

# A storage and transfer efficient data structure for variable scale vector data

Martijn Meijers, Peter van Oosterom and Wilko Quak  
Delft University of Technology (OTB—section GIS Technology)  
Jaffalaan 9, 2628 BX Delft, the Netherlands  
`{b.m.meijers,p.j.m.vanoosterom,c.w.quak}@tudelft.nl`

## Abstract

This paper deals with efficient data management of variable scale vector data. Instead of pre-building a collection of data sets on different scales, we create an index structure on the base data set (largest scale data) that enables us to extract a map at exactly the right scale the moment we need it. We present both the classic version of the tGAP (topological Generalized Area Partitioning) data structure for storing our variable scale map, as well as an ameliorated version, both based on topological concepts. We prove that the classic structure needs in a worst case scenario  $O(e^2)$  edges (with  $e$  the number of edges at largest scale). In practice we observed up to a factor 15 more edges in the variable scale data structure. The tGAP structure has been optimized to reduce geometric redundancy, but the explosion of additional edges is due to the changing topological references. Our main achievement finds its roots in the reduction of the number of edge rows to be stored for the ‘lean’ version (by removing the topological referential redundancy of the classic tGAP), which is beneficial both for storage and transfer. We show that storage space for the data set plus the index, is less than twice the size of the original data set. The ‘lean’ tGAP, as the classic tGAP, offers true variable scale access to the data and has also improved performance, mainly due to less data communication between server and client.

## 1 Introduction

There is a growing tendency to focus data management of spatial datasets on the highest level of detail and manage the other levels of detail as data that is automatically derived from this base data set (e.g. Bobzien et al., 2006; Ellsiepen, 2007; Stoter et al., 2008).

Basically there are two methods of managing a data set on different levels of detail (cf. Cecconi and Galanda, 2002): The multi-scale approach and the variable scale approach. The multi-scale approach works by creating several smaller scale versions of the map. Every time a map is needed on a specific scale the most appropriate scale from the pre-defined collection is chosen and displayed. Instead of pre-building a collection of maps on different scales the variable scale approach creates an index structure on the base map that enables you to extract a map at exactly the right scale the moment you need it. This means that when you want a map on a specific scale for a specific region it is constructed for you on the fly. Advantages of the variable-scale approach are that only one dataset needs to be managed and that data can be displayed at any scale.

This paper proposes a new data structure for the management of a variable-scale map product and is an improvement on the tGAP data structure as described in (Van Oosterom, 2005). The idea of the tGAP data structure is to run automatic map generalization on the base data set and instead of storing the result of the generalization on different scales the whole process of the generalization is stored in a tree structure where every node of the tree corresponds to the application of a cartographic generalization operator. Each generalization operator is performed at a specific level of detail (or scale). If a map is needed at a given level of detail the generalization tree structure is used to get the right data at the right level of detail. The implementation of the tGAP structure maintains a valid topological structure on all levels of detail by tracking which nodes, edges and faces are visible on each level of detail. The structure stores the node and face data very efficiently. However there is a lot of redundancy in the way the edges are stored in the model (for details, see Section 3). It turns out that in the worst cases  $O(n^2)$  edges (with  $n$  the number of edges in the largest scale map; see Section 4.4) have to be stored in the tGAP structure (and in practice we observed up to a factor 15 of edges to be stored; see Section 5). This paper describes how this redundancy can be removed without loss of functionality. The new data structure resolved the redundancy for edges so that every edge is stored only once. Saving storage space also implies saving data transfer times as one of the main application areas will be a variable-scale server in a web-based environment.

The classic tGAP structure is offering non-redundant geometric data storage for arbitrary levels of detail. Technically, the problem with the data structure is that too much data storage is needed. Analysis show that this is due to the high number of changing references in the data structure causing new versions of edge representations to be stored. Resulting in an unreasonable growth of edge data in the scale dimension. For 2D geographic information the scale dimension is considered to be the third dimension within the tGAP structure. A 3D spatial index is used to efficiently retrieve a spatial selection at a specific scale.

