Server side technology and interface for client

The tGAP (topological Generalized Area Partitioning) structure is a collection of data structures that enable generalisation of spatial data. These structures store results of a generalisation process, and allow selection of features to be shown for any required level of detail (LoD), performing in that way an on-fly map generalisation by feature selection in tGAP. The generalization process reduces the level of detail by merging unimportant features to more important features (see Figure 1). Data forming a partition of space is considered. Results of the generalisation process are stored in the tGAP structure. It stores the geometry only for features belonging to the highest level of detail. As we work with a topological model for storing spatial data, geometry is indeed stored only for edges forming boundaries of area features. References for the relation between a face, i.e. an area feature, and its boundary edges (typical references of a topological model) are stored in the tGAP structure. Specific references stored from the tGAP structure are those between highest LoD features, and features created during the generalisation. For area features created from merging, tGAP structure stores references to the merged features. Each feature is associated with an importance range, which is used for selecting the right features for a required LoD. A relation is established between importance values and (LoD, which is compatible and translated to) scale of a map[1].

Merging of less important faces to more important faces is based on importance values associated to each face. Figure 1 illustrates the generalization process for the map partition shown in 'Step 0'. The other maps in Figure 1 show the result of the generalization in steps, and are labelled according to that. The result of each step is a (map) partition. A map[2] is a collection of faces, and each face is constructed by the set of edges that form its boundary. The collection of faces that should be visible at a certain scale determines the collection of edges that should be visible, namely edges that are in the boundary of at least one of these faces. There is a last issue in the generalization process: boundary edges get simplified as the level of detail decreases. This can be also seen in Figure 1.

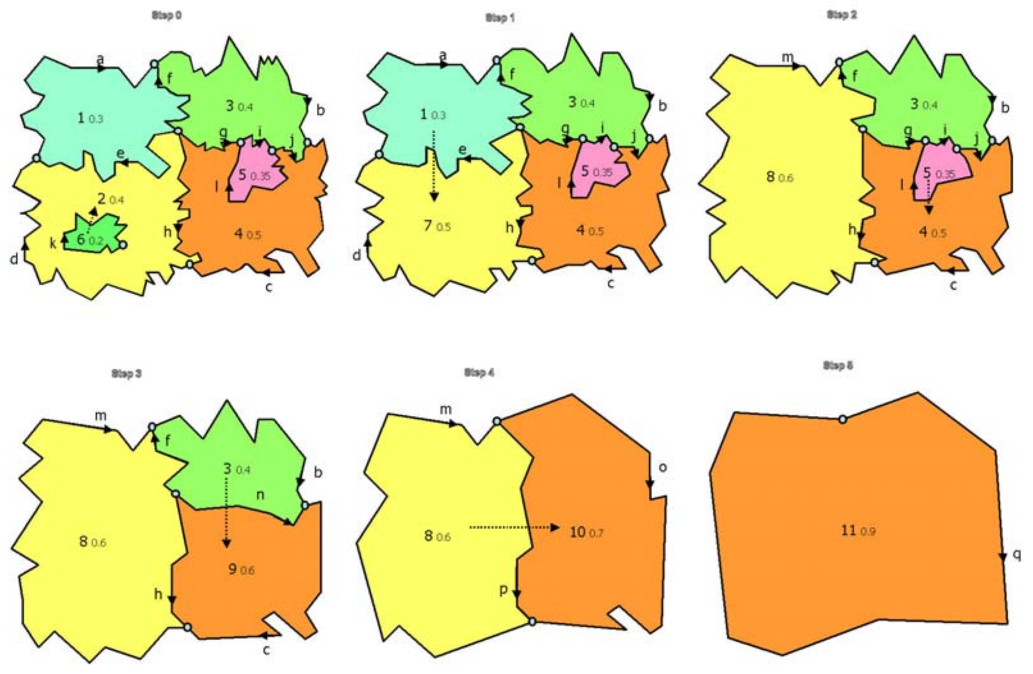

Figure 1. Merging of faces based on importance values. Different colours show different classes. Faces are numbered, and edges are labelled with letters. The subscript to a face number is its importance value.

[1] Considering this relation, we use the terms LoD and map scale interchangeably.

[2] For simplicity we will use the term 'map' throughout the text, meaning a map partition.

To capture the generalization process we need to keep track of the merging of faces in each step, how this is reflected to the boundary edges, and the simplification of edges. The data structures forming the tGAP take care of these three issues. The tGAP structure consists of a face tree holding the hierarchy of faces formed by merging, an edge forest that holds the corresponding relations between boundary edges, and BLG (Binary Line Generalization) trees, one tree for each edge that holds information about edge simplification.

The coming sections explain the tGAP structure and its implementation. Generalization process shown in Figure 1 is used for illustration. Section 1 describes how the tGAP is filled from the merging; Section 2 explains how tGAP is used to select the right features for a given importance level (to be translated to map scale). The implementation of tGAP structure in Oracle Spatial is explained in Section 3. Section 4 contains ideas about progressive transfer and visualisation on the client side.

