects the proper boundary of the infinite face. If such point does not exist, it means that point  $P$  is located in the infinite face of the map or one of its boundary entities. If, on the contrary, the first stop point exists,  $P$  must be located below it. To find next stop point some computations must be done to find next element traversed by  $SL$  (see Figure 4). If such element is an edge, it must be totally or partially contained in  $SL$ . Assuming  $P$  does neither belong to that edge nor coincides with any of its endpoints, next stop point will be its lowest endpoint, as this is the lowest boundary entity of the edge. In any other case, the actual stop point is last stop point and the location of  $P$  will have been found. If next traversed element is a face, then we should find the lowest intersection point of its proper boundary with  $SL$  and we will have found next posterior stop point. If such point does not exist, we will have reached the last stop point. Then there is no any proper

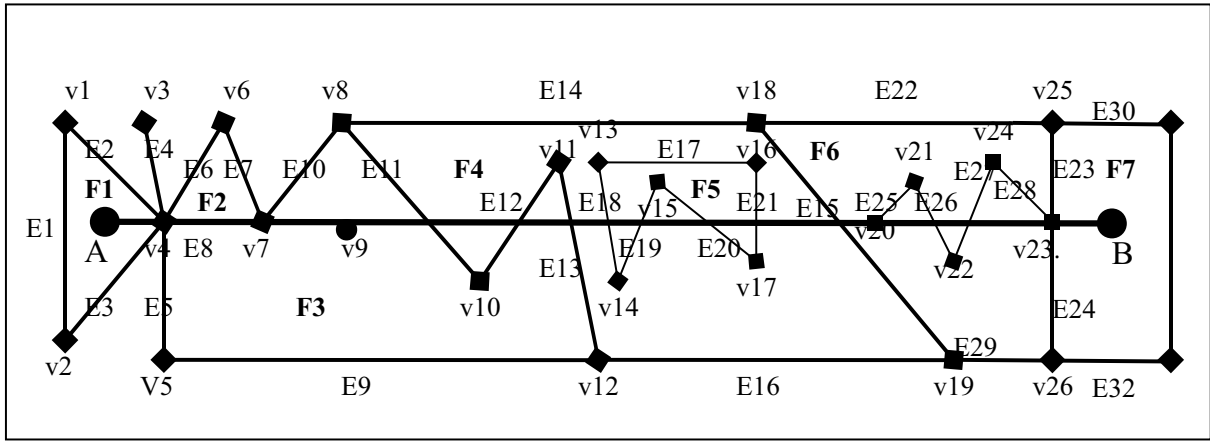


Figure 2: The sorted sequence of entities according to their closeness to A is F1-v4-E8-v7-F3-v9-E11-F4-E12-E13-E18-F5-E19-E20-E21-E15-F6-v20-E26-E27-v23-F7

boundary element intersecting SL at a point beneath last stop point, hence point P must be located at some entity belonging to the regularization of next entity traversed by SL from last stop point. Algorithm LOCATE.POINT consists in finding the sorted sequence of stop points. In order to avoid an edge being tested for intersection more than once, every edge will start being unmarked, but when it is tested for intersection with SL it becomes marked. Let's note that if an edge has been marked, its intersection with SL cannot be next stop point to be detected, as it is always above or coinciding with actual stop point. When an edge is marked, a pointer to it will be added to a list, whose elements will be unmarked at the end of the process. Auxiliary procedure ADVANCE takes as input a face, a boolean variable indicating if we are interested in testing for intersection its inner or its outer boundary, and the location of the actual stop point, and returns the location of next stop point. The reason for considering the boolean variable is that we will be interested in testing for intersection first those edges belonging to the outer boundary of the corresponding face. If such intersection does not occur, then we should test for intersection inner boundary edges. When last stop point is reached, the location of P is found straightforward (see Figure 5). Let's give a specification of the algorithm:

**LOCATE.POINT** (*point P, expanded PEG G\*, face PF, edge PE, vertex PV*)

*Input.* Point P. Expanded PEG G\*.

*Output.* Face PF, edge PE, or vertex PV of G\* where P lies in.

1. Compute open vertical half-line SL starting at P
2. Create a void list E\_ptr of pointers to edges
3. Compute the lowest intersection point of SL with the proper boundary of the infinite face of G\*. If such boundary edge does not exist then test if P belongs to some of its boundary entities, otherwise initialize stop point location (SP\_EDGE, SP\_VERTEX) and possible stop point location (PSP\_EDGE, PSP\_VERTEX) to the boundary entity, either edge or vertex, where the lowest intersection has occurred
4. **while** (the location of P has not been found) **do** // PF,PE,PV are NULL
5.     FIND\_NEXT\_TRAVERSED\_ENTITY (P, NULL, SP\_EDGE, SP\_VERTEX; F, E);
6.     **if** (next traversed entity is an edge) // E != NULL
7.         **then** mark it;
8.         **if** (P is beneath the lowest endpoint of E)
9.             **then** stop point location := the lowest endpoint of E; // SP\_VERTEX
10.            **else if** (P is at E) // 5.e
11.             **then** PE := E; // END
12.             **else** PV := lowest endpoint of E // END
13.         **else** ADVANCE (F, false, stop point location; possible stop point location, PV, PE);
14.         **if** (possible stop point location == stop point location)
15.             **then** ADVANCE (F, true, stop point location; possible stop point location, PV, PE);

