Server side technology for tGAP

August 10, 2007

This report presents the data structures we use for on-the-fly map generalisation. The generalisation process is prepared off-line, and its results are stored in the data structures. For any required level of detail (LoD)[1], the data structures provide the features to be shown, thus enabling an on-the-fly generalisation by feature selection. The data structures support a variable scale representation of an area partitioning without redundancy of geometry. Area features at the highest LoD are stored using a topological model; there is no redundancy of geometry in this level as the shared boundary edges between neighbour faces are stored only once. The generalisation process reduces the level of detail (mainly) by merging unimportant features to more important features (see Figure 1). For features created from generalisation, the data structures store references to the composing features of higher detail level.

The collection of data structures is called tGAP, for topological Generalised Area Partitioning. This report explains the tGAP structure and its implementation in Oracle. Generalisation process shown in Figure 1 is used for illustration. The map shows a part of Ameland island, in the north of the Netherlands, with objects sea, foreshore, beach and fore dune. Section 1 describes how the tGAP is filled from the generalisation process; Section 2 explains how tGAP is used to select features for a required LoD. The implementation of tGAP structure in Oracle Spatial is explained in Section 3. Section 4 gives some ideas about interaction with the client side.

1 Building tGAP structure

An automatic map generalisation is performed by generalisation operators ([2] gives examples of such operators). The generalisation process we perform includes some of the operators described by [2][2]:

Reclassification changes the class an object belongs to and merges it with its neighbours. For example, face 5 (sea) in Figure 1 'Step 0' is reclassified to foreshore and merged to face 4.

Elimination removes an object from the data set, and assigns the freed space to other objects; e.g. face 2 is eliminated in 'Step 3', it is first split and then its parts are merged with the neighbour faces, 6 and 7.

Collapse changes the type of geometry; an area feature is collapsed to a line or a point. This is needed, for example, when an area feature presenting a road in a large scale map should remain visible for a smaller scale, but only as a line feature.

Simplification reduces the granularity of an edge. See, for example, edge 'f' from 'Step 1' to 'Step 2' in Figure 1. Edge simplification is indeed performed for most of the edges in consecutive steps.

Reclassification is performed by merging less important faces to more important faces, based on importance values associated to each face and compatibility with neighbours. Elimination and collapse are performed by first splitting a face to parts, using, e.g. skeletons as described in [1]. Central axis may be given (in a data set) for roads or rivers. This will be used as the collapsed linear feature of a road (area). Split parts are then merged to their neighbours. During collapse, new linear or point features that are created become independent features in the map. During elimination, the new features are simply edge boundaries or nodes in the map.

[1] We use the terms LoD and map scale interchangeably.

[2] Other operators may be added later.