1. Creating the tGAP structure

The tGAP structure consists of a face tree, an edge forest, and BLG trees. Building of each structure is treated separately in the coming sections.

Face tree

Generalization is performed in steps. Each step merges two existing faces to a new face. The merged faces are replaced by the new face, which continues further in the merging process. The new face and the merged faces have a parent-child relation. The process ends when only one face is left. The hierarchy of faces created by this process is a binary tree. Figure 2 shows this hierarchy – the face tree created by the generalisation process of Figure 1. Leaf nodes in the tree are the original faces, i.e. faces at the highest LoD. The root of the tree is the complete area of the map, union of all the original faces. Faces created during the generalization process form the other nodes of the tree.

Figure 1 illustrates the 6 steps for the generalization of the map shown in Step 0. A dashed arrow shows the least important face and its most compatible neighbour (where the arrow is headed). In the original map, Step 0, face 6 has the lowest importance value, 0.2, and face 2 is its only neighbour, therefore the most compatible face. In step 1 the two faces are merged into a new face, labelled 7. The two faces labelled 6 and 2 cease existing at the importance level 0.2, whereas face 7 starts existing at this importance level. The importance value of face 7 is calculated, 0.5, and the face is considered for the next step in the merging process. The process continues until all is merged to one face, labelled 11. Figure 2 shows the hierarchy of faces created from this generalization process. A node in the tree is a face. Each node is associated with the importance range for which the node exists – the values below the node. In the right side of the tree are shown the steps performed to create the tree, each step is associated with its importance value. Nodes in the tree are levelled with the step in which they stop existing.

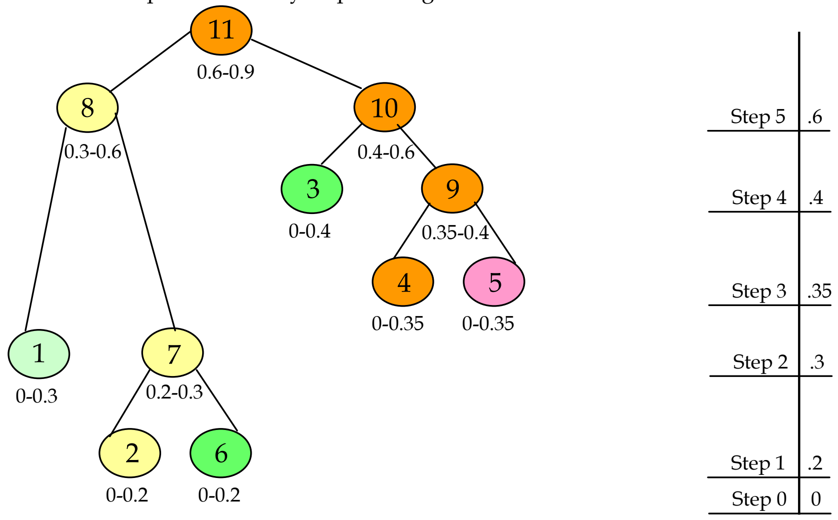

Figure 2. The face tree. Nodes in the tree are faces, and lines depict the merging of two faces into the parent face. Ranges associated to each node show the importance level at which the face is visible.

Merging is (currently) performed one by one: the least important face is merged to the most compatible neighbour. Merging is thus based on an importance value for each face, and compatibility between pairs of faces. The importance value of a face is calculated via a function e.g. as the product of the face area with the weight of the face class. The compatibility is calculated by another function, e.g. as the product of the length of the common boundary between two neighbour faces with a similarity value between their classes.

The merging process starts with the original faces. The least important face is selected first, then its most compatible neighbour. The two faces are merged to a new face, which gets the class of the more important face, i.e. the compatible neighbour. The importance value of the new face is calculated. The two merged faces are discarded from the next step of the merging process, and the new face is added. The next step continues in the same way: first selecting the current least important face, then its most compatible neighbour, merging these two faces to a new one that takes the class of the compatible neighbour. Each step performs a change in the map, while between two consecutive steps the map is unchanged. Therefore, the number of steps is equal to the number of changes a map undergoes. (For the current generalisation this is one less than the number of faces at the highest LoD.)

Edge forest

Area features can be stored in two different ways: explicitly storing their geometry, or storing the geometry of boundary edges together with references to faces of which they form the boundary. The second way of storing is not redundant, and it is known as topological storage (or topological model). Different topological models exist, e.g. the left-right topology, or winged edge topology. We store faces using the left-right topology without edge references. This model stores the edge geometry (as a directed arc with start and end node), together with references to the left and right face of the edge. Each face is then constructed from the list of edges that refer to it as a left or right face. That determines the types of face changes that effect edges. Edges that constitute the boundary of the two merged faces at a certain step undergo three kinds of changes. An edge disappears if it is part of the common boundary of the two merged faces. The other edges may continue existing, but the left or right face of each edge is changed. Two edges may join to form only one edge. An edge takes the importance value from the importance of the step in which it changed. The three cases of edge changes are illustrated with examples in the coming paragraphs.

