Variable-scale Topological Data Structures Suitable for Progressive Data Transfer: The GAP-face Tree and GAP-edge Forest

Peter van Oosterom

ABSTRACT: This paper presents the first data structure for a variable scale representation of an area partitioning without redundancy of geometry. At the highest level of detail, the areas are represented using a topological structure based on faces and edges; there is no redundancy of geometry in this structure as the shared boundaries (edges) between neighbor areas are stored only once. Each edge is represented by a Binary Line Generalization (BLG)-tree, which enables selection of the proper representation for a given scale. Further, there is also no geometry redundancy between the different levels of detail. An edge at a higher importance level (less detail) does not contain copies of the lower-level edges or coordinates (more detail), but it is represented by efficiently combining their corresponding BLG trees. Which edges have to be combined follows from the generalization computation, and this is stored in a data structure. This data structure turns out to be a set of trees, which will be called the (Generalized Area Partitioning) GAP-edge forest. With regard to faces, the generalization result can be captured in a single tree structure for the parent-child relationships—the GAP face-tree. At the client side there are no geometric computations necessary to compute the polygon representations of the faces, merely following the topological references is sufficient. Finally, the presented data structure is also suitable for progressive transfer of vector maps, assuming that the client maintains a local copy of the GAP-face tree and the GAP-edge forest.

KEYWORDS: Map generalization, topological structure, planar partition, client/server, progressive data transfer, geo-information system

Introduction

Data structures supporting variable scale data sets are still very rare. There are a number of data structures available for multi-scale databases based on multiple representations, that is, the data are used for a fixed number of scale (or resolution) intervals. These multiple representation data structures attempt to explicitly relate objects at different scale levels, in order to offer consistency during the use of the data. The drawbacks of the multiple representations data structures are that they do store redundant data (same coordinates, originating from the same source) and that they support only a limited number of scale intervals. Most data structures are intended to be used during the pan and zoom (in and out) operations, and in that sense multi-scale data structures are already a serious improvement for interactive use as they do speed-up interaction and give reasonable representations for a given level of detail (scale).

Peter van Oosterom, Delft University of Technology, Section GIS-technology, Jaffalaan 9, 2628 BX Delft, The Netherlands. Tel: +31 15-2786950; Fax +31 15 2782745. Email: <oosterom @otb.tudelft.nl>.

Need for Progressive Data Transfer

Another drawback of multiple representation data structures is that they are not suitable for progressive data transfer, because each scale interval requires its own (independent) graphic representation be transferred. Good examples of progressive data transfer are raster images, which can be presented relatively quickly in a coarse manner and then refined as the user waits a little longer. These raster structures can be based on simple (raster data pyramid) (Samet 1984) or more advanced (wavelet compression) principles (Lazaridis and Mehrotra 2001; Hildebrandt et al. 2000; Rosenbaum and Schumann 2004). For example, JPEG2000 (wavelet based) allows both compression and progressive data transfer from the server to the end-user. Also, some of the proprietary formats such as ECW from ER Mapper and MrSID from LizardTech are very efficient raster compression formats based on wavelets and offering multi-resolution suitable for progressive data transfer. Similar effects are more difficult to obtain with vector data and require more advanced data structures, though, recently, a number of attempts have been made to develop such structures (Bertolotto and Egenhofer 2001; Buttenfield 2002; Jones et al. 2000; Zhou et al. 2004).

Multi-scale / Variable-scale Vector Data Structures

For single (line) objects, a number of multi-scale/variable-scale data structures have been proposed: Strip-tree (Ballard 1981), Multi-Scale Line tree (Jones and Abraham 1986), Arc-tree (Günter 1988), and the Binary Line Generalization tree (BLG tree) (van Oosterom 1990). The Strip-tree and the Arc-tree are intended for arbitrary curves, not for simple polylines. The Multi-Scale Line tree is intended for polylines, but it introduces a discrete number of detail levels and it is a multi-way tree, meaning that a node in the tree can have an arbitrary number of children. The BLG tree is a binary tree for a variable scale representation of a polylines, based on the Douglas-Peucker (1973) line generalization algorithm. The BLG tree will be explained in more detail in a later section (see Figure 7). Note that these line data structures cannot be used for spatial organization (indexing, clustering) of multiple objects (as needed by variable scale or multi-scale map representations), so they only solve part of the generalization and storage problem.

One of the first multi-scale vector data structures designed to avoid redundancy was the reactive BSP-tree (van Oosterom 1989), which supports both spatial organization (indexing) and multiple level of details. Its main disadvantage, however, is that it is a static structure. The first dynamic vector data structure supporting spatial organization of all map objects, as well as multiple scales, was the Reactive tree (van Oosterom 1992; 1994). The Reactive tree is an R-tree (Guttman 1984) extension with importance levels for objects: more important objects are stored higher in the tree structure, which makes more important object more accessible. This is similar to the reactive BSP-tree, but the dynamic structure of the Reactive tree enables inserts and deletes, functions that the BSP-tree lacks. The BLG tree and the Reactive tree are eminently capable of supporting variable-scale/multi-scale maps composed of individual polyline or polygon objects.

Generalized Area Partitioning

The BLG-tree and Reactive-tree structures are not well suited for an area partitioning, since removal of a polygon results in a gap in the map and independent generalization of the boundaries of two neighbor areas results in small slivers (overlaps or gaps). Overcoming this deficiency was the motivation behind the development of the GAP tree (van Oosterom 1993). The BLG-tree, Reactive-tree, and GAP-tree data structures can be used together, while each supports different aspects of the related generalization process, such as selection and simplification, for an area partitioning (van Oosterom and Schenkelaars 1995).

Following the conceptualization of the GAP tree, several improvements were published to resolve limitations of the original data structures (van Putten and van Oosterom 1998; Ai and van Oosterom 2002; Vermeij et al. 2003). This paper introduces the new topological GAP tree, which combines the use of the BLG tree and the Reactive tree and avoids the problems of the original GAP tree—redundant storage and slivers near the boundary of two neighbor areas. Then the implementation of the structure is discussed, followed by an explanation of how to use it for progressive data transfer. Finally, a summary of the most important results is provided, together with suggestions for further research.

GAP Tree Background

The first tree data structure for generalized area partitioning (GAP tree) was proposed by van Oosterom (1993). The idea was based on first drawing the larger and more important polygons (area objects), so as to create a generalized representation. However, one can continue by refining the scene through the additional drawing of the smaller and less important polygons on top of the existing polygons (based on the Painters algorithm; see Figure 1). This principle has been applied to the Digital Land Mass System – Digital Feature Analysis Data (DLMS-DFAD) data sturcture (DMA 1986), because it already had this type of polygons organization. When tested with the Reactive tree and the BLG tree, it was possible to zoom in (zoom out) and obtain map representations with more (less)

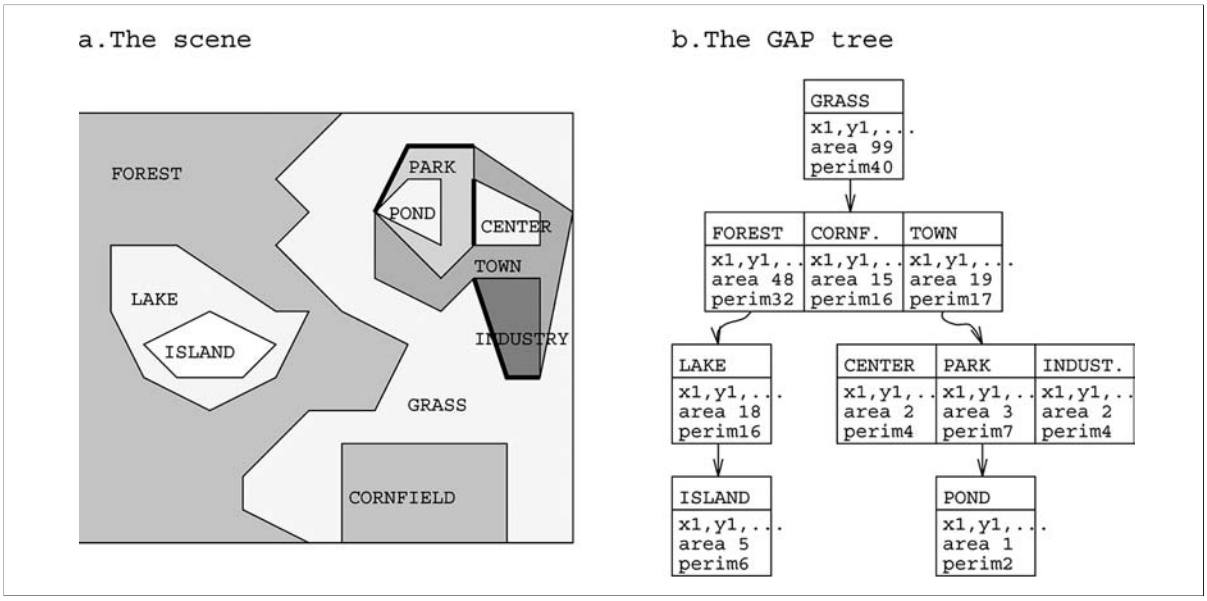


Figure 1. The original GAP tree (van Oosterom 1993).

detail of a smaller (larger) region in constant time (see Figure 2, left).