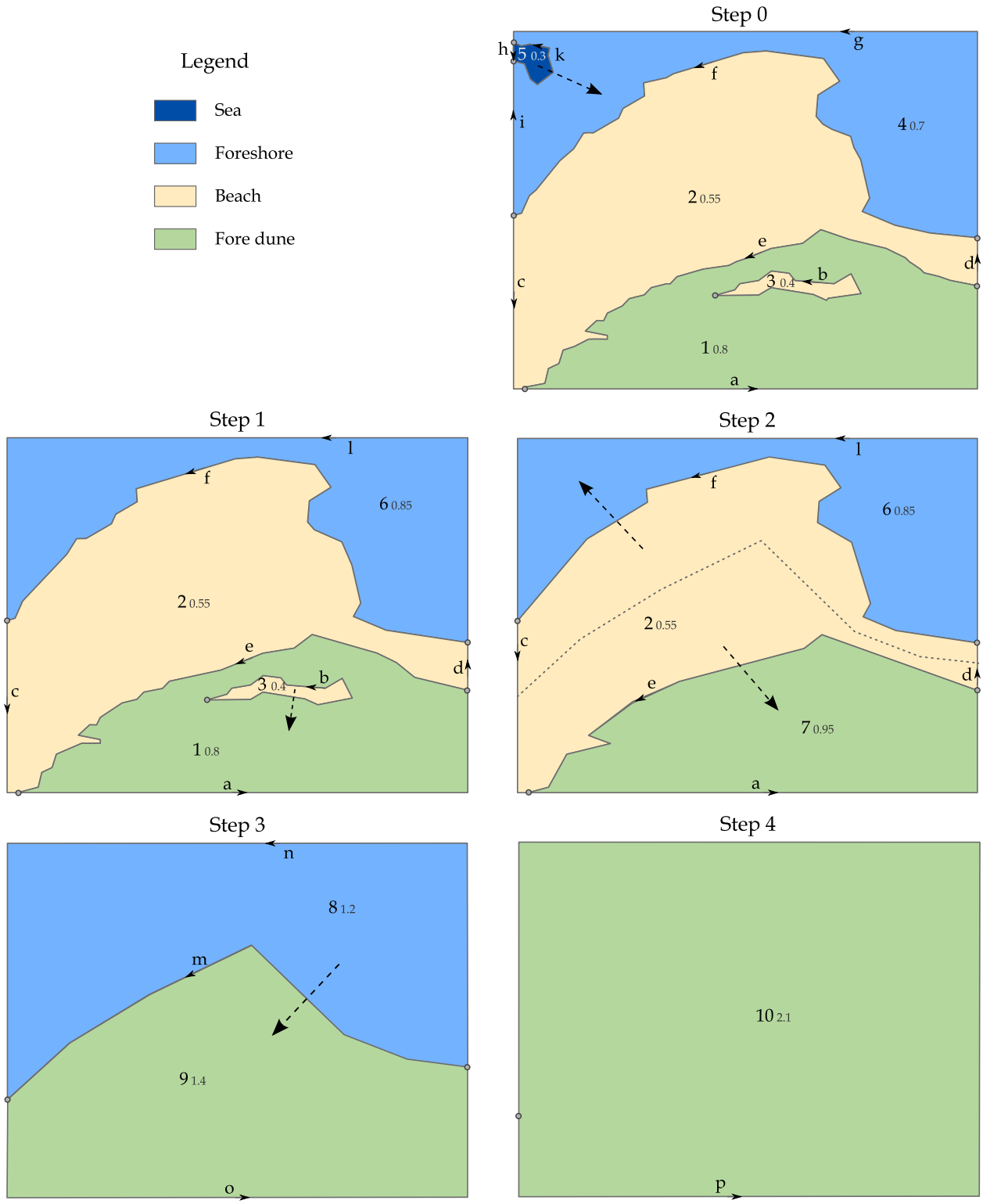

Figure 1: Steps of a generalisation process performing reclassification, elimination, and simplification of edges. Faces are numbered, and edges are labelled with letters. The subscript to a face number is its importance value.

Figure 1 illustrates the generalisation process for the area partition shown in 'Step 0'. The other maps show the result of the generalisation in steps, and are labelled according to that. A dashed arrow shows the least important face for the current step, and its most compatible neighbour (where the arrow is headed). In the next step, the least important face is merged to its most compatible neighbour, or it is split between its neighbours. In the original map, Step 0, face 5 has the lowest importance value, 0.3, and face 4 is its only neighbour, therefore the most compatible face. In step 1 the two faces are merged into a new face, labelled 6. Faces 4 and 5 cease existing at the importance level 0.3, whereas face 6 starts existing at this importance level. The importance value of face 6 is calculated, 0.85, and the face is considered for the next step in the generalisation process. Face 2 (importance value 0.55) is split between its neighbours: one part is merged with face 6 and the other with face 7. Faces 2, 6, and 7 disappear in Step 3, importance 0.55, and two new faces, 8 and 9, start existing at this step. The process continues until all is merged to one face, labelled 10.

The result of each generalisation step is an area partition. A partition 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. These are edges that are in the boundary of the visible faces. There is a last issue in the generalisation process: boundary edges get simplified as the level of detail decreases. This can be also seen in Figure 1. To capture the generalisation process we need to keep track of the merging, and possibly splitting of faces in each step, how this is reflected to 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 structure holding the hierarchy of faces, an edge forest that holds the corresponding relations between boundary edges, and BLG (Binary Line Generalisation) trees, one tree for each edge that holds information about edge simplification. Building of each structure is treated separately in the coming sections.

1.1 DAG structure of faces

Generalisation is performed by merging and splitting. Merging is 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 least important face is merged to its most compatible neighbour. *How to decide which face should be first split, then merged? A solution may be when compatibility is the same with all neighbours. Also which face should be collapsed? Collapsing to line features may be easier, dependent on a class attribute of area features. For example, roads and rivers (which form networks) are shown as lines in small scale maps, therefore areas belonging to these classes should be collapsed to lines. Compatibility value of such a class to any other class may be set to 0.*

Generalisation is performed in steps, starting with the original (i.e. highest LoD) faces. A generalisation step merges two neighbour faces to a new one, or it splits a face and merges its parts to the neighbours. The merged faces, together with the split face, are replaced by the new face(s), which continue further in the generalisation process. The process ends when only one face is left. Each step performs a change in the partition; the area partition is unchanged between two consecutive steps[3]. Therefore, the number of steps is equal to the number of changes a map undergoes. New face(s) and the merged faces have a parent-child relation. The hierarchy of faces created by this process is a directed acyclic graph, DAG. Leaf nodes are the original faces; the root is the full map extent. Faces created during the generalisation process form the other nodes in the DAG structure.

Figure 2 shows the face hierarchy created by the generalisation process of Figure 1. A node in the hierarchy is a face. 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 are levelled with the step in which they change. Faces 4 and 5 are merged to face 6 in Step 1; faces 1 and 3 are merged to face 7 in Step 2; in Step 3 face 2 is split and its parts are merged to faces 6 and 7, giving faces 8 and 9, respectively; faces 8 and 9 are merged in Step 4 giving the full map extent, face 10.

[3]The map visualising the partition may change though, because edges may be shown in different level of detail.

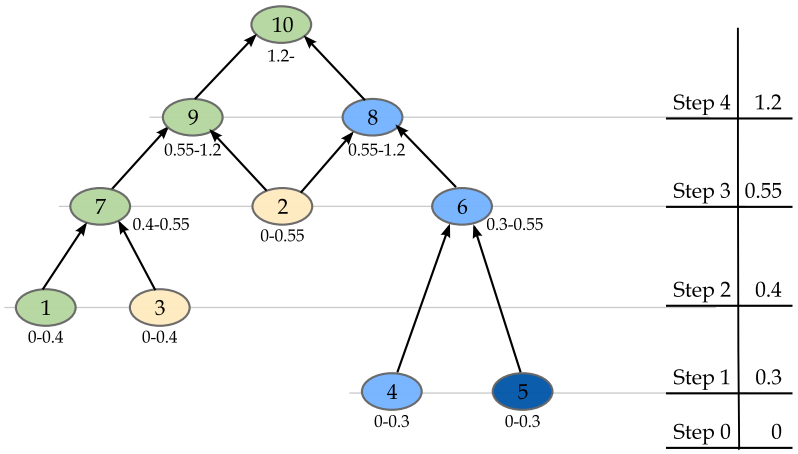

Figure 2: DAG structure. Nodes in the DAG structure are faces, and lines depict merging of two faces into the parent face.