Edge ‘k’ is common boundary between the merged faces 2 and 6 (see Figure 1), and it disappears at step 1, importance level equal to 0.2. An example of an edge that changes its left or right face is edge ‘h’ in step 1; its left face changes from 2 in step 0, to face 7 in step 1. Figure 3 shows the changes occurred to the edges in step 1. Edge ‘k’ disappears, edge ‘d’ changes its right face from 2 to 7, edges ‘e’ and ‘h’ change their left face from 2 to 7.

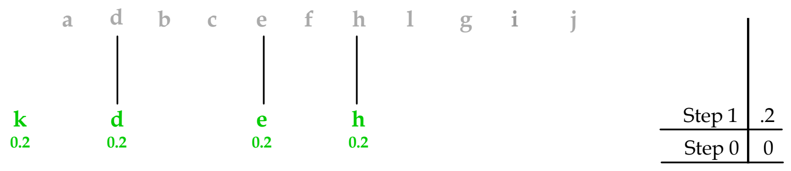

Figure 3. Changed edges in Step 1. The green colour shows edges to which change have occurred.

When a face disappears, like face 5 in Step 3 (see Figure 1), its boundary edge ‘i’ joins the connected edges ‘g’ and ‘j’. Figure 4 shows changes that occurred to edges in Step 3, importance equal to 0.35. The three edges ‘g’, ‘i’, ‘j’ joined to edge ‘n’, and disappear at that step. Edge ‘l’ disappears, as it is the common boundary between the merged faces, 5 and 4. Edge ‘c’ changes its left face from 4 to the new created face 9.

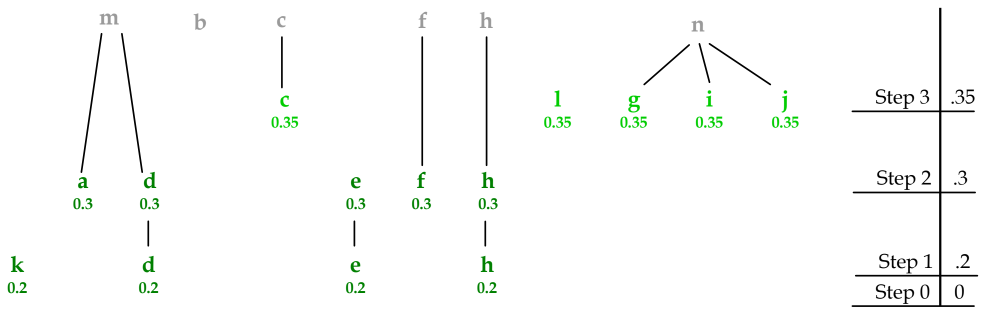

Figure 4. Changes edges in Step 3. The light green shows changes happening in this step, the dark green are changes occurred in previous steps.

Edges that disappear at a step remain isolated from the growing hierarchy of edges. This hierarchy does not have a single root; therefore it is not a tree. We call it a forest. The complete edge forest for the merging process of Figure 1 is shown in Figure 5. Ranges attached to an edge form the importance range at which the edge (associated with the left-right face information) exists. In the right sides there are the steps at which changes occurs, each step associated with its importance level. Nodes in the forest are aligned with the step at which the edge information is changed.

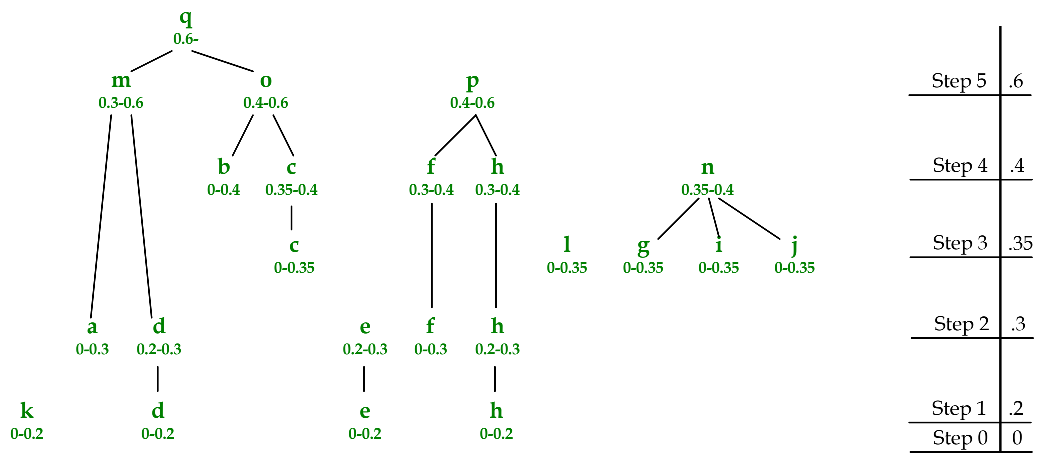

Figure 5. The edge forest. Ranges associated to a node show the importance values at which the edge information is unchanged.

The BLG trees

There is a BLG (Binary Line Generalisation) tree for each edge. The BLG tree stores the results of the Douglas-Peucker algorithm for line simplification. Douglas-Peuker algorithm is on the oldest and most popular algorithms used for line simplification. It uses the closeness of a vertex to a line segment to decide if a vertex will be included or not in the simplified version of the line for a given tolerance.