Computing the GAP Tree

If one has a normal area partition (and not DLMS DFAD data) one first has to compute the proper structure. This is driven by two functions. First, the importance function (for example: *Importance(a) = Area(a) * WeightClass (a)*) is used to find the least important feature *a* based on its size and the relative importance of the class it belongs to. Then the neighbor *b* is selected based on the highest value of *Collapse(a,b) = Length(a,b) * CompatibleClass(a,b),* with *Length(a,b)* being the length of the common boundary. Feature *a* is removed and feature *b* takes its space on the map. In the GAP tree this is represented by linking feature *a* as the child of parent *b* (and enlarging the original feature *b*). This process is repeated until only one feature is left covering the whole domain, forming the root of the GAP tree. Figure 1 gives a schematic representation of such a GAP tree.

Work by van Smaalen (1996; 2003) focuses on finding neighbor patterns, which might in turn be used for setting up an initial compatibility matrix. Bregt and Bulens (1996) give area generalization examples in the domain of soil maps, based on the same principles. Both van Smaalen and Bregt and Bulens use an adapted classification for the higher (merged) level of objects, instead of keeping the original classification at all levels of detail; e.g., deciduous forest and coniferous forest objects are aggregated into a new object classified as “forest” or “garden,” while house and parking place objects form the new object “lot.” This could also be done in the GAP tree.

Implementations and Improvements of the GAP Tree

Though the GAP tree may be computed for a source data set which has a planar partitioning topology, the GAP tree itself is not a topological structure. Each node in the GAP tree is a polygon, and this introduces some redundancy as parents and child may have some parts of their boundary in common. The first true GAP-tree construction based on topologically structured input was implemented by van Putten and van Oosterom (1998) for two real world data sets: Top10vector (1:10.000) and GBKN (1:1.000; see Figure 2 right). It turned out that finding the proper importance and compatibility functions (which drive the GAP-tree construction) is far from trivial and depends on the purpose of the map. In addition, two improvements were presented in the 1998 paper (at the conceptual level): 1) adding parallel lines to “linear” area features, and 2) computing a GAP tree for a large seamless data set.

Ai and Van Oosterom (2002) presented two other possible improvements to the GAP tree: One was that one should not assign the least important object to only one neighbor, but subdivide this object along its skeleton and assign the different parts to different neighbors/parents (the result is

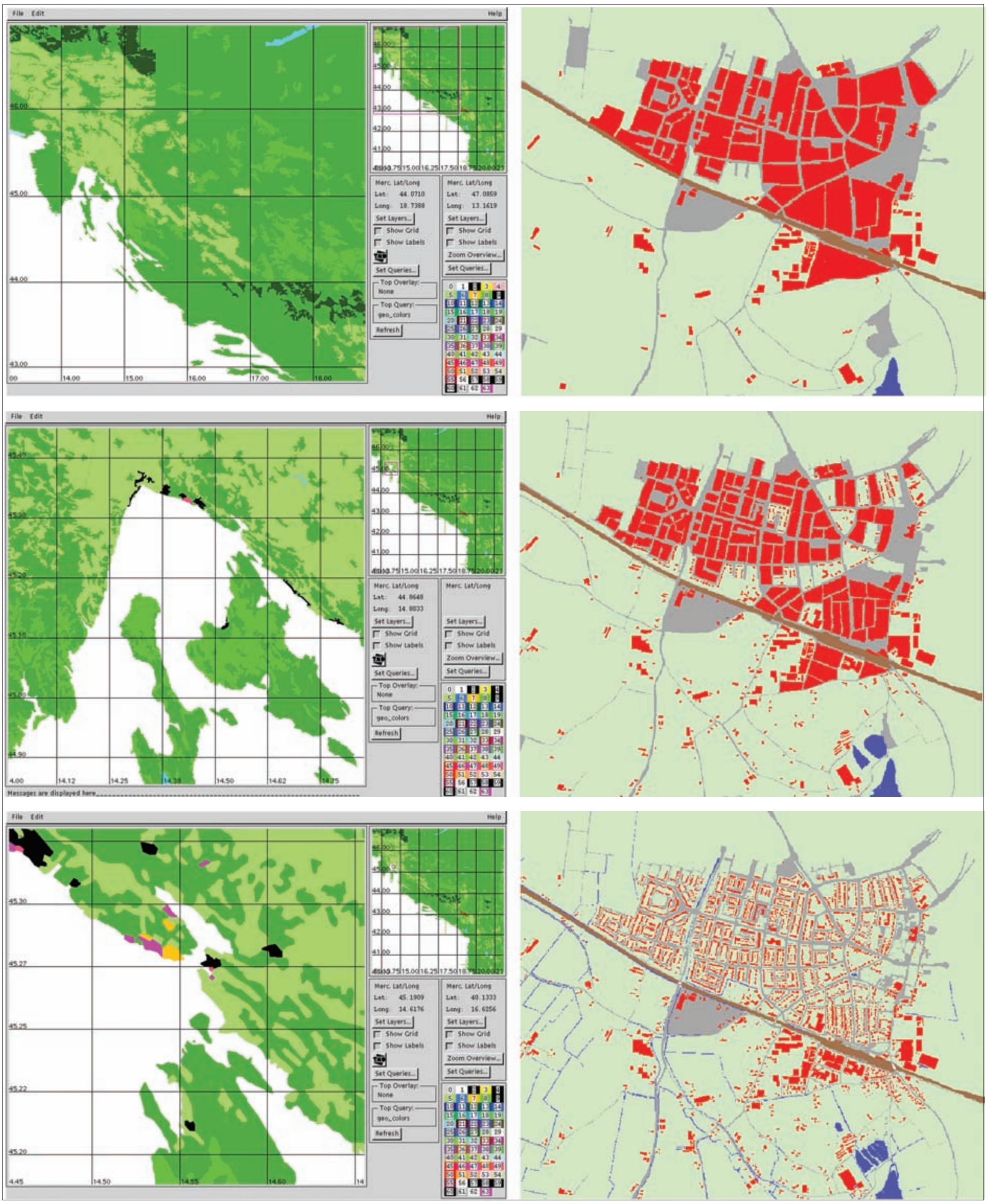


Figure 2. Left: GAP-tree principle applied to DLMS DFAD (add detail when zooming in). Right: GAP tree applied to large-scale topographic data set (shown at same scale).

not a tree but a directed acyclic graph: GAP-DAG). The second improvement concerned extending the neighborhood analysis by considering non-direct (sharing a common edge) neighbor areas as well. Both suggestions are based on an analysis using a Triangular Irregular Network (TIN) structure.

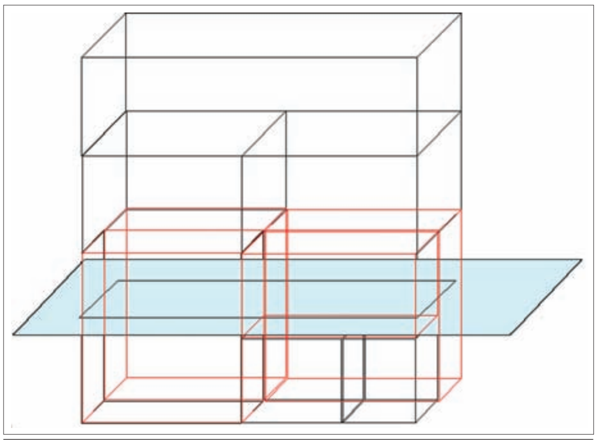


Figure 3. Importance levels represented by the third dimension (at the most detailed level (bottom) there are several objects, while at the most coarse level (top) there is only one object). The hatched plane represents a requested level of detail, and the intersection with the symbolic 3D volumes then gives the faces.

Topological Version of the GAP Tree

All improvements still result in a non-topological GAP structure, which means that it contains redundancy. Vermeij et al. (2003) presented a GAP-tree structure that avoids most redundancy by using a topological structure for the resulting GAP tree, not only for the input: thus the edges and the faces table both have attributes that specify the importance ranges in which a given instance is valid. The 2D geometry of the edges (and faces) is extended by the importance value range (on the z-axis) for which it is valid (see Figure 3). One drawback of this approach is that it requires considerable geometric processing at the client side—clipping edges, forming rings, and linking outer and possible inner rings to a face. A second drawback is that there is some redundancy introduced via the edges at the different importance levels: i.e., the coordinates of detailed edges are again present in the edge at the higher aggregation level.