1.2 Edge forest

Area features can be stored in two different ways: explicitly storing their complete 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 references), 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 type of changes an edge undergoes. All cases of edge changes are illustrated with examples in the coming paragraphs. An edge takes the importance value from the importance of the step in which it changed.

Edges that constitute the boundary of the 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. These three kind of changes are illustrated in Figure 3 that shows changes occurred to the edges in Step 1. Edge ‘k’ is common boundary between the merged faces 4 and 5 (see Figure 1), and it disappears at

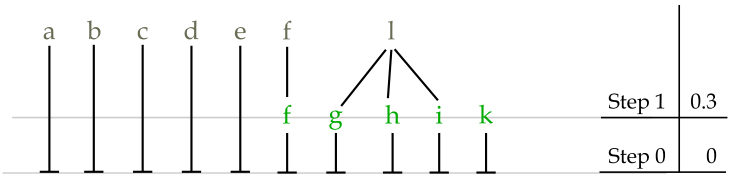

Figure 3: Existing edges in Step 1: edges f–k undergo changes, no changes to edges ‘a’–‘e’,

importance level equal to 0.3. Edge ‘f’ changes its right face from 4 in Step 0, to face 6 in Step 1. When face 5 disappears, its boundary edge ‘i’ need not exist separately; it joins the connected edges ‘g’ and ‘h’. Edges ‘g’, ‘’h’, and ’i’ are joined to a new edge ‘l’, and disappear at Step 1.

In case of splitting, the skeleton (or central axis) used for the split is added to the collection of edges. Existing edges are extended/snapped to the new edge(s) in order to close the area. In Step 3 of the generalisation of Figure 1, the closed areas are formed by parts of existing edges, ‘c’ and ‘d’. This can be seen in Figure 4, which shows the complete edge hierarchy for the generalisation process of Figure 1. In the right side there are the steps at which changes occurs, each step associated with its importance level. Edges that

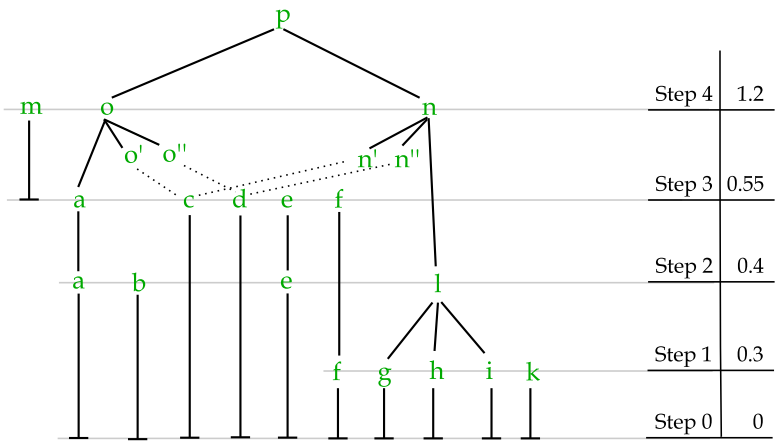

Figure 4: The edge forest for the generalisation process of Figure 1.

disappear at a step remain isolated from the growing hierarchy of edges. This hierarchy does not have a single root. We call it a forest.

1.3 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 the most popular algorithm 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 5). For each new line segment, distances of all inner vertices to the line segment are recalculated. 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.

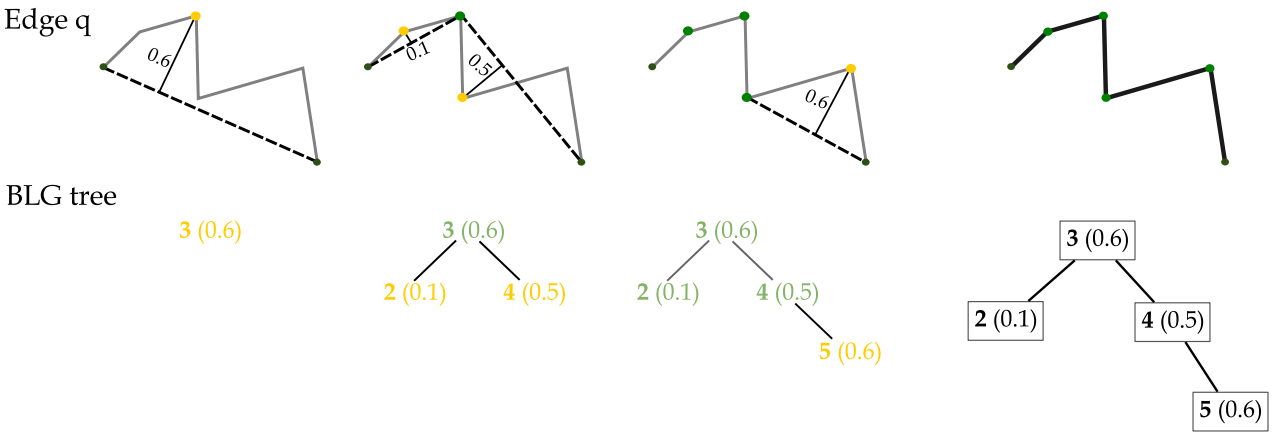

Figure 5: Three steps of Douglas-Peuker algorithm for edge simplification, and its BLG tree. The edge is drawn in dark gray line; edge approximation on each step is drawn in dashed line. Vertices in green are part of the approximation; in orange are vertices selected for inclusion in the next approximation.