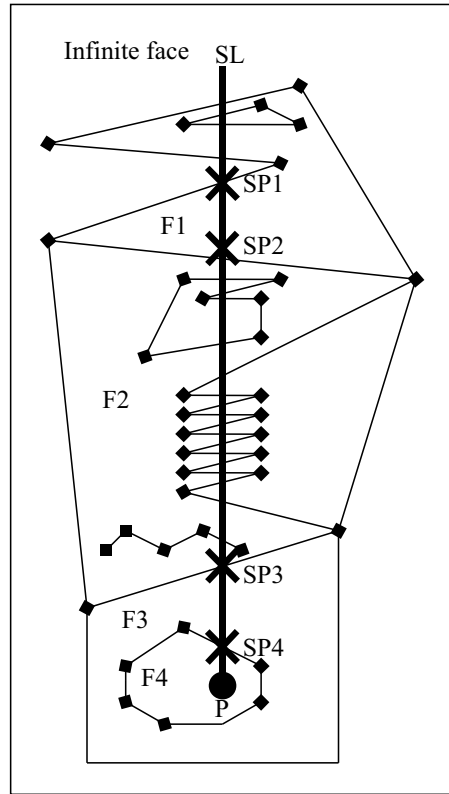


Figure 3: Example of steps followed in algorithm POINT\_LOCATION to find the location of P. The lowest intersection point of SL with the boundary of infinite face is the first stop point, SP1. Then some computations should be made for finding the entity SL continues walking through as well as the next stop point. They are, respectively, F1 and SP2. The procedure continues until last stop point, SP4, is found. Computations accomplished when we are placed at this point show that there is no stop point below it, that P is not coincident with it, and that F4 is the next face whose interior is traversed by SL, hence the algorithm detects that P belongs to F4. As P does not belong to the boundary of F4, P is located at the interior of F4.

```

16.           if (possible s p l == stop point location) // last stop point
17.               then if P coincides with a feature vertex V of F or is endpoint of a feature edge
18.                   then PV := V // END
19.                   else if it belongs to the interior of a feature edge
20.                       then PE := E // END
21.                       else PF := F; // 5.a, 5.c, END
22.           stop point location := possible stop point location;
23. unmark edges indicated in E_ptr

```

**ADVANCE** (*face F, bool inner, edge and vertex stop point loc; edge and vertex pos s p l, vertex PV, edge PE*);

```

1. if (inner == true)
2.     then L := list of the inner boundary edges of F
3.     else L := list of the outer boundary edges of F;
4. possible stop point location := stop point location;
5. for all unmarked edge E belonging to L do
6.     mark it and add a pointer to it to list E_ptr;
7.     if (E or one of its endpoints intersects SL at a point beneath possible stop point location)

```

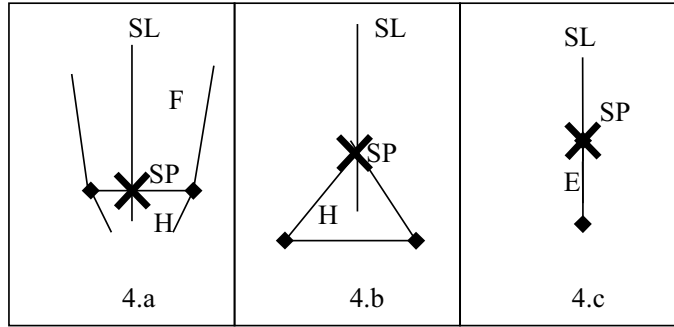


Figure 4: Possible cases that can arise when advancing downwards through adjacencies / incidences along semiline SL, departing from stop point SP. If such stop point belongs to the interior of a boundary edge (4.a), as the edge must belong to the proper boundary of a face F, the problem of walking is reduced to passing to face H adjacent to face F along the edge the stop point belongs to. Nevertheless, when the stop point coincides with a vertex at SL, a decision must be made about the path followed by SL. The possible paths are SL coming into a face H (4.b), or SL continuing walking along an edge E (4.c). In both situations it is not relevant if the entity immediate previous the vertex was either an edge or a face.

```

8.      then if (intersection occurs at an endpoint of E)
9.      then if (the intersection point is P) // 5.d
10.         then PV := endpoint of E // END
11.         else possible stop point location := endpoint of E;
12.     else if (the intersection point is P) // 5.b
14.         then PE := E // END
12.     else possible stop point loc := E;