The algorithm starts with the roughest approximation of an edge being the straight line connecting the two end nodes. For each inner vertex, the distance to this straight line is calculated. The furthest vertex from the straight line is included in the list of vertices forming the next approximation of the edge. This new approximation consists then of two line segments (see Figure 6). For each new line segment, distances of all inner vertices to the line segment are calculated. Again, the furthest vertices to each line segment are included for the next edge approximation. This process continues until all vertices have a distance assigned. This distance is considered as a tolerance value for the vertex, and it is used to decide if the vertex will be shown for a certain LoD.

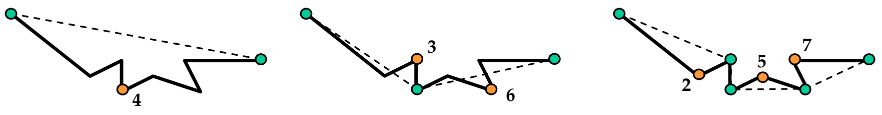

Figure 6. Three steps of Douglas-Peuker algorithm for edge simplification. The edge is drawn in thick black line; edge approximation on each step is drawn in thin dashed line. Vertices in green are part of the approximation; in orange are vertices selected for inclusion in the next approximation.

Figure 6 illustrates the application of Douglas-Peuker algorithm to an edge. The first step (left image) is the roughest approximation of the edge as the line segment connecting the start and end nodes of the edge, shown in green circles. The furthest vertex from this line segment is the 4th vertex (shown in orange). It is selected to be part of the approximation for the next step. Edge approximation in the second step is made of the line segment $\langle v_1, v_4\rangle$ between 1st and 4th vertex, and the line segment $\langle v_4, v_8\rangle$ between 4th and 8th vertex. For each line segment, the furthest vertex is calculated: vertex 3 is the furthest from $\langle v_1, v_4\rangle$ line segment, and vertex 6 is the furthest from $\langle v_4, v_8\rangle$ line segment. The 3rd and 6th vertex are added to the edge approximation for the next step, which is made of four line-segments $\langle v_1, v_3\rangle$, $\langle v_3, v_4\rangle$, $\langle v_4, v_6\rangle$, and $\langle v_6, v_8\rangle$. The third step calculates the distance of 2nd vertex from line $\langle v_1, v_3\rangle$, distance of 5th vertex from $\langle v_4, v_6\rangle$, and distance of 7th vertex from $\langle v_6, v_8\rangle$. Addition of these vertices to the third approximation produces the original edge. The algorithm finishes after the third step.

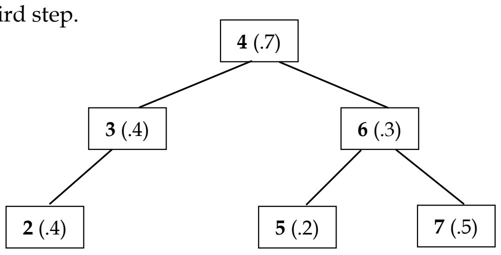

Figure 7. The BLG tree resulting from Douglas-Peuker edge simplification of Figure 6. Vertex number is shown in bold face, and tolerance for each vertex is given in brackets.

Results of the Douglas-Peuker algorithm are stored in a tree. Nodes of the tree are inner vertices of the edge, associated with the tolerance (i.e. the calculated distance). The root of the tree if the furthest vertex from the straight line connecting end nodes of the edge. Each step of the Douglas-Peuker algorithm adds to every leaf node v_i of the current tree (created from previous step) at most two nodes, added in the tree as children of v_i. The new nodes are the furthest vertices to the two line segments parting from v_i. They are one at the left and the other at the right of v_i, and are put accordingly in the tree to the left and right of node v_i. The tree formed in that way is a binary tree. Figure 7 gives the BLG tree for the edge simplification shown in Figure 6, and Figure 8 shows Douglas-Peuker algorithm for edges 'i' and 'j' together with their BLG trees storing the results of the algorithm.

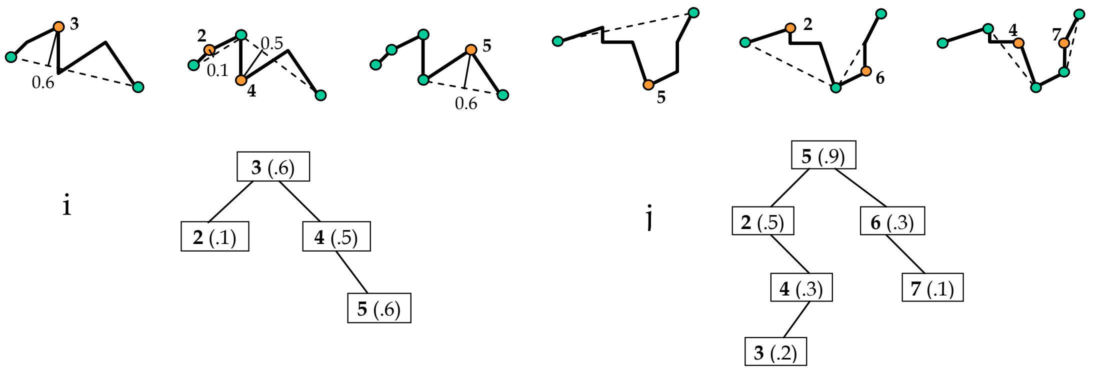

Figure 8. Douglas-Peuker simplification for edges 'i' and 'j', and their BLG trees.