Figure 5 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 third 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 $\langle v_1, v_3 \rangle$ between the first and third vertex, and the line segment $\langle v_3, v_6 \rangle$ between third and sixth vertex. For each line segment, the furthest vertex is calculated: vertex 2 is the furthest from $\langle v_1, v_3 \rangle$ line segment, and vertex 4 is the furthest from $\langle v_3, v_6 \rangle$ line segment. The second and fourth vertex are added to the edge approximation for the next step, which is made of four line-segments $\langle v_1, v_2 \rangle$, $\langle v_2, v_3 \rangle$, $\langle v_3, v_4 \rangle$, and $\langle v_4, v_6 \rangle$. The third step calculates the distance of the fifth vertex from line $\langle v_4, v_6 \rangle$. Addition of this vertex to the third approximation produces the original edge, Figure 5 top-right. The algorithm finishes after the third step.

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 start and end node of an edge are stored separately, not in its BLG tree. 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 5 bottom gives the BLG tree for results of Douglas-Peuker algorithm on the edge 'q' shown on top.

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', 'h', and 'i' are joined to edge 'l' in step 1 of the generalisation process (see Figure 1 and Figure 4). The geometry of edge 'l' is the union of geometries of the 'g', 'h', and 'i' edges. We use the BLG trees of the composing edges to form the geometry of the new edge 'l'. The common node between 'g' and 'h', and the common node between 'h' and 'i', are inner vertices for the new edge 'l', 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 6 illustrates Douglas-Peuker algorithm for an edge 'r' and its BLG tree storing the results of the algorithm. Figure 7(right) illustrates the calculation of the tolerance for the common node of

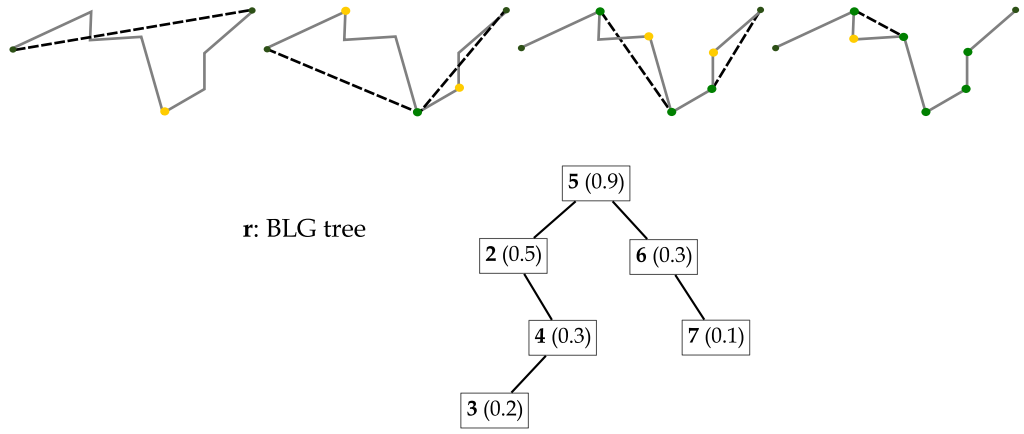

Figure 6: Douglas-Peuker algorithms for another edge, and its BLG tree.

two edges 'q' and 'r'. The tolerance value for the common node 'qr' is estimated from the top tolerance of edges 'q' and 'r', and the distance of node 'qr' to the straight line $\langle q_{sn}, r_{en} \rangle$ connecting the end nodes of the joined line. The formula for the calculation is: $tol_{qr} = \max\left\{tol_{root(q)}, tol_{root(r)}\right\} + dist(qr, \langle q_{sn}, r_{en} \rangle) = \max\{0.6, 0.9\} + 0.5 = 1.4$. When there are more than two edges to be joined, as it is the case of edge 'l' composed of three edges 'g', 'h', and 'r', the join is performed in steps. For 'l', BLG trees of 'g' and 'h' are joined first, then the joined BLG of 'g' and 'h' is joined with the BLG tree of 'i'. The tolerance of the common node is again estimated as previously. A common node is the root of a joined BLG, therefore it is the first vertex selected for joined edge approximation, and during visualisation will appear before any other vertex.

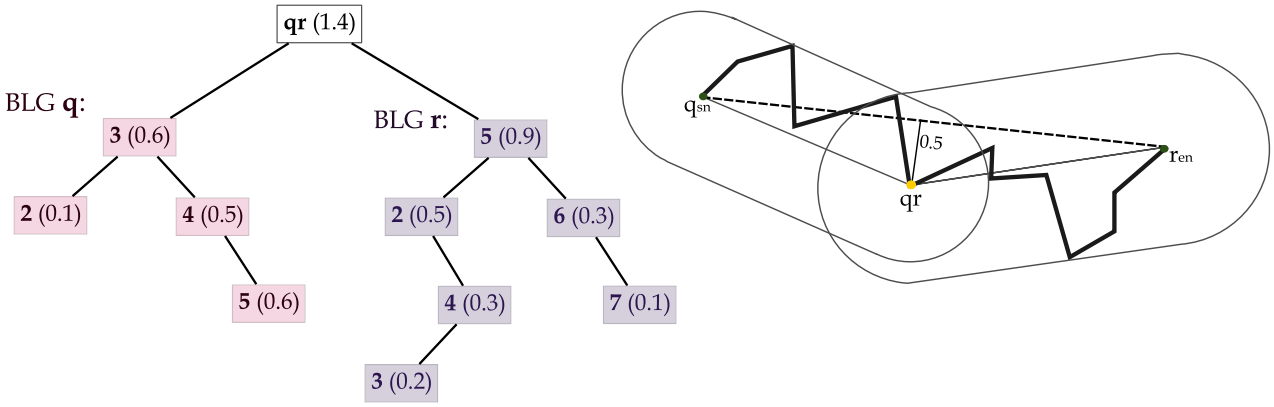

Figure 7: Joining of two BLG trees (left), and the tolerance calculation for the common node (right).