```

Once the description of the algorithm has been done, it is straightforward to see that whatever the location of P is, algorithm POINT\_LOCATION detects it. If point P belongs to the regularization of the infinite face, then the corresponding entity will be reported in line 3 of the algorithm. In any other case, we must see that the sorted set of stop points is detected and that last stop point makes possible the correct location of P. The location of first stop point is found in line 3. Then, assuming that every stop point previous to the current one has been correctly found, the algorithm detects next stop point in case it exists. If next stop point does not exist, the algorithm finds the location of P. Let's consider we are placed at the current stop point, at line 4 of the algorithm, and let's see that the algorithm detects next stop point. The current stop point location is known. Procedure FIND\_NEXT\_TRAVERSED\_ENTITY, called in line 5, will report the immediate posterior element to the current stop point. This element must exist because P cannot coincide with any stop point. If the procedure reports an edge (line 6), and P does not belong to such edge, then by definition next stop point must be its lowest endpoint, what is detected in line 9. If, on the contrary, P belongs to the interior of the edge, it is detected in line 10, and the corresponding edge is reported by the algorithm as the location of P (line 11). And, finally, if P has died at the lowest endpoint of the edge, it is detected in line 12, and the corresponding vertex is reported by the algorithm as the location of P. If procedure FIND\_NEXT\_TRAVERSED\_ENTITY reports a face (line 13), we must find the lowest intersection point of half-line SL with the proper boundary of the face. If such intersection point belongs to the outer boundary of the face it is detected by a call to procedure ADVANCE in line 13 and it is saved as possible stop point location (lines 7,8,9,12 of ADVANCE). If there is no such intersection possible stop point location coincides with stop point location (line 4 of ADVANCE). ADVANCE also detects if P coincides with the intersection point (lines 10, 11, 13 and 14), and in this case returns the location of P. If once the outer boundary has been tested for intersection, possible stop point location continues having the same value that the location of current stop point (line 14), it means that no element of the outer boundary has intersected SL at a point beneath current stop point. Then, the inner boundary will be tested for intersection with SL. The procedure is the same that we have just explained for the outer boundary. If the possible stop point

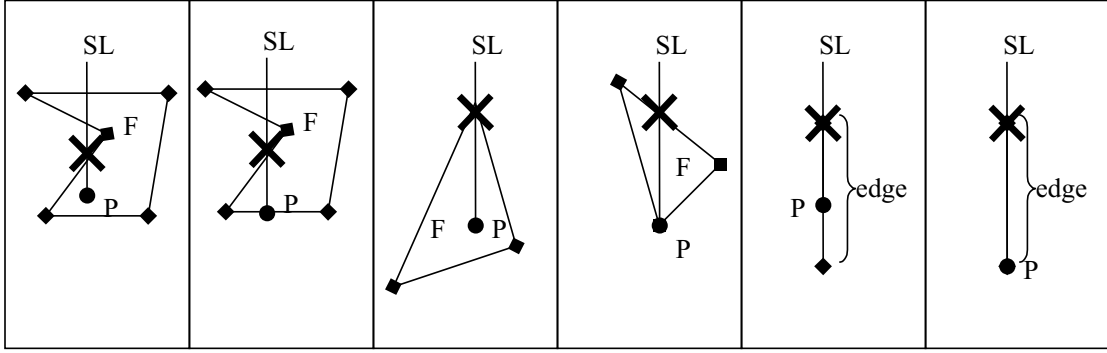


Figure 5: Determining the location of P once last stop point has been found. The cross signal represents the last stop point. In 5.a, 5.b, 5.c and 5.d next traversed entity is a face. Point P might be located at a proper boundary edge (5.b and 5.d), inside the face (5.a and 5.c) or at some of its features. If next traversed entity is an edge, point P might only be located at the edge or at its lowest endpoint.

location remains unchanged, then there is no proper boundary edge intersecting SL at a point lower than it, hence it only remains to see if it belongs to some feature or not, what is done in lines 17 to 21.

### 4.3 Intersection of an Expanded Plane Euclidean Graph $G^*$ and an expanded Polyline $L^*$

The algorithm to answer this query is based in that every vertex and edge in  $G^*$  belongs to the boundary of a face F, hence it suffices testing for intersection only those vertices and edges belonging to the boundaries of faces intersected by  $L^*$ . Hence, the number of edges and isolated vertices tested for intersection is equal to the number of intersecting edges and vertex features plus the number of non-intersecting edges and vertex features belonging to the boundary of some face intersecting the polyline.