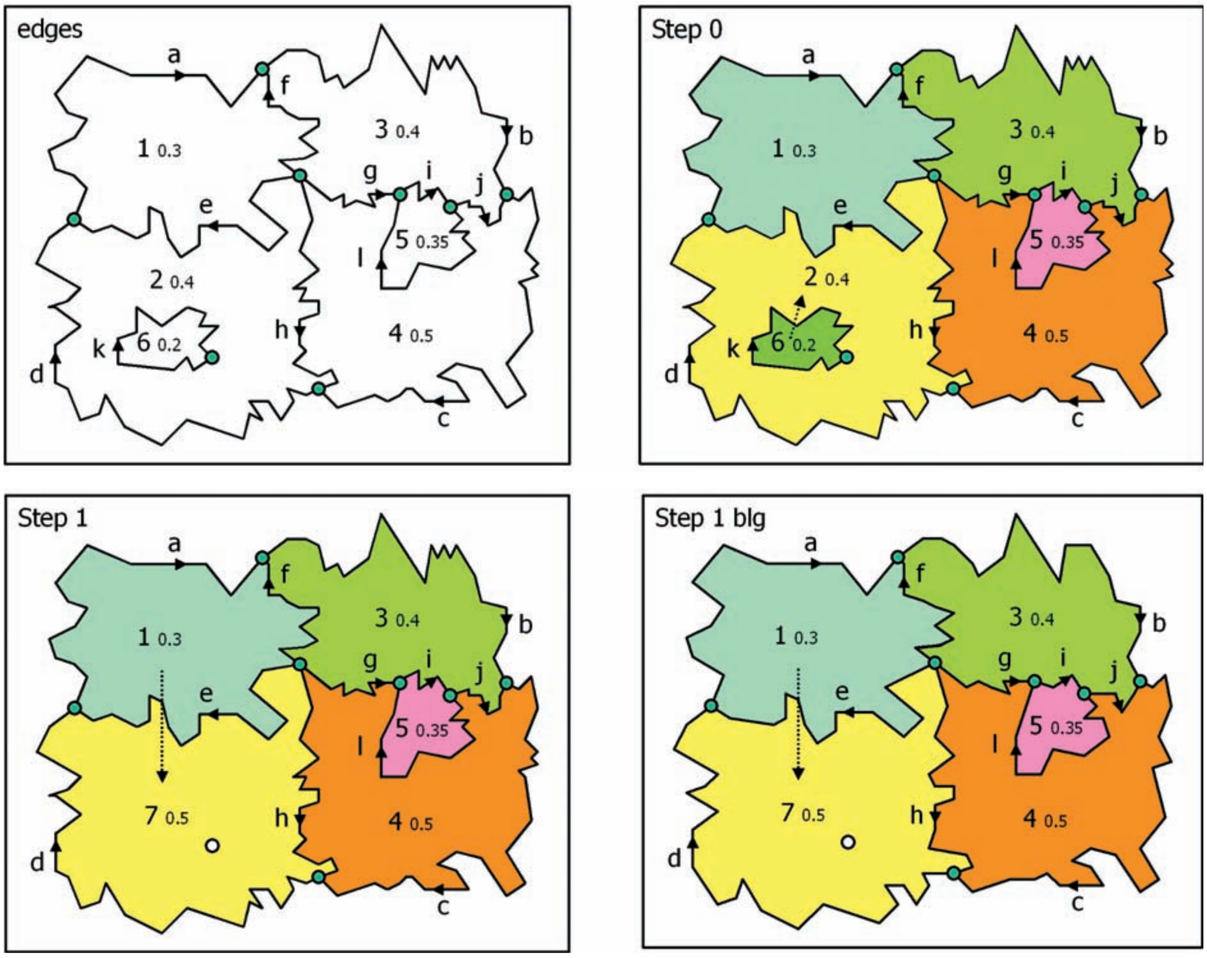


Figure 4, continued on page 336.

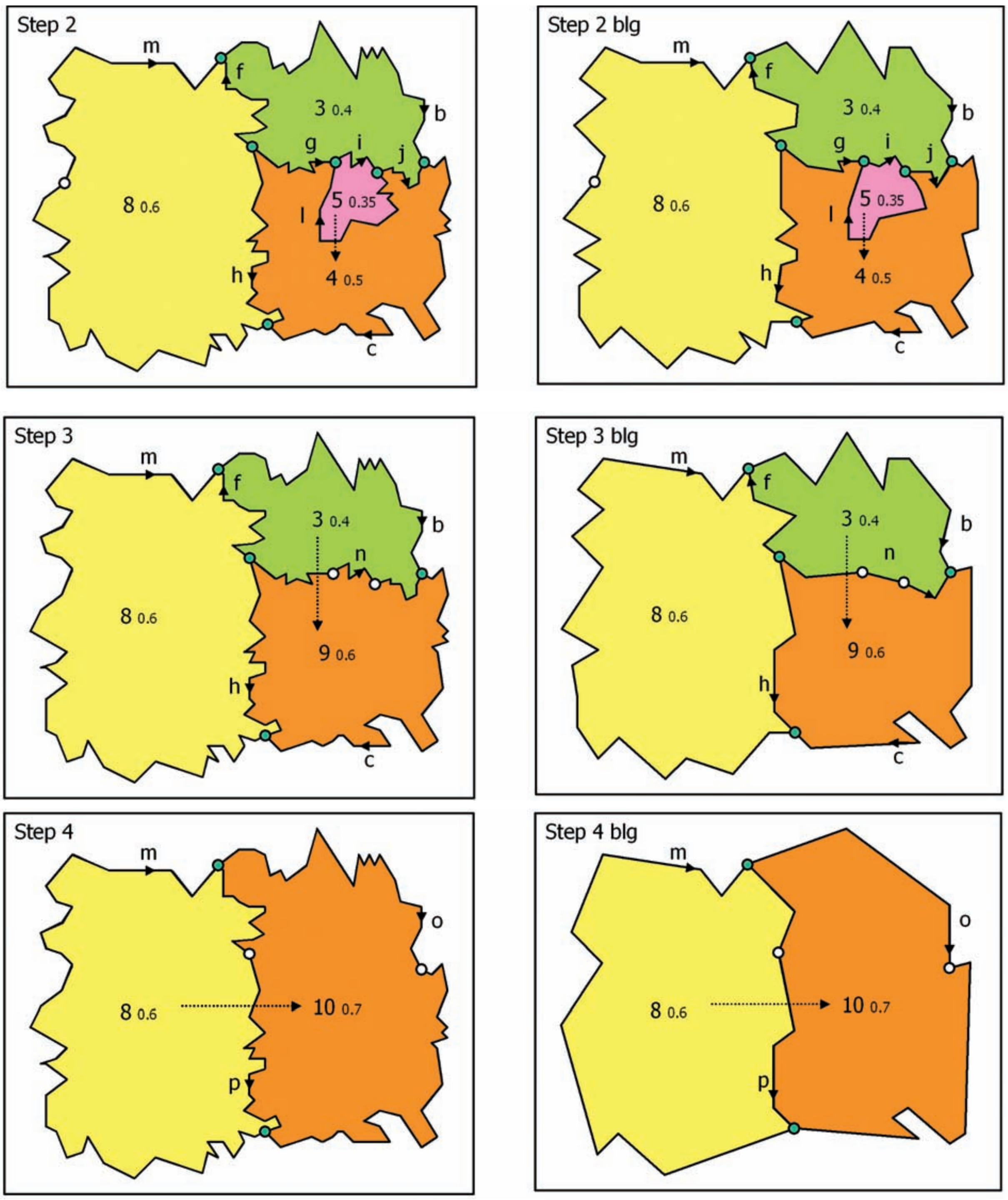


Figure 4, continued on page 337.

The first drawback can be avoided by using a winged edge topology (Baumgart 1975; van Oosterom 1997; van Oosterom and Lemmen 2001) together with aboxes for the edges. An abox (area box) of an edge is the union of the bboxes (bounding boxes) of the left and right faces. Note that an edge also has its own bbox (bounding box), which is always contained in its abox. Selecting the faces based on their bbox and the edges based on their abox (both at the appropriate importance level) enables the creation of all polygons by following the topology references (from face to edge and from edge to edge) without any geometric computations. This enables easy

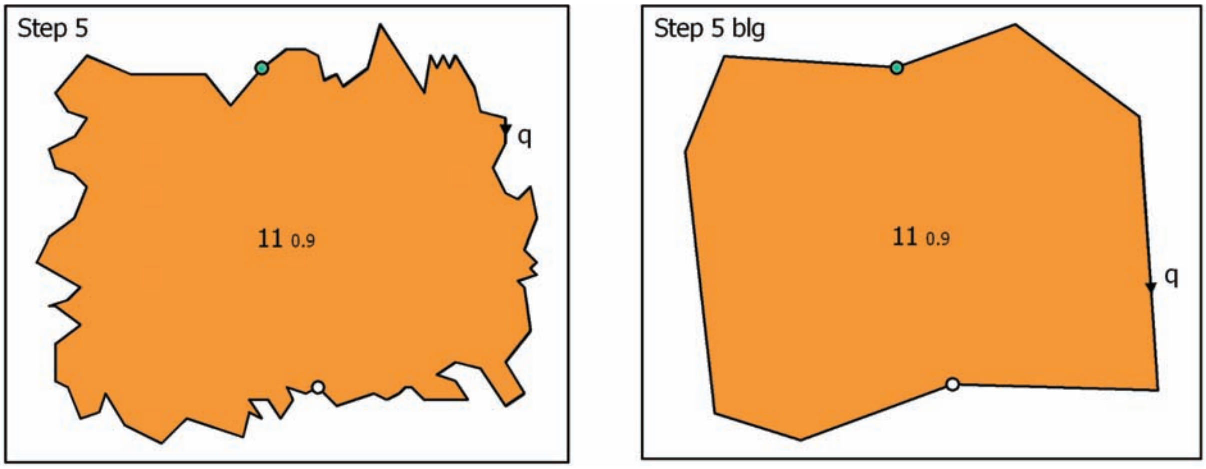


Figure 4. Generalization example in five steps, from detailed to course. The left side shows the effect of merging faces, while the right side shows the effect of also simplifying the boundaries via the BLG tree. Note that nodes are depicted in green/blue and removed nodes are shown for one next step only in white.

client implementation. In the next section the structure will be presented which also avoids the redundant storage of geometry at different levels of detail (in the edges).