There are cases when existing edges are to be joined to form only one edge boundary of a face at a lower LoD. For example, edges ‘g’, ‘i’, and ‘j’ are joined to edge ‘n’ in step 3 of the generalisation process (see Figure 1 and Figure 4). The geometry of edge ‘n’ is the union of geometries of the ‘g’, ‘i’, and ‘j’ edges. We use the BLG trees of the composing edges to form the geometry of the new edge ‘n’. For each part of ‘n’ – ‘g’, ‘i’, and ‘j’ – we use the simplification performed already, i.e. we use the BLG trees of ‘g’, ‘i’, and ‘j’. The common node between ‘g’ and ‘i’, and the common node between ‘i’ and ‘j’, are inner vertices for the new edge ‘n’, but they have no tolerance value assigned. Joining of BLG trees is done in pairs, and a tolerance value is calculated for the common node. Figure 9 (left) shows the joined BLG tree for edges ‘i’ and ‘j’. A tolerance value 1.4 is associated to the common node, named ‘ij’.

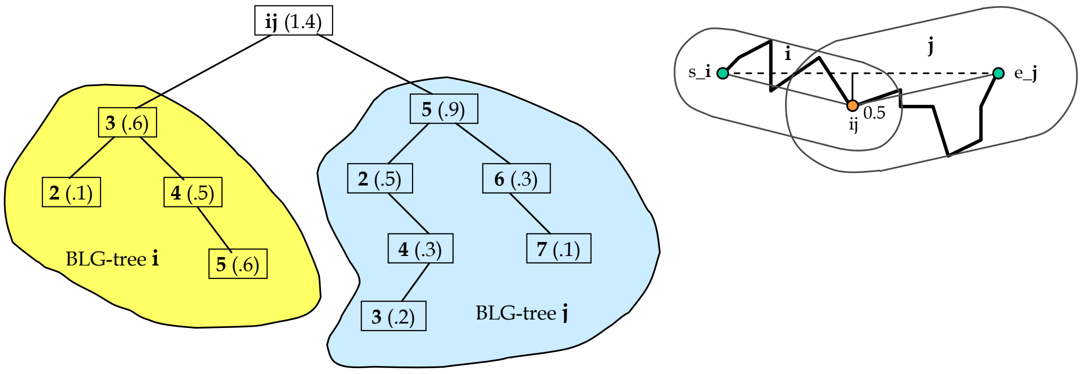

Figure 9. Joining of two BLG trees (left), and the tolerance calculation for the common node (right).

Figure 9 (right) illustrates the calculation of the tolerance for the common node. The tolerance value for the common node ‘ij’ is not calculated in the same way as for inner vertices of an edge (by Douglas-Peuker algorithm). It is estimated from the top tolerance of edges ‘i’ and ‘j’, and the distance of node ‘ij’ to the straight line connecting the end nodes of the joined line. The formula for the calculation is: $tol_{ij} = \max\{tol_{root(i)}, tol_{root(j)}\} + dist(ij, \langle s\_\mathbf{i}, e\_\mathbf{j}\rangle) = \max\{0.6, 0.9\} + 0.5 = 1.4$

When there are more than two edges to be joined, as it is the case of edge ‘n’ composed of three edges ‘g’, ‘i’, and ‘j’, the full joining is done in steps. For the example of edge ‘n’, BLG trees of ‘i’ and ‘j’ are joined first, then the joined BLG of ‘i’ and ‘j’ is joined with the BLG tree of ‘g’. The tolerance of the common node is again estimated as previously. Tolerance value for the common node is bigger than the tolerance of all inner vertices, which means a common node is the first vertex selected for joined edge approximation, and during visualization will appear before any other vertex.

2. Using tGAP structure

Once the tGAP structure is built, it can be used to select features that should be shown for a certain scale. Once a map scale is given, it is translated to importance value, which is used to select features. A face will be shown if the given importance value is in the importance range of the face. Figure 10 gives faces to be shown for an importance value equal to 0.38. The importance value 0.38 is in the importance range [0.35, 0.4) formed between steps 3 and 4. The map created from Step 3 is unchanged for values in this range. Faces to be shown are the leaf nodes of the (sub)tree created by cutting all nodes with importance values lower than 0.38. These are faces 3, 8, and 9; their importance ranges include the value 0.38. They form a partition of space, being leaf nodes of the binary (sub)tree.