We should start by establishing what kind of output we are interested in. The choice might depend on the particular query meaning. We will consider as output the sorted set of faces, edges and vertices found when moving from the source to the target point of  $L^*$ , according to the order established in section 4.2. If  $L^*$  is a closed chain, any vertex of its boundary can be chosen as starting endpoint of  $L^*$ . As  $L^*$  is a chain of straight line segments, an edge of  $G^*$  might be intersected several times. From now on we will consider a particular oriented segment of  $L^*$ , S, of source endpoint A and target endpoint B, intersections being computed for each segment independently. Figure 2 shows an example of how the entities intersecting S would be reported. A natural way of computing intersections with S consists in walking along the segment departing from A. Every time S comes the very first time into a face, every boundary element of the face must be visited and tested for intersection with S, in case it hadn't been previously done. A face in which this process has been accomplished will be called from now on a *visited face*. A sorted sequence of edges and feature vertices intersecting S will be maintained. When a face is visited, its intersecting boundary elements will be included in the sequence. Elements in the sequence with a chance of driving S to a non-visited face will be successively extracted to study if next element traversed by S is a non-visited face. In this case, it should be visited and the sequence updated. If the face had been already visited next element in the sequence will be extracted, and so on until the end of the segment is reached. We will recognize if a face has been already visited by marking it. During the walk along S, an edge totally or partially contained in S might be found. Such edges will be called from now on *contained edges*. If a contained edge is found that does not belong to any visited face, as we have no certainty that the segment reaches the interior of any of its incident faces, we are not interested in visiting them, and only the contained edge will be visited. An edge in this situation will be marked. Every edge carries information about the number of times it has been visited. If the edge is marked and none of its incident faces is marked, then it has been visited once. If at some posterior step one or both of its incident faces are visited, it will have been visited twice or three times respectively. And finally, if the edge is not marked and it is incident to one or two visited faces, it will have

been visited once or twice respectively. An edge will be considered type  $i$ , with  $0 \leq i \leq 3$ , if it has been visited  $i$  times. Let's note that every feature edge must be type 0 or 2.

The property the algorithm is based on is that, if when walking along  $S$  either the interior of a non-visited face or a non-visited contained edge is reached, its immediate previous element is a type 1 edge or a vertex incident to at least a type 1 edge. Let's see that the property holds. If  $S$  has just come into a non-visited face whose immediate previous element is an edge (Figure 6.a), the immediate previous element to the edge must be a visited face, hence the edge must be type 1. In case  $S$  has just reached a non-visited face whose immediate previous element is a vertex (Figure 6.b and 6.c), all edges incident to the vertex cannot be type 2 or 3 because at least one of them is a non-contained edge incident to the face that has not yet been visited, and all of them can neither be type 0 because the immediate previous element to the vertex must be either a visited face (Figure 6.b) or a visited contained edge (Figure 6.c). Hence, assuming there is no edge type 1, some edges incident to the vertex should be type 0 and some type 2 or 3, but this is not possible if there is not a type 1 edge incident to the vertex. The same proof is valid when  $S$  reaches a non-visited contained edge (Figures 6.d and 6.e). The previous property shows that for a non-visited face or a non-visited contained edge to exist, it is a necessary condition that there is a type 1 edge or a vertex incident to some type 1 edge immediately previous to it. However, it is not a sufficient condition, as we can observe in Figure 7, where every edge incident to vertex at P5 is a type 1 edge and, nevertheless, element immediately posterior to it is the edge intersecting  $S$  at P6.

Let's study when a type 1 edge gives access to an element that has not been visited. If the type 1 edge *crosses*  $S$ , then the immediate posterior element is always a non-visited face (Figure 6.a). However, nasty cases occur when the closure of a type 1 edge *touches*  $S$ . If the immediate posterior entity to the vertex is an edge (Figures 6.d and 6.e), then we will know if it has been visited or not depending on its type. If the entity first found when walking from the vertex is a face (Figures 6.b and 6.c), this face might also be an already visited one (see Figure 7, intersection point P2) or not (Figure 7, intersection point P7). Therefore, when the closure of a type 1 edge intersects  $S$  at some of its endpoints, we must check if next traversed element is marked or not.

We have seen that type 1 edges are all we need to detect the set of non-visited faces and non-visited contained edges. Hence, the procedure will consist, basically, in storing in a list every type 1 edge found when a face is being visited, or when a non visited contained edge is itself visited, sorted according to the following order, ' $<$ '. Let  $E1$  and  $E2$  be two edges whose closure intersects  $S$ . We will say  $E1 < E2$  if the intersection point of the closure of  $E1$  with  $S$  closest to  $A$ ,  $P1$ , lies between  $A$  and the intersection point of the closure of  $E2$  with  $S$  closest to  $A$ ,  $P2$ , or  $P1 = P2$  and  $E1$  is first found when rotating clockwise from  $S$  around  $P1$ . Next type 1 edge will successively be extracted from the list, testing if immediate posterior element (to it or to its corresponding endpoint) is a non-visited one and, in this case, repeating the process and updating the list.