Topological GAP-face Tree and GAP-edge Forest

Throughout this paper, one example is used to illustrate the new topological GAP tree. Different subsections (and different figures) will explain additional details related to this structure. Figure 4 shows the scene: the edges are at the top left, and the faces (each with a color according to their classification) are at the top right. The faces are assigned a number id and a computed importance value (shown in a smaller font). The edges are assigned a letter id (for illustration purposes; numbers are used in a normal implementation). Note that all edges are directed, as is normally the case in a topological structure; i.e., there is a left- and right-hand side.

Faces in the Topological GAP Tree

In the example above, in every step the least important object is removed and its area is assigned to the most compatible neighbor (as in the normal GAP tree). In the first step of Figure 4 for the least important object, face 6 has been added to its most compatible neighbor, face 2. This process is continued until there is only one face left. A slight difference with the original GAP tree is that a new id is assigned to the enlarged, more important face. The enlarged version of face 2 (augmented by face 6) becomes face 7, and faces 2 and 6 are not used at this detail level. However, the enlarged face keeps its classification as indicated via the color of the faces. In our example, face 7 has the same classification as face 6 (and both are shown in yellow), but the importance of this face is recomputed: as it becomes larger, the importance increases from 0.4 to 0.5. In the next step, face 1 (the least important with importance value 0.3) is added to face 7 (the most compatible neighbor), and the result is face 8, with its importance increasing from 0.5 to 0.6. Then face 5 (importance 0.35) is added to face 4 (best neighbor), and the result is face 9 (with importance increasing from 0.5 to 0.6). This process continues until one big face is left; in our example this is face 11.

From the conceptual point of view the generalization process is the same and the original (face) GAP tree can easily be rewritten to the "new style" GAP tree for faces (Figure 5). The advantage of the new-style tree is that is it a binary tree (the original tree was n-ary). As in the original GAP tree, every node in the new-style tree contains an indication of the importance level of the corresponding face. This is not shown in the Figure 5, but it will become clear in Table 1 which gives all the attributes of the faces (and the edges). Unlike in the original GAP tree, however, the nodes contain no explicit polygons; only topological face information is given. This means that the edges are also stored so that the faces can refer to them.

Edges in the Topological GAP Tree

Vermeij et al. (2003) described the set of edges needed by the topological faces in the GAP tree. These edges have topological references to the faces left and right, as proposed by Vermeij et al. (2003). During the generalization process, when faces are merged, three different outcomes may occur:

- An edge is removed; e.g., edge 'e' in step 2;
- Two or three edges are merged in one edge; e.g., edges 'a' and 'd' merge into edge 'm' in step 2, or edges 'g', 'i', and 'j' merge into edge 'n' in step 3);
- Only edge references are changed; e.g., the reference to the right face of edge 'h' changes from 7 to 8 in step 2.

The parent–child relationships between the different (versions of the) edges are maintained; because, as it turns out, these relationship again form a tree structure (Figure 6). However, there is no single edge root (but there is a face root in the GAP-face tree), since some edges are removed (option 1) and they form a local root. Just like faces, the edges have an associated importance range, indicating when they are valid. Edge importance ranges are given in Figure 6 for GAP-edge trees as well as the GAP-edge forest. Note that the top edge node (in this case 'q') is not showing an upper importance value, while all other nodes have both a lower and a upper importance value associated with the edge.

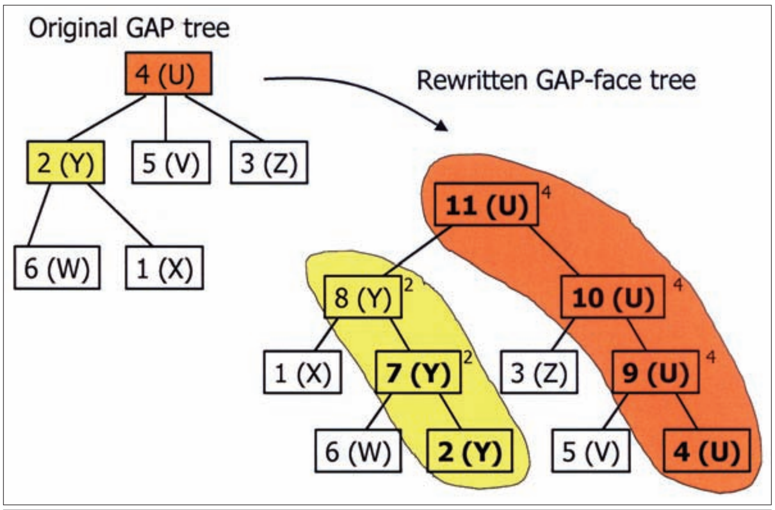


Figure 5. The classic GAP tree rewritten as the GAP-face tree (with a new object Id whenever a face changes and the old object Id appearing in a small font to the upper right of a node). The class is shown in brackets after the object Id.

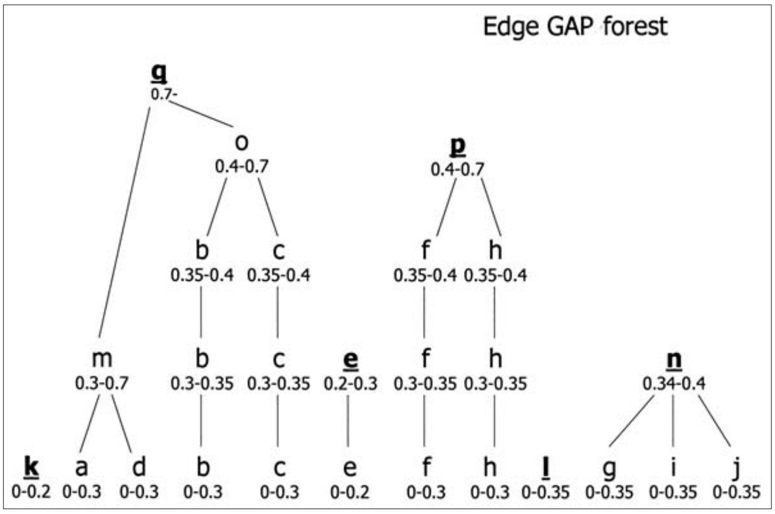


Figure 6. GAP-edge forest (with important ranges). Note that the edges shown in bold and the underlined letters k, q, e, p, l, and n are the roots of the different GAP-edge trees.

BLG Trees in the Topological GAP Tree

When edges are represented via BLG trees, the structure delivers an appropriate number of points per scale level, together with the related tolerance value. Note that the relationship between the tolerance value and scale is direct, e.g., at a given scale, one can use the size of a pixel of the display screen as the tolerance value. However, the relationship between importance and scale is less direct; as a rule of thumb, because "an optimal screen has a constant information density, one can keep on adding faces with a lower importance until the specified number of faces is reached; e.g., 1000. An alternative could be applying the Radical Law nf = naC √(Ma/Mf)x, where "nf is the number of objects shown at the derived scale, na is the number of objects shown on the source material, Ma is the scale denominator of the source map, and Mf is the scale denominator of the derived map (Töpfer and Pillewizer 1966). The exponent x depends on the symbol types (1 for point symbols, 2 for line symbols and 3 for area symbols) and C is a constant, frequently with a

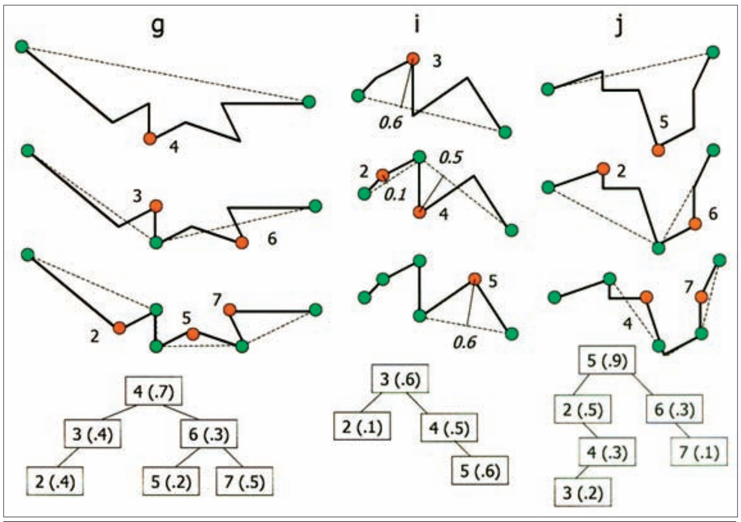


Figure 7. Three example BLG trees with edges g, I, and j. A node in the BLG tree contains a point (number) and a tolerance value (in brackets).