In this paper we give an overview of some design alternatives we considered to solve to problem of the growing number of redundant edges and present our final solution. The rest of the paper is structured as follows: In section 2 we give an overview of previous work that is done in this area. In section 3 we describe the structure that we wish to improve in more detail and in section 4 we give a few of the alternatives for improvement. Experiments done on these alternatives and our resulting structure are described in 5. Finally in 6 we conclude with a discussion and summarize the most important contributions of this paper and present a number of open problems to be addressed for further improvements.

## 2 Previous work

Most research on the management of variable scale datasets is done on the multi-scale approach where a fixed set of layers is managed. This might stem from the paper map production process where it is very expensive to produce products at different scales. In this digital era it could be possible to manage vector data at arbitrary levels of detail (and disseminate this data via web services). Few solutions are known for this kind of variable scale data access.

Buttenfield and Wolf (2007) have a pyramid structure (called MRVN) that is able to represent a data set at a multiple scales while maintaining topology. To achieve this, all the topological nodes of the original dataset cannot be removed. If the scale range is very big the resulting number of nodes can still be very large making this a method that works for a limited scale range. In Xinlin and Xinyana (2008) the Zoom quad-tree is presented. In the zoom quad-tree all objects of the original dataset are

stored in nodes of the tree dependent on the size of the object. The initially sparse tree is filled with generalized versions of the original features. As described in the paper it is not clear whether the structure can maintain a polygonal partition on the different levels of detail. In the original GAP-tree data structure Van Oosterom (1995) a scale-less structure was described that could manage a polygonal partition. Disadvantage of the structure is that full polygons are stored at various levels of detail making it a very redundant structure.

The first attempt at a fully topological hierarchical structure was done by Vermeij et al. (2003). The structure worked by extending all tables of a standard topological model with two extra attributes (a minimum and maximum scale). Now a generalization algorithm is run on the dataset. The algorithm works by replacing nodes, edges and faces with other nodes edges and faces at a lower level of detail. Instead of deleting the old nodes, edges and faces their max-scale is set meaning that the object is not needed anymore from that scale. By retrieving all nodes, edges and faces that are needed for a specific level of detail a complete topology at that level can be reconstructed. The big disadvantage of this structure is that it produces a lot of redundant data. An ameliorated version (the tGAP structure) was therefore proposed by (Van Oosterom, 2005), for which the first implementation results were described in Van Oosterom et al. (2006). This structure is described in more detail in the next section.

### 3 Classic tGAP structure

This section first summarizes the classic tGAP structure as it will be the basis for the improved version described in the next section. The datasets that are currently supported within the tGAP structure have to be modeled as a two dimensional polygonal map, i.e. it is a partition of the plane in a geometric sense, without gaps and overlaps. The physical storage of the data takes place in a database management system (DBMS) in an extended topological, node-edge-face data structure. The exact table definitions are given in Figure 1.

Each polygon of the map is represented by a topological face (this is a one-to-one relation). The level of detail (LoD) can be regarded as third dimension and represented by the concept of ‘importance’. The importance of objects is based on their size and feature classification. E.g. a large forest area can have lower importance than a small city area. A functional spatial index on a 3D bounding box (bbox) is used to efficiently access the 2D spatial data extended by the third dimension: the importance (or scale) range for which a certain representation is valid.

#### 3.1 Filling the face table

As we want to reduce the LoD for display at smaller scales, we have to generalize our original data. A generalization process reduces the number of polygonal objects, based on the importance. The object that has the least importance is removed first. Plain removal of the object is not allowed, because a gap would exist after this operation. Therefore we let the most compatible neighbor take the space of the object to be removed. Based on the shared boundary length and the feature class compatibility this neighbor is chosen. The merging operation creates a new object. This new object then has a new identity and is given the feature class of the most compatible neighbor. The importance of this object is recomputed (and several different options have been tested for this: e.g. taking the sum of the importance of the two merged objects). This process continues until only one object is left.