Let's explain the algorithm in more detail. Let's start by assuming we are placed at the source endpoint of  $S$ ,  $A$ , whose location is known (after applying algorithm POINT\_LOCATION if  $S$  is the first segment in  $L^*$ ). It could be inside a face, an edge, or a vertex. If it is located on a vertex, the vertex must be reported. In any other case, the corresponding element will be reported later. Whatever the situation is, we must be able to find next element,  $e$ , immediately posterior to  $A$ . As we have already seen, procedure FIND\_NEXT\_TRAVERSED\_ENTITY accomplishes this task. If such element is an edge, then it will become a type 1 edge by marking it and will be added to a list  $list\_L$  of type 1 edges. If, on the contrary,  $S$  has succeeded in cutting a face  $F$ , it must be visited. Auxiliary procedure VISIT accomplishes this task: the face is marked, and the status of its boundary edges updated. The closure of every type 0 boundary edge in  $F$  is tested for intersection with  $S$ . If it intersects  $S$  it will be sorted in  $list\_L$  according to the order ' $<$ '. When a point feature contained in  $S$  is found, an edge NULL will be added to its corresponding position of  $list\_L$ , and the point feature will be added at the tail of a list  $list\_v$  of feature vertices. Once procedure VISIT has finished, an iterative process begins:  $list\_L$  is scanned until next type 1 edge is found. Every element previous in the list to the type 1 edge has no interest any more, so intersecting entities they produce will be reported according to the order established in 4.2 and the corresponding edges (including the type 1 edge) deleted from the list. If the closure of the type 1 edge intersected  $S$  at some of its endpoints, other type 1 edges incident to the intersecting endpoint closest to  $B$  might exist, and should also be removed from  $list\_L$ , as the corresponding intersecting vertex will have already been correctly handled. When an edge NULL is found, first element in  $list\_v$  is reported. Then, we must study if element immediately posterior to the

type 1 edge or to its corresponding endpoint, is a non-visited face or edge, and in this case it will be visited and the corresponding intersecting edges added to an auxiliary list of edges `list_l` that will be merged with `list_L`. Every time an edge or a face is marked, a pointer to it will be added in a list. When the segment dies, elements in both lists are unmarked. The algorithm must detect the location of the segment endpoint, as it is the source endpoint of next segment in  $L^*$ . Every edge whose closure intersects  $S$  will belong sooner or later to `list_L`, that will be initially void and will finish void. When one edge is deleted from the list, it is reported. Faces, on the contrary, must be reported when they are visited.

### LINE\_PEG\_INTERSECTION

*Input:* Expanded PEG  $G^*$  and expanded polyline  $L^*$ . It can be either open or closed.

*Output:* Sorted list of faces, edges and vertices of  $G^*$  intersecting  $L^*$

1. LOCATE\_POINT (source endpt of  $L^*$ ; F, E, V) and report the entity A belongs to if it is a vertex V;
2. Let `list_L`, `list_l` be lists of edges; `list_e`, `list_v` and `list_f` list of pointers to edges, vertices and faces resp.;
3. **for all** segment S belonging to  $L^*$  **do**
4.   B := target endpoint of S;
5.   FIND\_NEXT\_TRAVERSED\_ENTITY (B, F, E, V; FA, EA);
6.   F := NULL; E:= NULL; V:= NULL; // location source endpoint next S in  $L^*$
7.   **if** (segment continues walking along edge EA) // EA != NUL
8.     **then** add EA to `list_L`, mark it, and add a pointer to it to `list_e`;
9.     **else** VISIT (S, FA; `list_L`) and report FA; // S continues walking inside FA
10.  **while** (`list_L` has any type 1 edge) **do**
11.   E1:= next type 1 edge in `list_L`;
12.   delete of `list_L` every el. until E1 (included), reporting the sorted els intersecting S;
13.   **if** (S dies somewhere at the closure of E1)
14.     **then** make V (endpoint of E1) or E (E1) represent the location of the death place
15.     **else if** some endpoint (E1) belongs to S // intersection at endpoint of type 1 edge
16.       **then** F\_N\_T\_E (B, NULL, NULL, intersecting endpoint(E1); FA, EA);
17.       delete from `list_L` every edge incident to the intersecting endpoint;
18.       **if** EA != NULL // next traversed entity is contained edge
19.         **then if** E2 is type 0 // non-visited
20.         **then** add it to `list_l`, mark, pointer to `list_e`;
21.         **else if** FA is non-visited // remains unmarked
22.         **then** VISIT (S, FA; `list_l`) and report FA;
23.       **else** F\_N\_T\_E (B, NULL, E1, NULL; FA, EA);
24.       VISIT (S, FA; `list_l`) and report FA; // EA= NULL
25.       MERGE\_LISTS (`list_L`, `list_l`);
26.  **if** (`list_L` is void)
27.   **then if** (V=NULL and E=NULL) // S has not died at a type 1 edge
28.     **then** F := face after edge E1 (if E1 does not exist, F := FA)
29.   **else** get last element E2 of `list_L`
30.     **if** S dies somewhere at the closure of E2
31.     **then** either V := endpoint of E2 or E:=E2;
32.     **else** F is next face traversed from E2;
33.  delete every element in `list_L`, reporting sorted the intersecting elements;
34.  unmark elements indicated in `list_e` and `list_f`;