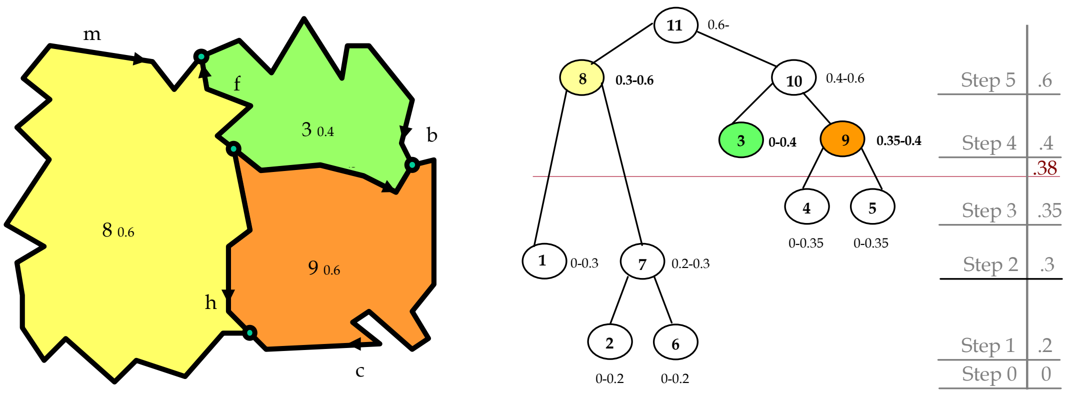

Figure 10. Faces to be shown for the importance level 0.38.

Edges to be shown at the importance value 0.38 are leaf nodes in the forest remained after cutting nodes with importance less than 0.38. Those are the edges that include the importance value 0.38 in their importance range. They are the boundaries of faces to be shown for that importance, namely, faces 3, 8, and 9. Figure 11 shows the edges to be displayed at the importance value 0.38.

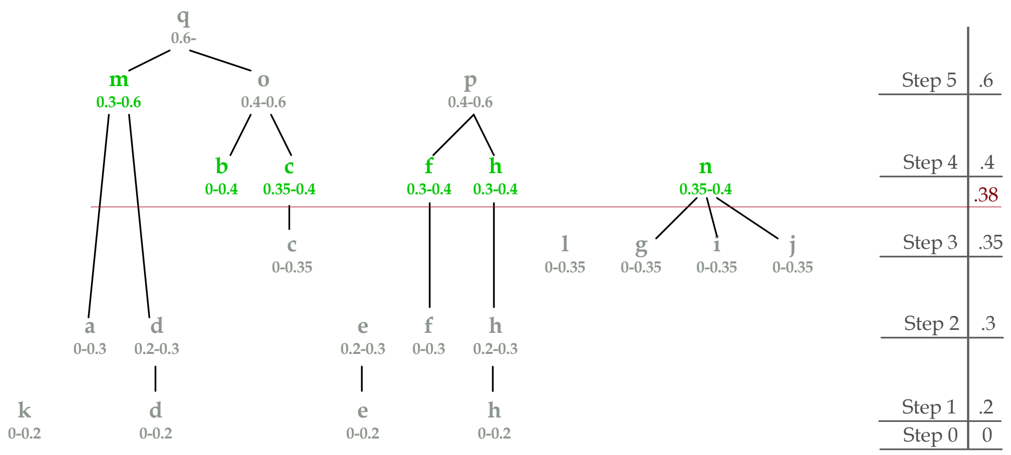

Figure 11. Edges to be shown for the importance level 0.38. Selected edges are drawn in orange.

A relation can be established between map scale and tolerance values of edge vertices. Once a scale is chosen for a map, it has to be translated to a tolerance value. For each edge to be shown at that scale, this tolerance value is used to select nodes from its BLG tree. The selected vertices, together with the start and end node of the edge, form the edge approximation that is to be displayed.

The Douglas-Peuker algorithm is non-monotonic for the Euclidean distance used for tolerance calculation. That is to say, going down the tree does not guarantee decreasing tolerance values. For example, the BLG tree of edge 'j' – Figure 8, right – has decreasing tolerance values, whereas the BLG trees of edges 'g' and 'i' – Figure 7 and Figure 8 left, respectively – do not have decreasing tolerance values. For a given tolerance value, a BLG tree is descended to select vertices that will form the edge geometry for that value. For example, the geometry of edge 'g' (see Figure 6, and Figure 7 for its BLG tree) for a tolerance value equal to 0.32 is made of vertices v_2, v_3, and v_4, together with its start and end node. Figure 12 shows how the BLG tree of edge 'g' is descended to select the right vertices for the given tolerance 0.32. The tolerance of the root is bigger than the given tolerance, therefore v_4 is selected. Its right child, v_6, has a tolerance smaller than 0.32. The node is not selected; descending stops at that node. Its left child instead has a bigger tolerance that .32. Node v_3 is selected; descending goes further in this direction. The only child of v_3, node v_4, has a tolerance bigger than 0.32 and it is therefore selected.

Comment [t1]: A paper: "M. A. Abam, M. de Berg, P. Hachenberger, and A. Zarei, Streaming Algorithms for Line Simplification, *to appear, 23rd ACM Symp. on Computational Geometry (SOCG)*, 2007" seems to propose a monotonic algorithm for a Fréchet distance.