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. A given map scale 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 8 gives faces to be shown for an importance value equal to 0.48. The importance value 0.48 is in the importance range [0.4, 0.55) formed between steps 2 and 3. The map created from Step 2 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 0.48. These are faces 2, 6, and 7; their importance ranges include the value 0.48.

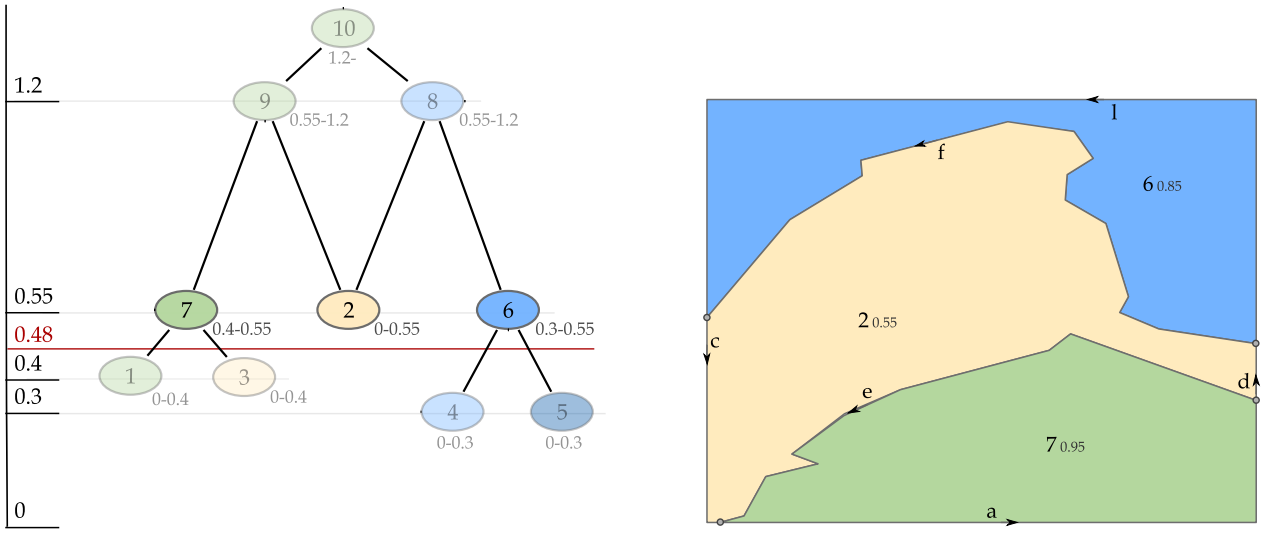

Figure 8: Faces to be shown for the importance level 0.48.

Edges to be shown at the importance value 0.48 are leaf nodes in the forest remained after cutting nodes with importance less than 0.48. Those are the edges that include the importance value 0.48 in their importance range. They are the boundaries of faces to be shown for that importance, namely, faces 2, 6, and 7. Figure 9 shows the edges to be displayed at the importance value 0.48.

A relation can be established between map scale and tolerance values of edge vertices. A given map scale is to be translated to a tolerance value. This tolerance value is used to select nodes from the BLG trees of

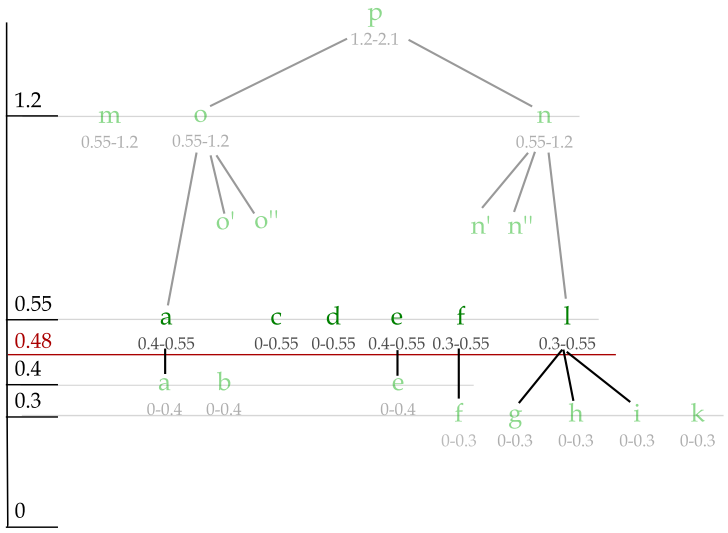

Figure 9: Edges to be shown for the importance level 0.48: the end nodes of the sub forest created by the 0.48 level.

visible edges for that scale. The selected vertices, together with the start and end node of the edge, form the edge approximation that is to be displayed.

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 'q' (Figure 5) does not have decreasing tolerance values, whereas edge 'r' (Figure 6) has 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 'q' for a tolerance value equal to 0.52 is made of vertex v_3 together with 'q' start and end node. The tolerance of the root is bigger than the given tolerance, therefore v_3 is selected. Its children v_2 and v_4 have a tolerance smaller than 0.52; therefore are not selected. Descending stops at these nodes. Although the left child of v_4, node v_5, has a tolerance bigger than 0.52, it is not selected. Figure 10 shows how the BLG tree of edge 'q' is descended to select the right vertices for the tolerance value 0.52.

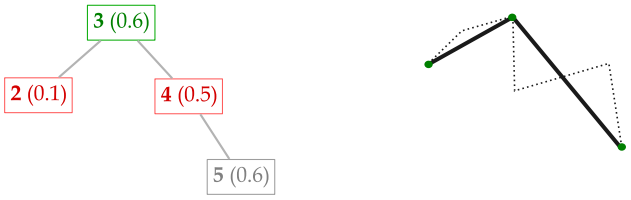

Figure 10: Descending a BLG tree to select vertices to be shown at tolerance value 0.52. Nodes in green are selected; nodes in red show where the selection stops; nodes in gray are not checked.