**VISIT** (segment S, face F; list of pointers to edges `list_l`);

/\* This procedure takes as input a segment S and a non-visited face F. When the procedure finishes, the face will be a visited one.\*/

1. mark F;
2. **for all** edges E of F **do**
3.   **if** E is a type 0 edge
4.     **then if** closure(E) INTERSECTS S
5.       **then** update `list_l`;

6. **for all** feature vertices  $V$  of  $F$  do
7.     **if**  $V$  is contained in  $S$
8.         **then** add a NULL edge to its corresponding place in  $list\_l$ ;
9.         add a pointer to it to the tail of  $list\_v$ ;

Procedure VISIT handles correctly every non-visited face. First the face is marked. Then every boundary element will be visited. The closure of every type 0 edge and every vertex feature is tested for intersection with  $S$  (lines 4 and 7 respectively). If an edge is found that intersects  $S$ , it is added to its corresponding place in list  $list\_l$ . In case we have a feature vertex contained in  $S$  a NULL edge is added to its place in list  $list\_l$  and the corresponding vertex to the tail of a list  $list\_v$ .

Let's see that algorithm LINE\_PEG\_INTERSECTION detects every intersection. As we are computing the intersections of  $G^*$  with the polyline as independent intersections of  $G^*$  with every segment in the polyline, it suffices that the algorithm works for one such segment, let's say the first one, and that the connection between consecutive segments is properly accomplished. To see that the algorithm does not miss any intersection, as only contained edges or elements belonging to intersecting faces may intersect  $S$ , and as we have already seen that procedure VISIT handles properly every boundary entity of a face intersecting  $S$ , it suffices to see that each contained edge not belonging to any visited face and every face intersecting  $S$  is detected. We will use the established order of intersecting entities to inductively demonstrate it. The first step consists on seeing that the first either contained edge or face intersecting  $S$  is visited. The location of the source endpoint of  $S$  is known (for the source endpoint of  $L^*$  it is detected by algorithm LOCATE\_POINT, and for consecutive segments it is given in lines 14,18,31,32). In line 5, procedure FIND\_NEXT\_TRAVERSED\_ENTITY detects first contained edge,  $EA$ , in case it exists, or reports first face,  $FA$ , cut by  $S$ . In the former case, the edge is added to list  $list\_L$ . In the latter case, the face is visited (line 9) and its intersecting edges are sorted and added to list  $list\_L$ . Now we must see that if  $e$  is a non-visited element, that can be either a non-visited contained edge  $E$  or face  $F$  whose interior is cut by  $S$ , and we assume that all previous contained edges have been detected and previous faces whose interior intersects  $S$  have been visited, then  $e$  will be visited. The non-visited element must have as immediate previous element or as element incident to the immediate previous element a type 1 edge, which must have been detected and must be next element in  $list\_L$ . If the segment does not die at such edge (otherwise procedure FIND\_NEXT\_TRAVERSED\_ENTITY couldn't be called), the call to procedure, made in lines either 16 or 23 depending on the location of the intersection point, finds next entity traversed by  $S$ . If next traversed element is a non-visited face (lines 21 and 23), then it will be visited (lines 22 or 24), and if it is a non-visited contained edge (lines 18,19) it is visited right now (line 20). In both cases, intersecting entities are sorted in a list  $list\_l$ , that will finally be merged with  $list\_L$ . The location of the segment endpoint must be detected. If the segment dies at a type 1 edge, it is detected in line 13; if it dies in the face next traversed from last type 1 edge, it is detected in line 28. Otherwise it must die at last element of  $list\_L$  or at the face after it. It is detected in lines 29-32. If the location of the source endpoint of  $L^*$  is a vertex, it is reported. Every face is reported when it is visited. The rest of intersecting entities will be, sooner or later, represented in  $list\_L$ , and every element in it is reported in its correct order (lines 12 and 33).