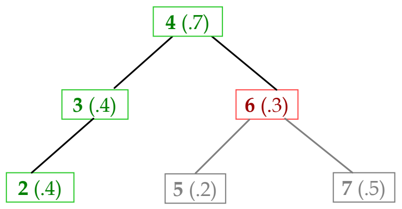

Figure 12. Descending a BLG tree to select vertices to be shown at tolerance value 0.32. Nodes in green are selected; the node in red is where the selection stops; its children (in grey) are not selected either.

3. Implementation of tGAP structure in Oracle Spatial

The tGAP structure is implemented as a collection of tables in Oracle Spatial (see Figure 13). Information about faces is stored in a tGAP_face table: face identifier, minimum bounding box, area size, importance range (imp_low and imp_high columns), parent id that allows to build the face tree, and a class attribute that is used from generalization. Information about edges is split into tables tGAP_blg, tGAP_node, and tGAP_edge. Geometry of edges is stored in the first two tables, which store BLG trees and start and end nodes, respectively. To remove redundancy, a BLG tree in the tGAP_blg table stores only the inner vertices of its edge. The start and end node of an edge are stored in the tGAP_node table, and retrieved via references start_node_id and end_node_id in the tGAP_blg table. Other columns in the BLG table store the BLG tree in tree_source, child1_id and child2_id store references to BLG trees for the joined BLG trees, and top_tolerance stores the root tolerance. When tree_source is filled, then child1_id and child2_id are empty, and vice versa. The tGAP_edge stores information of the edge forest: edge identifier, left and right face references, importance ranges that change with face references, and a reference to the corresponding BLG tree in tGAP_blg table.

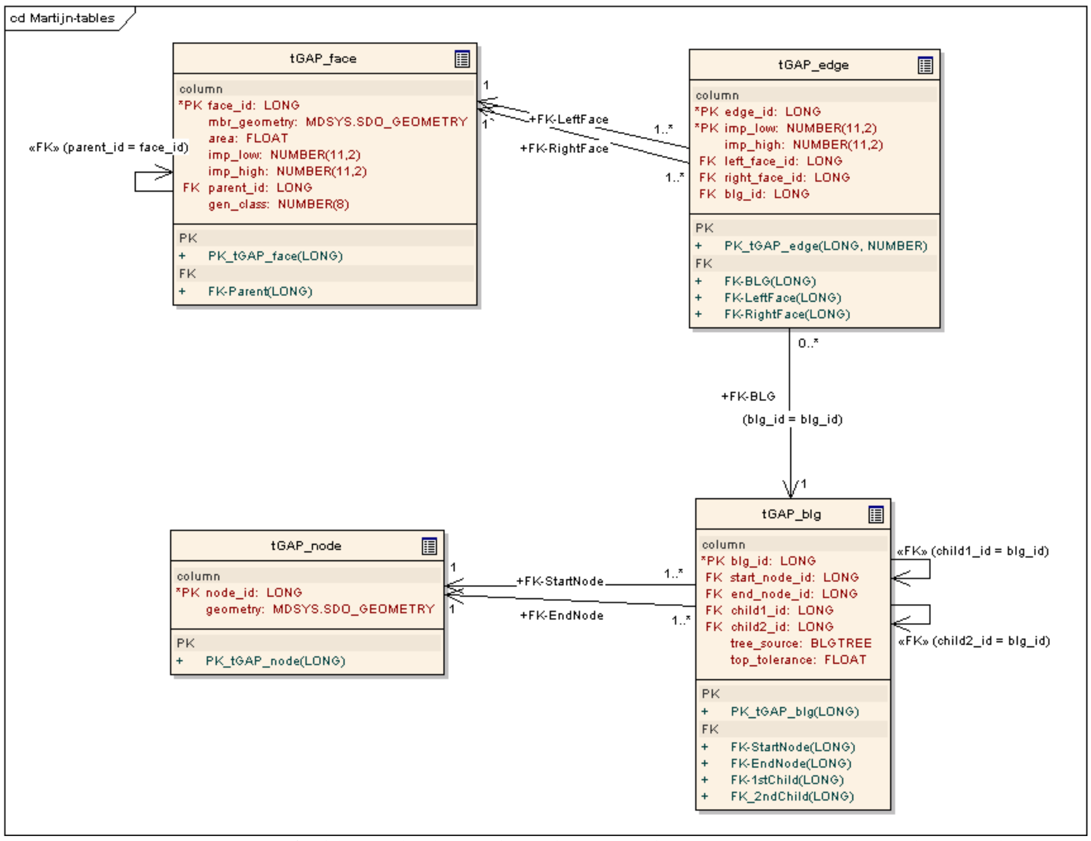

Figure 13. Diagram of tables and relationships that store the tGAP structure in Oracle Spatial.

A type BLGTREE is created for storing a BLG tree as a list of BLG nodes (in PL/SQL code):

```
create type BLGnode as object (
       x_coord number,
       y_coord number,
       left_node number,
       right_node number,
       tolerance float
);
create type BLGTREE as object varray(524288) of BLGnode;
```

The structure doesn't store explicitly the vertex position in the original (i.e. highest LoD) edge. Vertex positions are calculated following left and right references in the tree, when building edge geometry for a given tolerance.