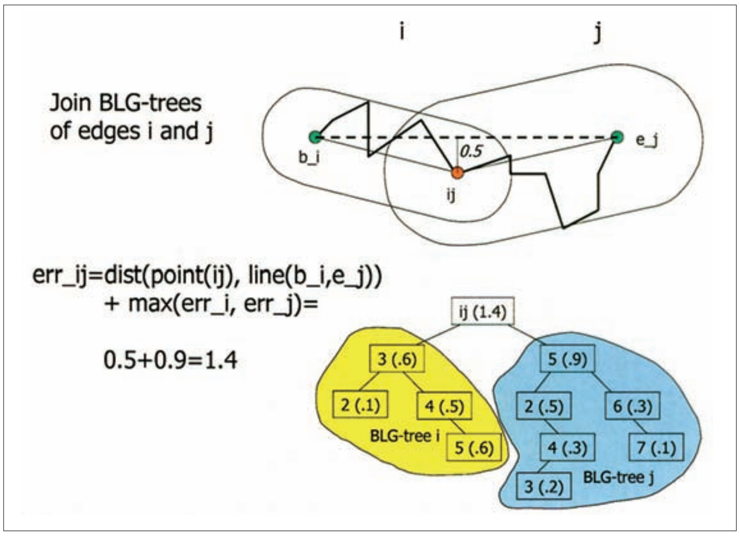


Figure 8. First step in merging the edges g, i, and j. The BLG trees of i and j are joined (the worst-case estimation for the new top level tolerance value is 1.4).

value of 1. Figure 7 shows three different edges of our scene with their corresponding BLG tree depicted below (nodes indicate point number and error values).

Standard BLG-tree structures are not exactly new, but they can be used satisfactorily within the edge GAP-forest structure to represent edges. As indicated above, due to the removal of edges, it is possible for some edges to merge into larger edges. Thus, instead of storing redundant edge polylines at different scales/importance levels, it was decided to join the BLG trees of the merging edges. To merge three edges, for instance, 'g', 'i' and 'j', two steps are required: first 'i' and 'j' are merged (see Figure 8), then edge 'g' is merged (see Figure 9). Note that only the top tolerance value is computed every time two BLG trees are merged. Also the worst-case estimation of the new top tolerance value 'err_ij', according to the formula given in (van Oosterom 1990; 1992) and reformulated in Figure 8 as *'err_ij = dist(point(ij), line(b_i, e_j) + max(err_i, err_j)'* only uses the top-level information of the two participating trees.

A small improvement, which keeps the structure of the merged BLG tree unaffected (i.e., the lower level BLG tree can be reused), is to compute the exact tolerance value ('err_ij_exact') of the new approximated line, which is less than or equal to the estimated worst case ('err_ij'). In Figure 8, this would be the distance from point 5 of edge 'j' to the dashed line. This tolerance value would be 1.1, which is less than the worst-case estimate of 1.4. The drawback is that one has to descend to the lower-level BLG tree to perform the computation (this may be a recursion), which, even though done only once during the creation of the structure may still be time-consuming. The advantage is that during the use of the structure (which probably happens more often than creation), a better estimate becomes available and a descend to the BLG tree is no longer needed. For example, assuming we need a tolerance of 1.2, then, with the worst case estimate, one has to descend to the two-child BLG tree (which will not happen with the computed tolerance). Conclusion: stick to the proposed structure of the simple merging of BLG trees but consider real computation of the top-level tolerance value.

Depending on the requested tolerance value, the (joined) BLG tree is traversed in order to produce the appropriate detail level. Note that this may imply that a point, which used to be associated with a node (at a high detail level, low importance value) may be also removed. This is needed if one performs extreme generalization; e.g., the highest detail data is at the level of ownership parcels, but the user would like to see municipalities and so he or she zoomes out. Supposing all original nodes had been used (as in Vermeij 2003), the result would still be too much detail presented (which also takes time to transfer from server to client).

Storage Structure of the GAP-face Tree and GAP-edge Forest

Now that the structure and creation process of the GAP-face tree and GAP-edge forest have been explained and illustrated with an example, the next step is the actual implementation of these structures. An important aspect is the storage structure; to enable its implementation, an object-relational model such as mainstream DBMSs, Oracle, DB2, Informix, Ingres, MySQL, PostgreSQL, can be used.

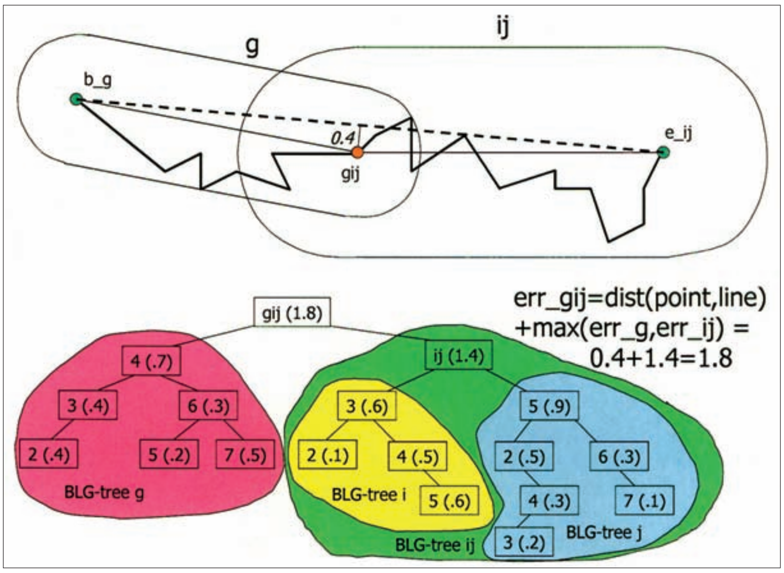


Figure 9. Second step in merging the edges g, i, and j. The BLG trees of ij and g are joined. Note that for BLG tree ij the worst-case tolerance value estimation of 1.4 was used and, therefore, the tolerance band does not touch the polyline in the middle as might have been expected.

Faces in the Topological GAP-tree Storage Structure

The storage structure of the GAP-faces is given in Table 1. Here are some notes:

- The column "step" is not stored. It was added for illustration purposes (and shown in italics), so as to link the steps of the creation process to the rows in the table;
- The column "ace_id" is the primary key in the table;
- The columns "imp_low" and "imp_high" indicate the importance range for which a given record is valid, and the column "imp_orig" indicates the original importance of the face (note that imp_orig >= imp_high. Take a look at face 2 with imp_orig=0.40; when merged with face 6 at the importance level of 0.20, the resulting face 7 has imp_orig=0.50);
- Where an area has one or more islands, the column "first_edges" does not only refer to the edge being part of the outer boundary (the first in the list of references; see face 2 in Table 1), but also to an edge that's part of the inner boundary (i.e., it is a variable length array of references);
- The edge references are labeled depending on the direction of the loop; i.e., clockwise loops designate outer boundaries and anti-clockwise loops designate inner boundaries. A plus sign (+) means the edge direction is correct for the loop, a minus sign (-) means that the edge direction has to be reversed;
- The polygon of a face can be reconstructed by following the references from the first_edges column in the winged edge structure, at the given importance range;
- The column "bbox" contains only symbolic values; the actual coordinate values will, of course, differ for each row in a real-life situation. Instead of storing the bbox, it could also be computed using a functions in the database;
- The GAP face-tree can be reconstructed by linking every child to its parent via the "pid" column (parent identifier); only the root, face 11, does not have a parent.