#### 4.4 Intersection of an Expanded Plane Euclidean Graph $G^*$ and an expanded region $R^*$

As a quite natural extension of the two previous algorithms, the method chosen to answer this query consists on finding the sorted sequence of entities in  $G^*$  intersecting every connected component of the boundary of  $R^*$ , directly given by algorithm LINE\_PEG\_INTERSECTION, and the entities in  $G^*$  completely contained in  $R^*$ . We assume that, although  $R^*$  does not belong to  $G^*$ , it corresponds to the definition of region in it, hence it is allowed to have cuts, holes and punctures. Let's describe the general procedure. Faces will be classified as marked faces, such that we know if they intersect  $R^*$ , or are completely inside or outside it, and unmarked faces, the rest of the faces. At the beginning of the procedure every face is unmarked. Those faces whose interior or the interior of some of its boundary edges intersect some boundary entity of  $R^*$  will become marked type 1 faces. To detect the type 1 faces, we will apply algorithm LINE\_PEG\_INTERSECTION to every connected component of the boundary of  $R^*$ . Every face adjacent to one of these type 1 faces contained

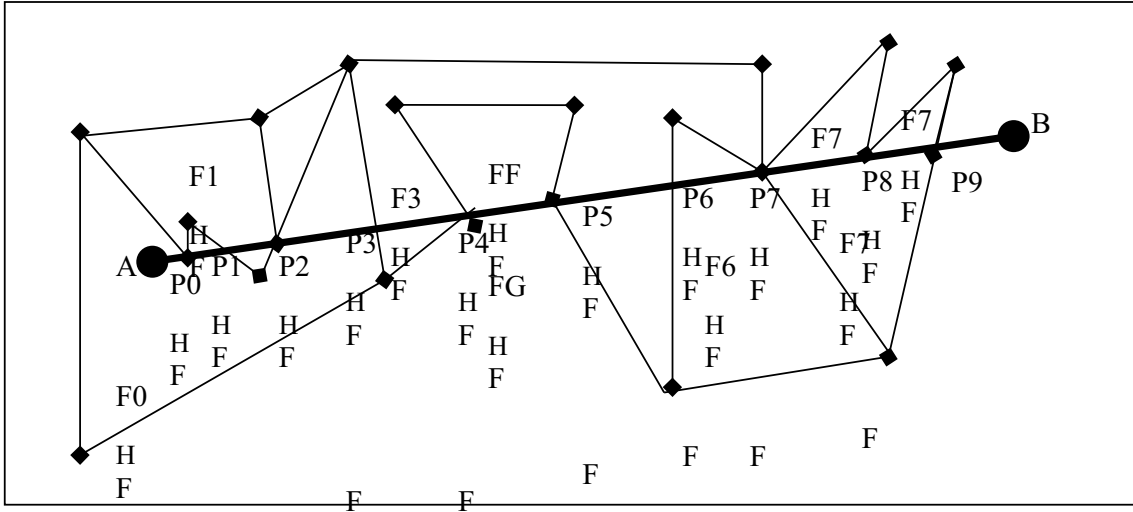


Figure 6: Illustration of algorithm LINE\_PEG\_INTERSECTION.

in  $R^*$  will become a type 2 face. If each of these type 2 faces is detected, then every face contained in  $R^*$  also will, it suffices finding every unmarked face adjacent to a type 2 face, as every face adjacent to a type 2 face must be completely contained in  $R^*$ . Then every type 0 face adjacent to a type 2 face will become itself a type 2 face, and so on. With this procedure, every face completely contained in  $R^*$  is detected, as it is not possible to have a face contained in  $R^*$  not connected through an adjacency path of type 0 faces to a type 2 face because, for this to happen, the type 0 face should reach through every type 0 faces adjacency path a type 1 face, but if the type 0 face is inside  $R^*$ , it is not possible to reach any type 1 face through adjacencies without reaching before a type 2 face, as every face adjacent to a type 1 face inside  $R^*$  is a type 2 face. The output of the algorithm will be, for each connected component of the boundary of  $R^*$ , the set of sorted entities intersecting it (according to the order established in 4.2), plus those features belonging to some type 1 face that are completely inside  $R^*$ , plus the set of faces completely contained in  $R^*$ .

### REGION\_PEG\_INTERSECTION

*Input* . Expanded PEG  $G^*$  and expanded region  $R^*$ .

*Output*. Set of type 1 faces, with the sequence of elements intersecting every connected component of the boundary of  $R^*$ , plus those features belonging to some type 1 face that are completely inside  $R^*$ , plus the set of type 2 faces.