3 Implementation of tGAP structure in Oracle Spatial

Figure 11 shows a diagram of classes for storing tGAP information. It holds similarity with a topological model of spatial data: faces, edges, nodes, and their relationships: left-face, right-face, start-node, and end-node. AreaFeat class holds information about faces: an identifier, importance range, and the class to which

it belongs[4]. Operations of this class, faceGeometry, mbrGeometry, and areaSize, calculate the geometry of a face from its boundary edges, the bounding box, and the area size, respectively. The bounding box is needed for fast selection (using spatial indices); the area size is used for calculation of importance values in the generalisation process. For speeding up these processes, the two attributes may be stored. Edge class holds information on edges: an identifier, edge geometry (stored as a BLG tree), and importance range that is a multiple valued attribute. Class operations, ls2geometry, aboxGeometry, and lineLength, calculate edge geometry from the BLG tree, the bounding rectangle of both left and right faces of the edge, and length of the edge. The bounding rectangle will be used for fast edge selection; line length is used during generalisation for calculation of compatibility with left and right faces. Both may be stored as attributes of Edge. Node class stores identifier and geometry information for nodes. Relationships StartNode and EndNode hold the start and end node of each edge, respectively. Relationships LeftFace and RightFace hold left faces and right faces of an edge, respectively. An edge may have several left faces, and several right faces (at different LoD). Parent relationship between faces holds the parent-child relations in the DAG structure. There are

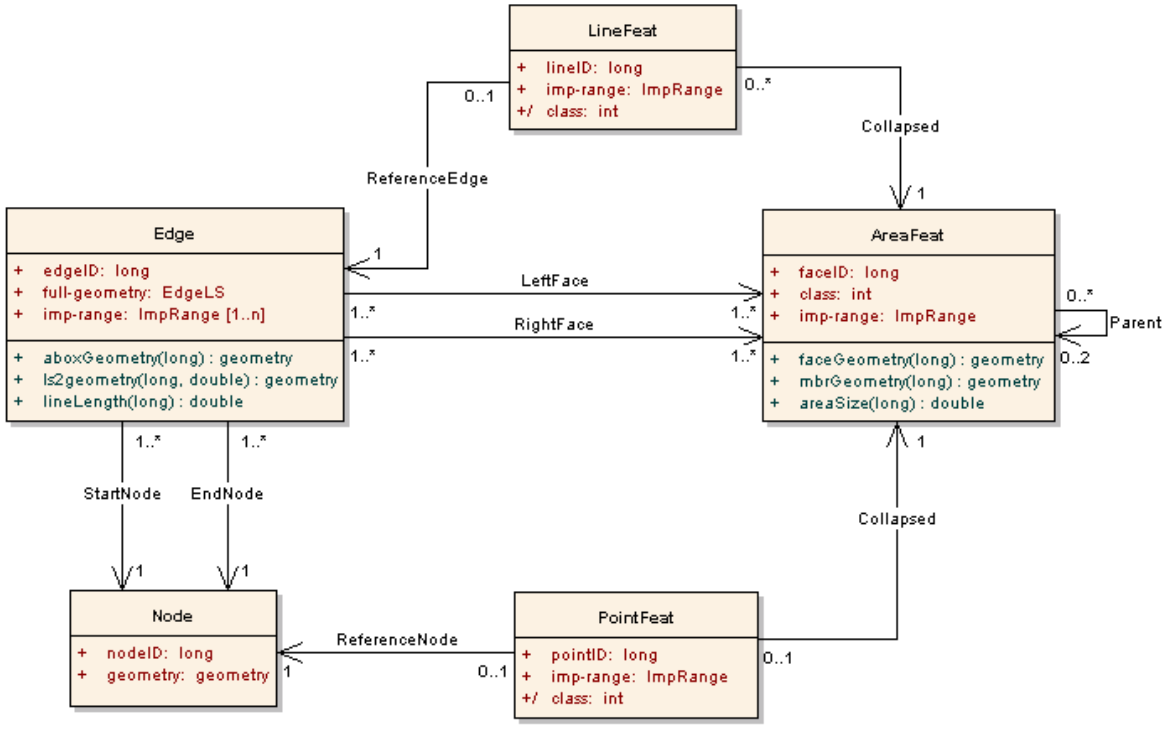

Figure 11: UML class model for tGAP structure.

two more classes in the diagram: LineFeat representing a line formed by an area collapse, and PointFeat representing a point feature formed by an area collapse. The class of the area feature is transferred to the line or point feature after collapse. (*Are there cases when this should be different?*) Collapsed relationship with AreaFeat class holds the relation between an area feature and its collapsed line or point feature. The geometry of a LineFeat object is kept at Edge and retrieved via ReferenceEdge relationship. Similar for a PointFeat object, its geometry is stored by Node and retrieved via ReferenceNode relationship.

tGAP information is to be stored in Oracle 10g Spatial. The class model of Figure 11 is translated to Oracle tables shown in Figure 12. Information about faces is stored in AreaFeature table. Information about edges is split in two tables: EdgeGeometry and EdgeLOD. The first table contains an identifier for an edge, its BLG tree(s), start and end node references. Table EdgeLOD stores left-, right-faces of an edge as they change during the generalisation, and the (corresponding) importance ranges. LineFeature table stores information of lines created from collapsing areas: reference to edge in EdgeGeometry table for the geometry, reference

[4]The class attribute holds values of the classification that forms the area partitioning.

to the corresponding face in AreaFeature table, and the importance range of the feature (different from the importance range of the corresponding face). Node information is stored in Node table. Information about points created from collapsed areas is stored in PointFeature table; it is similar to the information stored about line features. Arrows associating tables show foreign key relationships. Compatibility matrix and class weights, used from the generalisation process, may also be stored as tables.