The first SQL view definition given below is for adding polygons to the "tgap_face" table. As reported by van Oosterom et al. 2002, the add-on action is based on the return_geometry function, which uses the edge table "under water." The second SQL view definition adds the bbox of the face (and other attributes, such as area and perimeter). Note that the results of the views are normally not explicitly stored (unless it is a materialized view for performance reasons).

```
create view tgap_face_v1 as
select f.face_id, f.imp_low, f.imp_high, f.imp_orig,
    f.first_edges, f.class, f.pid,
    return_polygon(f.face_id) shape
from tgap_face f;
create view tgap_face_v2 as
select f.face_id, f.imp_low, f.imp_high, f.imp_orig,
    f.first_edges, f.class, f.pid, f.shape, get_
    bbox(f.shape) bbox,
```

step	**face id**	imp low	imp high	imp orig	*first edges*	class	pid	*bbox*
0	1	0.00	0.30	0.30	+a	X	8	(xl,yl,xh,yh)
0	2	0.00	0.20	0.40	+d,-k	Y	7	(xl,yl,xh,yh)
0	3	0.00	0.40	0.40	+b	Z	10	(xl,yl,xh,yh)
0	4	0.00	0.35	0.50	+c	U	9	(xl,yl,xh,yh)
0	5	0.00	0.35	0.35	+i	V	9	(xl,yl,xh,yh)
0	6	0.00	0.20	0.20	+k	W	7	(xl,yl,xh,yh)
1	7	0.20	0.30	0.50	+d	Y	8	(xl,yl,xh,yh)
2	8	0.30	0.40	0.60	+m	Y	10	(xl,yl,xh,yh)
3	9	0.35	0.60	0.60	+c	U	11	(xl,yl,xh,yh)
4	10	0.40	0.60	0.70	+o	U	11	(xl,yl,xh,yh)
5	11	0.60	-	0.90	+q	U	-	(xl,yl,xh,yh)

Table 1. The 'tgap_face' with the topological multi-scale face information representing the GAP face-tree.

```
    get_area(f.shape) area, get_perimeter(f.shape)
   perimeter
from tgap_face_v1 f;
```

The SQL "create view" statement specifies the name of the new view (e.g., tgap_face_v1) and then the SQL select statement which defines the view is given. In this case nearly all attributes are copied straight from the original table (tgap_face) and a new attribute with the name "shape" is added and computed via the return_geometry function. Something similar happens with the second view (tgap_face_v2), which adds the attributes bbox, area, and perimeter to the first view, tgap_face_v1.

Edges in the Topological GAP Tree Storage Structure

Table 2 describes the edges and faces of the GAP-edge forest. Again, a number of notes:

- The columns "edge_id" and "imp_low" together form the primary key in this table. The ids of the merged edges are shown in bold. New importance range versions of edges that have the same id are in normal fonts. For example, there are two versions of edge d and they are identified by the primary key pairs (d, 0.00) and (d, 0.20);
- Again, the column "step" is not stored, but only added for illustration purpose and shown in italics;
- The references to faces left and right are given in the columns "face_left" and "face_right" (changed references after step 1 for new versions of existing edges are in bold);
- The winged edge references to edges are given in the columns "edge_fl" (first left), "edge_fr" (first right), "edge_ll" (last left), and "edge_lr" (last right). The changed references are again shown in bold. The references are signed according to the direction (same interpretation as in the GAP-face tree table);
- The GAP-edge trees (in the forest) can be reconstructed by linking every child to its parent via the "pid" column ("interesting" parents, i.e., versions created by merging other edges, are shown in bold). Note that there are now several roots —k, l, e, n, p, and q—i.e., edges that do not have a parent;
- The column "abox" contains symbolic values. Actually, this column does not need to be physically present and could be implemented as a view, by computing the union of the bboxes of the left- and right-hand side faces. This column is also shown in italics;
- Further, it was decided to store the source BLG trees (related to the edges at the highest detail) in a separate table, as the different rows may refer to same BLG tree (and, therefore, they are shown in italics).

If one does not change the orientation of the faces (polygons), then only 2 of the 4 edge-to-edge references are really needed: edge_lr and edge_fl (assuming clockwise outer boundary loops). Removing the other two edge-to-edge references does not only save storage space, but it may also save a number of rows. In our example in Table 2 the edge rows (c, 0.30) and (f, 0.35) could be omitted, because the only difference with their parent version is in the columns edge_fr or edge_ll.

One could consider dropping all edge-to-edge references. This would result in saving the storage cost of two more columns (edge_lr and edge_fl) and that of those rows that only have changes in these two colums. In our example, the edge rows (b, 0.30), (b, 0.35), and (m, 0.40) would be removed in addition to the previous removal of the edge rows (c, 0.30) and (f, 0.35). The price one would have to pay for this saving though is that client will have to do more searching in order to connect the proper edges after orienting them in the proper direction, instead of just following the given references. Note that connecting the edges can be considered to be a purely non-geometric operation as long as the end/begin points of the edges that are supposed to meet are exactly equal (and these can be used as a kind of topology node/point identifiers).

In the case of multiple loops, one has to find out which of these loops is the outer loop. This can be accomplished by using the first_edges column from the tgap_face table. However, if the edge

step	**edge**	**imp**	imp	face		edge				pid	*abox*	*BLG-tree*
	id	**low**	high	left	right	fl	*fr*	*ll*	lr			(blg_id)
0	a	0.00	0.30	0	1	-d	-e	+b	-f	**m**	*Union(l,r)*	*tree+xy..*
0	b	0.00	0.30	0	3	-a	-f	+c	-j	b	*Union(l,r)*	*tree+xy..*
0	c	0.00	0.30	0	4	-b	-j	+d	-h	c	*Union(l,r)*	*tree+xy..*
0	d	0.00	0.20	0	2	-c	-h	+a	-e	d	*Union(l,r)*	*tree+xy..*
0	e	0.00	0.20	2	1	+h	+f	-d	+a	e	*Union(l,r)*	*tree+xy..*
0	f	0.00	0.30	1	3	+e	+g	-a	+b	f	*Union(l,r)*	*tree+xy..*
0	g	0.00	0.35	3	4	+f	+h	+i	-l	**n**	*Union(l,r)*	*tree+xy..*
0	h	0.00	0.20	4	2	+g	+e	-c	+d	h	*Union(l,r)*	*tree+xy..*
0	i	0.00	0.35	3	5	-g	-l	+j	+l	**n**	*Union(l,r)*	*tree+xy..*
0	j	0.00	0.35	3	4	-i	+l	-b	+c	**n**	*Union(l,r)*	*tree+xy..*
0	k	0.00	0.20	2	6	-k	-k	+k	+k	-	*Union(l,r)*	*tree+xy..*
0	l	0.00	0.35	4	5	+j	-i	-g	+i	-	*Union(l,r)*	*tree+xy..*
1	d	0.20	0.30	0	**7**	-c	-h	+a	-e	**m**	*old*	*old*
1	e	0.20	0.30	**7**	1	+h	+f	-d	+a	-	*old*	*old*
1	h	0.20	0.30	4	**7**	+g	+e	-c	+d	h	*old*	*old*
2	**m**	0.30	0.40	**0**	**8**	**-c**	**-h**	**+b**	**-f**	m	*Union(l,r)*	*BLG a+d*
2	b	0.30	0.35	0	3	**-m**	-f	+c	-j	b	*old*	*old*
2	c	0.30	0.35	0	4	-b	-j	**+m**	-h	c	*old*	*old*
2	f	0.30	0.35	**8**	3	**+h**	+g	**-m**	+b	f	*Union(l,r)*	*old*
2	h	0.30	0.35	4	**8**	+g	**+f**	-c	**+m**	h	*Union(l,r)*	*old*
3	**n**	0.35	0.40	**3**	**9**	**+f**	**+h**	**-b**	**+c**	-	*Union(l,r)*	*BLG g+i+j*
3	b	0.35	0.40	0	3	-m	-f	+c	**-n**	**o**	*old*	*old*
3	c	0.35	0.40	0	**9**	-b	**-n**	+m	-h	**o**	*Union(l,r)*	*old*
3	f	0.35	0.40	8	3	+h	**+n**	-m	+b	**p**	*old*	*old*
3	h	0.35	0.40	**9**	8	**+n**	+f	-c	+m	**p**	*Union(l,r)*	*old*
4	**p**	0.40	0.60	**10**	8	**+o**	**-m**	**-o**	**+m**	-	*Union(l,r)*	*BLG f+h*
4	**o**	0.40	0.60	**0**	**10**	**-m**	**+p**	**+m**	**-p**	**q**	*Union(l,r)*	*BLG b+c*
4	m	0.40	0.60	0	8	**-o**	**-p**	**+o**	**+p**	**q**	*old*	*old*
5	**q**	0.60	-	**0**	**11**	**-q**	**-q**	**+q**	**+q**	-	*Union(l,r)*	*BLG m+o*

Table 2. The 'tgap_edge' with the topological multi-scale edge information representing the GAP-edge forest.

reference column is also dropped (because of storage efficiency reasons), a modest geometric computation may be needed (e.g., to compute the area enclosed by the loops; the loop with the largest area is the outer boundary), or the face_left and face_right references could be explicitly marked with the minus sign, indicating that the current edges are part of the inner boundary. Keeping the first_edges colum in the tgap_face table and at least two edge references (edge_lr and edge_fl) in the tgap_edge table may thus be advisable.