1. Compute, using algorithm POINT\_LOCATION, the location of every vertex feature of  $R^*$  in  $G^*$ . Make the faces containing it type 1 and add them to a list L1. If the vertex feature of  $R^*$  belongs to some feature of  $G^*$ , mark the feature in  $G^*$ .
2. Compute, using algorithm LINE\_PEG\_INTERSECTION, the intersection of every edge feature of  $R^*$  with  $G^*$ . If an edge feature does not intersect any boundary element of  $G^*$ , LINE\_PEG\_INTERSECTION will report the face containing it, that will become type 1. Any other face of  $G^*$  intersecting any feature edge of  $R^*$  must also become type 1. Add the type 1 faces found to list L1. Mark the features of every face in L1 intersecting  $R^*$ .
3. Compute, using algorithm LINE\_PEG\_INTERSECTION, the intersection of every connected component of the proper boundary of  $R^*$  with  $G^*$ . Find the faces intersecting some element of the proper boundary of  $R^*$ , add them to list L1 and make them become type 1. Mark every intersecting feature of these faces.
4. **for all** type 1 face  $F$  **do**
5.     report  $F$ ;
6.     **for all** unmarked feature contained in  $F$  **do**
7.         **if** (one of its vertices is inside  $R^*$ )
8.             **then** the feature is contained in  $R^*$  and we report it;

```

9.   for all unmarked face A adjacent to F do
10.    if (one vertex in A is contained in R*)
11.     then transform A in type 2;
12.     add A to list L2;
13.     while L2 is not void do
14.       extract & delete first face,A,from L2;add it to L3;
15.       for all adjacent unmarked face B to A do
16.         transform B in type 2;
17.         add B to L2;
18.     else mark A; // A is outside R*
19. report faces in L3; // set of faces completely contained in R*

```

#### 4.5 Low-Level Primitives necessary for implementing the previous algorithms

The CGAL Library provides a data type representing a directed straight ray in the two-dimensional Euclidean plane. This library also offers the tools to find the relative position of a point related to an edge, and the functionality needed to manipulate lists in optimal time.

The MultiMap Library provides functions to mark and unmark edges and to iterate through the lists of entities given by any total topological relation.

## References

- [1] Bertolotto, M.: *Geometric Modeling of Spatial Entities at Multiple Levels of Resolution*. Ph.D.Thesis. Department of Computer Science. University of Genova. DISI-TH-1998-01, 1998.
- [2] Clementini, E., Di Felice, P.: *A comparison of methods for representing topological relationships*. Information sciences 80, 1-34, 1994.
- [3] Clementini, E., Di Felice, P.: *A Model for Representing Topological Relationships Between Complex Geometric Features in Spatial Databases*. Information Sciences, 90 (1-4): 121-136, 1996.
- [4] Clementini, E., Di Felice, P., van Oosterom, P.: *A small set of formal topological relationships suitable for end-user interaction*. Advances in Spatial Databases-Third International Symposium. Springer-Verlag. Singapore. SSD'93 LNCS 692, 277-295, 1993.
- [5] De Floriani, L., Marzano, P., Puppo, E.: *Spatial queries and data model*. Spatial Information Theory - A theoretical basis for GIS, A.U. Frank, I. Campari (Eds.), LNCS Vol.716, Springer-Verlag, pp.113-138, 1993.
- [6] de Berg, M., van Kreveld, M., Overmars, M., Schwarzkopf, O.: *Computational Geometry. Algorithms and Applications*. Springer, 1997.
- [7] Edelsbrunner, H., Guibas, L.J., Stolfi, J.: *Optimal point location in a monotone subdivision*. SIAM J.Computing, 15:317-340, 1986.
- [8] Egenhofer, M.: *A formal definition of binary topological relationships*. Third International Conference on Foundations of Data Organization and Algorithms. Lecture Notes In Computer Science, Vo. 367, Pp. 457-472, 1989.
- [9] Egenhofer, M.J.: *A model for detailed binary topological relationships*. Geomatica, 47 (3&4), 261-273.
- [10] Egenhofer, M., Franzosa, R.: *Point-set topological spatial relations*. International Journal of Geographical Information Systems, 5, 2, pp.161-174, 1991.
- [11] Egenhofer, M.J., Herring, J.: *Categorizing binary topological relationships between regions, lines and points in geographic databases*. Tech. Report, Department of Surveying Engineering, University of Maine, Orono, ME 1991.

- [12] <http://www.CGAL.org>.
- [13] Kainz, W.: *Spatial relationships-topology versus order*. Proceedings of Fourth International Symposium on Spatial Data Handling. pp. 814-819, 1990.
- [14] Kovalevsky, V.A.: *Finite topology as applied to image analysis*. Computer Vision, Graphics, and Image Processing, 46, pp.141-161, 1989.
- [15] Lundell, A.T., Weingram, S.: *The Topology of CW Complexes*. Van Nostrand Reinhol Comp., 1969.
- [16] *OpenGIS Simple Features Specification for SQL. Revision 1.1*. OpenGIS Project Document 00-049. 1999.
- [17] Preparata, F.P., Shamos, M.I.: *Computational Geometry: an Introduction*, Springer-Verlag, 1985.
- [18] Puppo, E., Dettori, G.: *Towards a Formal Model for Multiresolution Spatial Maps*. Advances in Spatial Databases, Max J. Egenhofer, John R. Herring (Eds.), LNCS Vol.951, Springer-Verlag, pp.152-169, 1995.