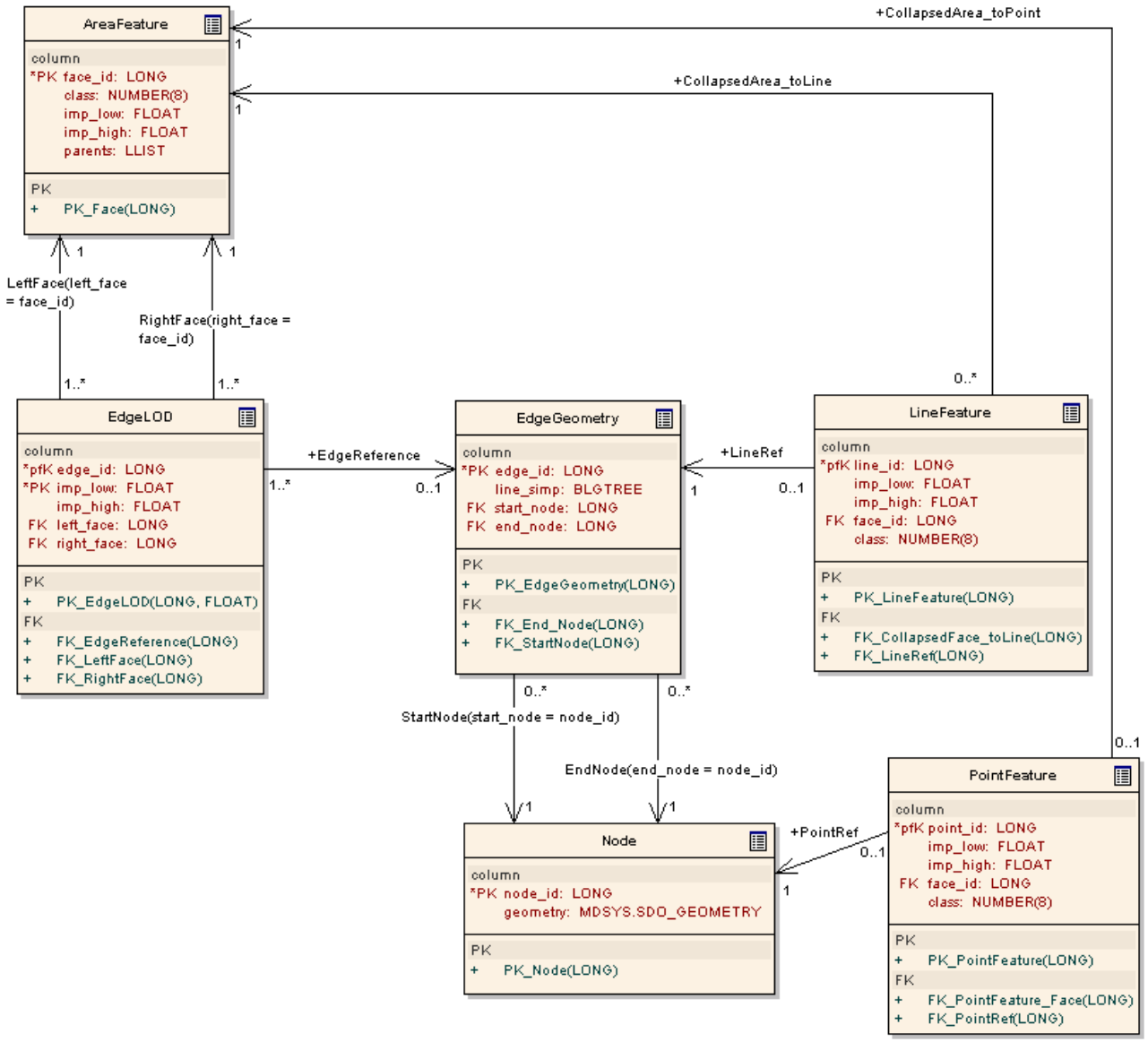

Figure 12: Diagram of tables and relationships to store tGAP information in Oracle Spatial.

Table EdgeGeometry stores the BLG trees of edges in a data type `BLGTREE`. This can be a recursive structure:

```
struct BLGTREE {
    double x_coord;
    double y_coord;
    float tolerance;
    struct BLGTREE *left_node;
```

```
        struct BLGTREE *right_node;
    }
```

For joined edges, `x_coord`, `y_coord` are the coordinates of the common node, `tolerance` is the evaluated tolerance of this node, and `left_node`, `right_node` point to the BLG trees of the edges. For example, for edge 'qr' in Figure 7, `left_node` points to the BLG tree of 'q', and `right_node` points to the BLG tree of 'r'. The `BLGTREE` type does not store explicitly the vertex position in the original edge. When building edge geometry for a given tolerance, vertex positions can be calculated following left and right references in the tree.

Tables storing information about our example generalisation (Figure 1) are given below. Information about faces (Figure 2) is stored in AreaFeature table.

AreaFeature				
face-id	**class**	**imp-low**	**imp-high**	**parents**
1	1	0.00	0.40	7
2	2	0.00	0.55	8, 9
3	2	0.00	0.40	7
4	3	0.00	0.30	6
5	4	0.00	0.30	6
6	3	0.30	0.55	8
7	1	0.40	0.55	9
8	3	0.55	1.20	10
9	1	0.55	1.20	10
10	1	1.20	2.10	

Table 1: Values in AreaFeature table for generalisation of Figure 1.

Information about the edge forest (Figure 4) is stored in tables EdgeGeometry and EdgeLOD. Table 2 shows values in EdgeLOD table.