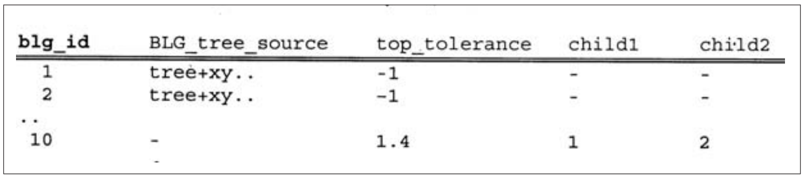
The SQL view definition for adding the abox to the tgap_edge table is given below (note that this view uses the bboxes of faces computed in another view, tgap_face_v2):

```
create view tgap_edge_v1 as
  select e.edge_id, e.imp_low, e.imp_high,
  e.face_left, e.face_right,
    e.edge_fl, e.edge_fr, e.edge_ll, e.edge_lr,
    e.pid, e.blg_id, union(l.bbox, r.bbox) abox
from tgap_edge e, tgap_face_v2 l, tgap_face_v2 r
where e.face_left=l.face_id and e.face_
right=r.face_id;
```

The structure of the SQL select statement used in the definition of the view tgap_edge_v1 is now a little different: instead of just using one table (or view) and adding something (after computation), the information is obtained from several tables or views (tgap_edge and tgap_face_v2). Actually, the tgap_face_v2 view is used in two different places, the left and right side ('l' and 'r' aliase). The "where" clause specifies the conditions under which the records from different views have been combined; e.g., e.face_left=l.face_id. In SQL terms, this kind of operation is called a join-of-tables (or views). The view tgap_edge_v1 copies the old edge attributes and adds the attribute abox based on the function "union" and the bbox input from the left and right tgap_face_v2 views.

The Role of the BLG Tree in the Topological GAP-tree Storage Structure

The model uses DBMSs spatial extensions and follows OGC's Simple Feature Specification for SQL (OGC 1999) for spatial data types, such as the bbox and the abox. Because the BLG-tree data type for variable-scale polylines is not available, one has to implement this data type. This has been done in the Postgres context (Oosterom and Schenkelaars 1995) and in the Oracle special context (Vermeij 2003). Assume for the time being that we have a BLG-tree data type for our implementation. The table "tgap_blg" stores all the sources of BLG trees (tree structure, tolerance values, point coordinates) related to the edges at the most detailed level and, symbolically, also all the merged BLG trees. The merged BLG trees can be rewritten to merge pairs of BLG trees (see Figures 8 and 9) and, therefore, we only need two columns with references "child1" and "child2" and the tolerance value. Table 3 shows the header of the table with some sample rows (rows 1 and 2 contain two-source BLG trees, and row 10 cotains a merged BLG tree).



blg_id	BLG_tree_source	top_tolerance	child1	child2
1	tree+xy..	-1	-	-
2	tree+xy..	-1	-	-
..				
10	-	1.4	1	2

Table 3. The 'tgap_blg' with the BLG trees of the edges in the GAP-edge forest.

The SQL view definition that hides the differences between the source BLG trees and merged BLG trees is given below:

```
create view tgap_blg_v1 as (
   select b.blg_id, b.BLG_tree_source BLG_tree
   from tgap_blg b
   where b.top_tolerance = -1)
union all (
   select b.blg_id, merge_BLG(b.top_tolerance,
   b.child1,b.child2) BLG_tree
   from tgap_blg b
   where b.top_tolerance <> -1);
```

The view "tgap_blg_v1" has a different structure than the previously presented view, as it is not based on one SQL select statement but two. The first SQL select statement selects the source (or leaf) BLG trees (based on the condition b.top_tolerance = -1 in the "where" clause) and simply copies all attributes. The second SQL statement in this view takes care of merged, non-leaf, BLG trees. The result of the two SQL select statements is glued together with the SQL "union all" operation. The SQL view definition to combine edges and their corresponding BLG trees is:

```
create view tgap_edge_v2 as
select e.edge_id, e.imp_low, e.imp_high, e.face_
   left, e.face_right, e.edge_fl, e.edge_fr,
   e.edge_ll, e.edge_lr, e.pid, e.abox, b.BLG_tree
from tgap_edge_v1 e, tgap_blg_v1 b
where e.blg_id=b.blg_id;
```

The view "tgap_edge_v2" adds the BLG tree to the edge information via a join of the previously defined views tgap_edge_v1 and tgap_blg_v1.

The Role of the Reactive Tree in the Topological GAP-tree Storage Structure

Available spatial indexing (and clustering) should be used to efficiently select the relevant rows from the tables; i.e., via a query search rectangle and specified importance value. The Reactive tree is designed for indexing spatial objects with importance values, but it is not generally available. Although in an extensible DBMS, one can add its own implementation of new index structures, this operation is far from trivial. This notwithstanding, it has been done for the Reactive tree in Postgres (van Oosterom and Schenkelaars 1995). In practice, a pseudo Reactive tree could be used, which is based on the 3D R-tree in which the third dimension is used for the valid importance range (and the first two dimensions are from the bbox and abox in the face and edge table, respectively). The advantage is that one does not have to implement an own indexing structure and can simply reuse the standard 3D R-tree for 2D objects with their corresponding importance ranges. So, the 3D block to be indexed becomes (xl,yl,imp_low,xh,yh,imp_high). The 3D block does not have to be physically present; as long as it can be returned by a function with the box and importance range as inputs, then the computed 3D block can be used in a functional index, such as available in Oracle 9i and higher (van Oosterom et al. 2002). This applies to both the face and edge table. For example, the functional index on the face table is:

```
create index tgap_face_idx on
   tgap_face(compute_3D_block(get_bbox(return_
   polygon(face_id)), get_imp_range(imp_low,
   imp_high)))
indextype is 3D_rtree;
```

Note that the edge table is joined with the table containing the BLG trees, and, as a result, one can use views for easy access. The result of the two queries: 1) give all faces in importance range Y and with bbox overlapping, given rectangle X' and 2) give all edges (with points up to tolerance value) in importance range Y and abox overlapping, given rectangle X' can, due to indexing (and clustering of the 3D R-tree), be obtained very quickly and sent to the (web) client. The web client has all the edges—at the required level of detail—needed for forming the GAP-tree polygons corresponding to faces (partially) overlapping the given search rectangle X (probably related to the current window on the screen of the user).

Server–client Set-up and Progressive Refinement

Data received by a client could be progressively refined as follows. The server starts by sending the most important nodes in GAP face-tree/edge-forest (including top levels of associated edge BLG trees) in a certain search rectangle. The client builds a partial copy of GAP/BLG-structure, which can then be used to display the coarse impression of the data. Every (x) second(s) this structure is displayed, and the polygons are shown at the then available resolution on the screen. The server keeps on sending more data and the GAP/BLG-structure at the client side is growing, such that the next time it is displayed with more detail. The possible criteria for ending this refinement could be:

- 1000 objects (meaningful information density);
- Required importance level has been reached (with associated error tolerance value); or
- The user interrupts the client.

Conclusion

Summary of Main Results

This is the first time ever that a non-redundant geometry, variable-scale data structure has been presented. The previous versions of the GAP tree had some geometry redundancy, primarily between the polygons at a given scale and/or between the scales. The key to the solution presented in this paper was applying a full topological structure, though this is (far) more complicated than topological structures designed for the traditional single-scale data sets. The topological GAP tree is very well suited for a web environment—client requirements are relatively low (no geometric processing of the data at the client side) and progressive refinement of vector data is supported (allowing quick feedback to the user).

The values that remain crucial for the quality of GAP-tree generalization are the importance value of the involved feature classes (and importance function) and the compatibility values between two different feature classes (and compatibility function). More research is needed in this area to automatically obtain good generalization results for real-world data.

Implementation, Tuning, and Performance Testing

Future work will include the actual implementation of the presented concept and further testing with large data sets (millions of rows). Of course, both the data loading/creation of the GAP-face tree and GAP-edge forest (with their BLG trees) and the use of the information must be tested; perhaps even with a varying number of edge-edge references—0, 2 (edge_lr and edge_fl), or 4 (edge_fl, edge_fr, edge_ll, and edge_lr), as mentioned earlier on. Next, the performance of this solution has to be compared to an alternative multi-scale representation (which has less freedom in scale than our true variable scale structure) in a desktop GIS environment (i.e., GIS front-end relatively close to the geo-DBMS server). The second set of performance tests would be within a distributed set-up, with the server and web-based clients taking advantage of progressive refinement. The Douglas-Peucker algorithm used in the BLG tree could in theory generate topological errors such as (self) intersecting lines. The GAP-face tree and GAP-edge forest structures would not have a problem with these (self) intersections, but, if possible, these (self) intersections should be avoided. However, as de Berg et al. (1998) mention avoiding such intersections is not easy.