Tables storing information about our example generalization (Figure 1) are given below: information about the face tree (Figure 2) is stored in tGAP_face table, and information about the edge forest (Figure 5) is stored in tGAP_edge table.

tGAP_face table

id	mbr_geometry	area	imp_low	imp_high	parent_id
1			0	0.3	8
2			0	0.2	7
3			0	0.4	10
4			0	0.35	9
5			0	0.35	9
6			0	0.2	7
7			0.2	0.3	8
8			0.3	0.6	10
9			0.35	0.4	10
10			0.4	0.6	11
11			0.6	0.9	

tGAP_edge table

id	imp_low	imp_high	blg_id	left_face_id	right_face_id
a	0.00	0.30	*a*	0	1
b	0.00	0.40	*b*	0	3
c	0.00	0.35	*c*	0	4
d	0.00	0.20	*d*	0	2
e	0.00	0.20	*e*	2	1
f	0.00	0.30	*f*	1	3
g	0.00	0.35	*g*	3	4
h	0.00	0.20	*h*	4	2
i	0.00	0.35	*i*	3	5
j	0.00	0.35	*j*	3	4
k	0.00	0.20	*k*	2	6
l	0.00	0.35	*l*	4	5
d	0.20	0.30	*d*	0	7
e	0.20	0.30	*e*	7	1
h	0.20	0.30	*h*	4	7
m	0.30	0.60	*m*	0	8
f	0.30	0.40	*f*	8	3
h	0.30	0.40	*h*	4	8
c	0.35	0.40	*c*	0	9
n	0.35	0.40	*N*	3	9
o	0.40	0.60	*O*	0	10
p	0.40	0.60	*P*	10	8
q	0.60		*Q*	0	11

A 3D functional index is built to insure fast access to features in a given spatial extent and a given importance range. It is based on a spatial index on bounding boxes of faces (or edges), plus importance values.

4. **Progressive transfer and visualisation**

Given a certain spatial extent (i.e. search rectangle) and a certain scale from the client, the server selects data from tGAP tables based on the spatial extent and a calculated importance range from (current and previous) scale. Then it starts sending these data progressively. The server sends edges ordered by their importance values (imp_high attribute alone, or in combination with imp_low?). Edge information sent by the server is their geometry, together with left and right face references. The topology of faces is to be built at the client side. A full importance range can be split into several intervals. The server collects all edges falling in an interval, which form boundaries of a partition of the given spatial extent. A complete (partition) collection is signalled to the client, which starts building topology for this collection edges. Faces are shown in the client screen. Other edges coming from the server start appearing in the screen. When a signal for (another) complete edge collection is coming, the client starts building topology for the new faces, and visualises them in the screen.

Edge geometry can be calculated in the server side and send to the client. This requires re-sending edge geometry any time a more detailed shape is needed. Done differently, full information about edge can be sent to the client only once. Client has functionality to build the right geometry (edge detail) for any tolerance/scale. The following paragraph goes in more detail about ways to perform these.

Edge geometry can be created in the server from the BLG tree and a tolerance (calculated from scale) given from client. Edge geometry is then sent to the client. When more detail is needed for a received edge, a complete new (edge) geometry should be created and sent for the required detail. Another possibility is to send, instead of edge geometry, the BLG tree of the edge, together with its start and end node. In this case, the BLG structure, including functionality to create edge geometry, is required in the client side. A similar approach for progressive transfer of data is followed in the GiMoDig project. A third possibility is to rewrite a BLG tree in the server side as a sequence of vertices, and send this sequence to the client. For each vertex the sequence contains information on vertex position (in the highest LoD edge), vertex-coordinates, and vertex tolerance. The sequence can be ordered on tolerance values, which allows fast selection of vertices to be shown for an edge at a given scale/tolerance. For example, the order of vertices for the BLG tree of Figure 7 is $\langle(v_4, 0.7); (v_3, 0.4); (v_2, 0.4); (v_6, 0.3); (v_7, 0.5); (v_5, 0.2)\rangle$ – tolerances are not monotonically decreasing, whereas edge 'j', Figure 8 right, has decreasing tolerances: $\langle(v_5, 0.9); (v_2, 0.5); (v_6, 0.3); (v_4, 0.3); (v_3, 0.2); (v_7, 0.1)\rangle$. Selected vertices for a given tolerance will be ordered by the client according to the vertex position for visualisation on the screen.

Literature

1. Martijn Meijers (2006). *Implementation and testing of variable scale topological data structures*. Master's Thesis TU Delft, 2006, 114 p.

2. Peter van Oosterom (1990). *Reactive Data Structures for Geographic Information Systems*. PhD-thesis Department of Computer Science, Leiden University, December 1990.

3. Peter van Oosterom (2005). Variable-scale topological data structures suitable for progressive data transfer: the GAP-face tree and GAP-edge forest *Cartography and geographic information science, 32*, 331-346.

4. GiMoDig – "Geospatial Info-Mobility Service by Real-Time Data-Integration and Generalisation" (http://gimodig.fgi.fi/)