EdgeLOD				
edge-id	**imp-low**	**imp-high**	**left-face**	**right-face**
a	0.00	0.40	1	0
b	0.00	0.40	3	1
c	0.00	0.55	2	0
d	0.00	0.55	2	0
e	0.00	0.40	1	2
f	0.00	0.30	2	4
g	0.00	0.30	4	0
h	0.00	0.30	5	0
i	0.00	0.30	4	0
k	0.00	0.30	5	4
a	0.40	0.55	7	0
e	0.40	0.55	7	2
f	0.30	0.55	2	6
l	0.30	0.55	6	0
m	0.55	1.20	9	8
n	0.55	1.20	8	0
o	0.55	1.20	9	0
p	1.20		10	0

Table 2: Values in EdgeLOD table for generalisation of Figure 1.

A BLG tree follows the steps of a Douglas-Peuker algorithm, but may not really be needed. A simpler type `LS`, e.g. a sequence of vertices holding info of x, y coordinates, vertex position in the original line, and vertex

tolerance, may do as well. This type can be easier for sending to the client, and for being processed by the client. Type LS can be an array of

```
struct LSvertex {
    double x_coord;
    double y_coord;
    float tolerance;
    int vertex_id;
}
```

Section 4 dealing with the relation with client contains more explanation on this choice.

From the first implementation [3] it was seen that a 3D functional index based on a spatial index on bounding boxes of face, and importance values insured fast access to features. We plan to send edges to the client, therefore we can build a 3D index from a spatial index on bounding boxes of edges, and importance values. The first implementation also showed that storing left-, right-face references and associated importance range is quite expensive. We may leave out face references; store full importance range for edges, e.g. range [0, 0.55] for edge 'f' (see Figure 4); store centroid for each face in DAG to which face information will be attached. Topology should be rebuilt from edges before sending info to the client. Face information can be retrieved from the centroid (via point–inside–polygon).

3.1 Algorithm for building tGAP

The assumption is that we start with a data set that is stored in a left-right topology model, and it forms an area partition. This source data presents the highest LoD. Face and edge information is read from source data tables into arrays in memory. Face array is sorted by importance value, and is always kept in this order after removal or insertion of new faces. The first element of the array is to be processed in each iteration (as it has the lowest importance value). Words in italic indicate functions/procedures.

```
Create Face & Edge list from original data      /* SQL statements to source data tables */
set n = number of original faces

/* Assign importance values to faces */
for i = 1 to n
    imp-low(i) = 0
    imp-high(i) = Importance(i)      /* e.g. the function may be Area(i)*ClassWeight(i) */
end for
Order faces on their importance in Face list

do until one face is left
    Get the list of Face(1) neighbours      /* from the list of Face(1) edges */
    set k = 1      /* counter for neighbours */
    /* Calculate the compatibility with Face(1) */
    for each neighbour j
        comp(k++) = Compatibility(1,j)      /* e.g. as Length(Boundary(1,j)) * ClassSimilarity(1,j) */
    end for
    Decide for splitting or only merging      /* e.g. split if compatibility is the same for all neighbours */
    if splitting
        Split face 1 into parts
        for each part i
            Merge part i with its neighbour m to n++
        end for
    else      /* face 1 will be merged to its best neighbour */
        Find the best neighbour best-nbhd from compatibilities in comp[k]
        Merge face 1 with best-nbhd to n++
```

```
    end if
end do

Merge face a with face b to c
    Build the corresponding edge forest
    imp-high(b) = imp-high(a)      /* imp-high(a) is the importance of the step */

    /* Set parent-child relation */
    pid(b) = c
    pid(a) = c

    /* Fill information for the new face c */
    Class(c) = Class(b)
    imp-low(c) = imp-high(a)
    imp-high(c) = Importance(c)      /* e.g. (Area(a) + Area(b)) * ClassWeight(c) */

    Update table information for faces a, b, and & c
    Remove faces a & b from Face list
    Add face c according to importance order to the Face list
```

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. 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 collection is signalled to the client, which starts building topology for this edge collection. Faces are shown on the client screen. Other edges coming from the server start appearing on the screen. When a signal for (another) complete edge collection is coming, the client starts building topology for the new faces, and visualises them on screen.

Edge geometry can be created in the server from the BLG tree and a tolerance (calculated from scale) given from client. This 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. Thus, an edge geometry is re-sent any time a more detailed shape is needed.

Another possibility is to send for each edge its BLGTREE or its LS, together with the start and end nodes of the edge. In the first 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 [4]. In the second case a sequence LS (stored in EdgeGeometry, or transferred from a BLGTREE in the server side) is sent to the client. 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 edge 'q' (Figure 5) is $\langle(v_3, 0.6); (v_4, 0.5); (v_5, 0.6); (v_2, 0.1)\rangle$ – tolerances are not monotonically decreasing, whereas edge 'r', Figure 7, 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.

References

[1] Tinghua Ai and Peter van Oosterom. Gap-tree extensions based on skeletons. In Dianne Richardson and Peter van Oosterom, editors, *10th International Symposium on Spatial Data Handling*, pages 501–513, 2002.

[2] Martin Galanda. *Automated Polygon Generalization in a Multi Agent System*. PhD thesis, University of Zürich, 2003.

[3] Martijn Meijers. Implementation and testing of variable scale topological data structures. Master's thesis, TU Delft, June 2006.

[4] GiMoDig project. Geospatial info-mobility service by real-time data-integration and generalisation, 2004. Project site: http://gimodig.fgi.fi/.

[5] Peter van Oosterom. *Reactive Data Structures for Geographic Information Systems*. PhD thesis, Department of Computer Science, Leiden University, December 1990.

[6] Peter van Oosterom. 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, 2005.