Possible Further Enhancements of the Topological GAP Tree

Future research aimed at further improving the functionality of the GAP tree could include the following:

- Data editing (at the most detailed level), dynamic structures, only local updates, and some propagation of the changes to the higher importance values;

- Removing the last piece of geometric redundancy by introducing a "tgap_node" table. Geometric redundancy between neighbor faces (and also between different generalized representations) was removed by introducing the tgap_edge table; however, the start and the end points of edges are still redundantly stored in all the touching edges (in their corresponding BLG trees). If the start and end points are removed and replaced by two references in the tgap_edge table (or even better, in the tgap_blg table in order to avoid storage of redundant references)—start_node and end_node—then the last piece of geometric redundancy will be removed. Additionally, the removal would fit well with the concept of the BLG tree (see Figure 7). The start and end points are treated differently from the intermediate points, which are all stored in the normal nodes of the BLG tree. It does not seam to be useful to have references from the nodes back to the edges (or faces). The tgap_node table would have the following attributes: node_id, imp_low, imp_high, and location. The primary key will be node_id (without imp_low), since for every node there will only be one version. The drawback of the solution, besides the introduction of a new table, is the introduction of additional references, which do take some space. Via a function (which can be used in a view), the start and end points of an edge can be "glued" to the intermediate points and the return value could again be a normal polyline.
- Including such non-area objects as point and line objects. A point object can be stored (quite independent of the other structures) in its own table where every row has also an importance value range (the Reactive tree can be used for spatial/importance indexing). A line object with reference to its BLG tree can be stored in its own table and indexed with a Reactive tree (based on importance range and location). Perhaps it may also be useful to combine two less important lines into one more important line. The question is how to detect that this extension is needed. Once combined, the lines can be merged similar to joins of merged area boundaries, and a BLG tree is computed. It should be noted here that the point and line objects are independent, not related to the area object and the tables used to represent them, which simplifies this extension. It gets more complicated when the different feature types have to be related to each other and the topological relationships between point, line, and area objects have to be maintained. Such an extension requires further investigation.
- Changing from area to line or point representation for a given object. This step is similar to that of the normal GAP tree when removing an area, except that it introduces line or point features which are then related/linked to the previous area representation. For roads this may be better than enlarging with strips as suggested by van Putten and van Oosterom 1998.

ACKNOWLEDGEMENTS

This paper is the result of the research program "Sustainable Urban Areas" carried out by Delft University of Technology. Thanks to Wilko Quak, Edward Verbree, Maarten Vermeij, and Rod Thompson for a critical review of the draft version of this paper. Further, the constructive comments of the CaGIS reviewers are greatly appreciated. Their comments were of great help, but all (remaining) errors are the sole fault of the author.

REFERENCES

Ai, T., and P. van Oosterom. 2002. GAP-tree extensions based on skeletons. In: D. Richardson and P. van Oosterom (eds), *Advances in Spatial Data Handling*, 10th International Symposium on Spatial Data Handling, Ottawa, Canada, 9-12 July 2002. Berlin, Germany: Springer-Verlag. pp. 501–13.

Ballard, D. 1981. Strip trees: A hierarchical representation for curves. *Communication of the Association for Computing Machinery* 14: 310-21.

Bertolotto, M., and M.J. Egenhofer. 2001. Progressive transmission of vector map data over the World Wide Web. *GeoInformatica* 5(4): 345-73.

Bregt, A., and J. Bulens. 1996. Application-oriented generalization of area objects. In: *Methods for the Generalization of Geo-Databases.* Netherlands Geodetic Commission, Delft, The Netherlands. pp. 57-64.

Baumgart, B.G. 1975. A polyhedron representation for computer vision. In: *Proc. AFIPS National Computer Conference*, Vol. 44, pp. 589-96.

Buttenfield, B.P. 2002. Transmitting vector geospatial data across the Internet. In: M. J. Egenhofer, and D. M. Mark (eds), Proceedings, GIScience 2002. *Lecture Notes in Computer Science* 2478: 51-64. Berlin, Germany: Springer Verlag.

de Berg, M.T., van M. J. Kreveld, M.J., and S. Schirra. 1998. Topologically correct subdivision simplification using the bandwidth criterion. Cartography and Geographic Information Systems 25(4): 243-57.

DMA (Defense Mapping Agency). 1986. Product specifications for digital feature analysis data (DFAD): Level 1 and level 2. *Technical report*, DM, Aerospace Center, St. Louis, Mo.

Douglas, D.H., and T. K. Peucker. 1973. Algorithms for the reduction of the number of points required to represent a line or its caricature. *The Canadian Cartographer* 10(2): 112–22.

Günther, O. 1988. *Efficient structures for geometric data management.* Number 337 in Lecture Notes in Computer Science. Springer-Verlag, Berlin.

Guttman, A. 1984. R-trees: A dynamic index structure for spatial searching. *ACM SIGMOD* 13:47-57.

Hildebrandt, J., M. Owen, and R. Hollamby. 2000. CLUSTER RAPTOR: Dynamic geospatial imagery visualisation using backend repositories. In: *Proceedings of the 5th International Command and Control Research and Technology Symposium (ICCRTS).*

Jones, C.B. and Abraham, I.M. 1986. Design considerations for a scale-independent cartographic database. In *Proceedings 2nd International Symposium on Spatial Data Handling*, Seattle, 348-398.

Jones, C. B., A. I. Abdelmoty, M.E. Lonergan, P. M. van der Poorten, and S. Zhou. 2000. Multi-scale spatial database design for online generalisation. In: *Proceedings, 9th International Symposium on Spatial Data Handling*, Beijing, China, Sec. 7b, 34-44.

Lazaridis, I., and S. Mehrotra. 2001. Progressive approximate aggregate queries with a multi-resolution tree structure. In: *International Conference on Management of Data Archive, Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, Santa Barbara, California, United States. pp. 401–12.

OGC (OpenGIS). 1999. OpenGIS Simple Features Specification for SQL, Revision 1.1. OpenGIS Project Document 99-049.

van Oosterom, P. 1989. A reactive data structure for geographic information systems. In: *Auto-Carto 9*, Baltimore, Maryland. pp. 665-74.

van Oosterom, P. 1990. Reactive data structures for geographic information systems. PhD thesis, Department of Computer Science, Leiden University, The Netherlands.

van Oosterom, P. 1992. A storage structure for a multi-scale Database: The Reactive-tree. *International Journal, Computers, Environment and Urban Systems* 16(3): 239-47.

van Oosterom, P. 1993. The GAP-tree, an approach to "on-the-fly" map generalization of an area partitioning. In: J.C. Müller, J.P. Lagrange, and R. Weibel (eds), *GIS and Generalization, Methodology and Practice.* London, U.K.: Taylor & Francis. ch. 9, pp. 120-32. Presented at the GISDATA Specialist Meeting on Generalization, Compienge, France, 15-19 December 1993.

van Oosterom, P. 1994. *Reactive data structures for geographic information systems*. Oxford, U.K.: Oxford University Press.

van Oosterom, P., and V. Schenkelaars. 1995. The development of an interactive multi-scale GIS. *International Journal of Geographical Information Systems* 9(5): 489-507.

van Oosterom, P. 1997. Maintaining consistent topology including historical data in a large spatial database. In: *Auto-Carto 13*, Seattle WA, 8-10 April 1997. pp. 327-36.

van Oosterom, P., and C. Lemmen. 2001. Spatial data-management on a very large cadastral database. *Computers Environment and Urban Systems* 25(4-5): 509-28.

van Oosterom, P., J. Stoter, W. Quak, and S. Zlatanova. 2002. The balance between geometry and topology. In: *Advances in Spatial Data Handling, 10th International Symposium on Spatial Data Handling*, Berlin, Germany, 2002. pp. 209-24.

van Putten, J., and P. van Oosterom. 1998. New results with generalised area partitionings. In: *8th International Symposium on Spatial Data Handling*, Vancouver, International Geographical Union. pp. 485-95.

Rosenbaum, R., and H. Schumann. 2004. Remote raster image browsing based on fast content reduction for mobile environments. In: T. Chambel, N. Correia, J. Jorge, and Z. Pan (eds), *Eurographics Multimedia Workshop*, Nanjing, China. pp 13-9.

Samet, H. 1984. The quadtree and related hierarchical data structures. *ACM Computing Surveys archive* 16(2): 187–260.

van Smaalen, J.W.N. 1996. A hierarchic rule model for geographic information abstraction. In: *Proceedings, SDH'96*, Delft, The Netherlands. pp. 4b.31.

van Smaalen, J.W.M. 2003. Automated aggregation of geographic objects—A new approach to the conceptual generalisation of geographic databases. PhD thesis, Wageningen University, The Netherlands.

Töpfer, F., and W. Pillewizer. 1966. The principles of selection. *Cartographic Journal* 3: 10-16.

Vermeij, M.J. 2003. Development of a topological data structure for on-the-fly map generalization. Geodetic Engineering, MSc thesis, Delft University of Technology, June 2003, Delft, The Netherlands.

Vermeij, M., P. van Oosterom, W. Quak, and T. Tijssen. 2003. Storing and using scale-less topological data efficiently in a client-server DBMS environment. In: *Proceedings of the 7th International Conference on GeoComputation*, University of Southampton, Southampton, UK, 8-10 September 2003.

Zhou, X., S. Prasher, S. Sun, and K. Xu. 2004. Multiresolution spatial databases: Making web-based spatial applications faster. In: Jeffrey Xu Yu, Xuemin Lin, Hongjun Lu, et al., Proceedings, The Sixth Asia Pacific Web Conference (APWeb'04), 14-17 April, 2004, Hangzhou, China. *Lecture Notes in Computer Science* 3007: 36-47.