During this merging process the importance range for all objects is also created and stored. This range is intimately related to the importance assigned to all faces present at the largest scale. The

```

CREATE TABLE tgap_faces
(
    face_id integer,
    parent_face_id integer,
    imp_low numeric,
    imp_high numeric,
    imp_own numeric,
    feature_class_id integer,
    area numeric,
    bbox geometry
);
(a) Face table

CREATE TABLE tgap_edges
(
    edge_id integer,
    imp_low numeric,
    imp_high numeric,
    start_node_id integer,
    end_node_id integer,
    left_face_id integer,
    right_face_id integer,
    geometry geometry
);
(b) Edge table

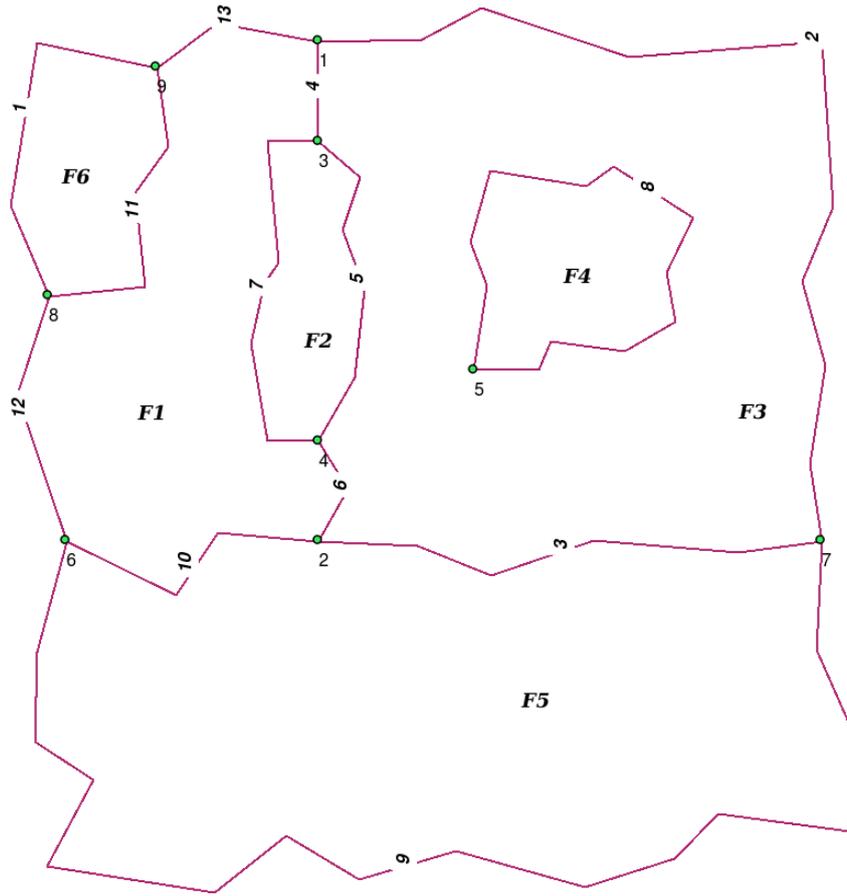
CREATE TABLE tgap_nodes
(
    node_id integer,
    imp_low numeric,
    imp_high numeric,
    geometry geometry
);
(c) Node table

```

Figure 1: Table definitions for the classic tGAP structure.

importance is stored with all the faces as the ‘imp\_own’ attribute that clearly defines the ordering of the generalization process. The importance range (stored with an ‘imp\_low’ and an ‘imp\_high’ attribute for each face) defines the lifespan of objects in the LoD dimension and allows selection of the right objects at an arbitrary LoD (using an importance level for selection, ‘imp\_sel’).

The importance range for the objects is created as follows: The objects used as starting point will be assigned an imp\_low value of 0. The example in figure 2(a) and table 1 shows that the imp\_low value of all original faces (1-6) is indeed 0. Then, in each generalization step, two objects their lifespan will be ended and a new one will be created. In our example, face 1 is the least important face, and is merged with its most compatible neighbor (face 5), a new object (face 7) is formed. Both ended objects are assigned the importance own value of the least important object, named ‘imp\_remove’, as their importance high attribute (face 1 has an imp\_own of 150, this is assigned to both face 1 and 5 as imp\_high value). The new object that is formed in the generalization step will be assigned the sum of the own importance of the two old objects and the imp\_high as the imp\_low value. The resultant of this process is that the sum of all own importance of the original objects is equal to the importance high value of the last remaining object. This means that the sum of importance for all objects valid at any given scale (LoD) in the complete map does not change.



(a) The map of the initial configuration, with  $\text{imp\_sel} = 0$  (note that the nodes, edges, and faces are labeled with their identity)

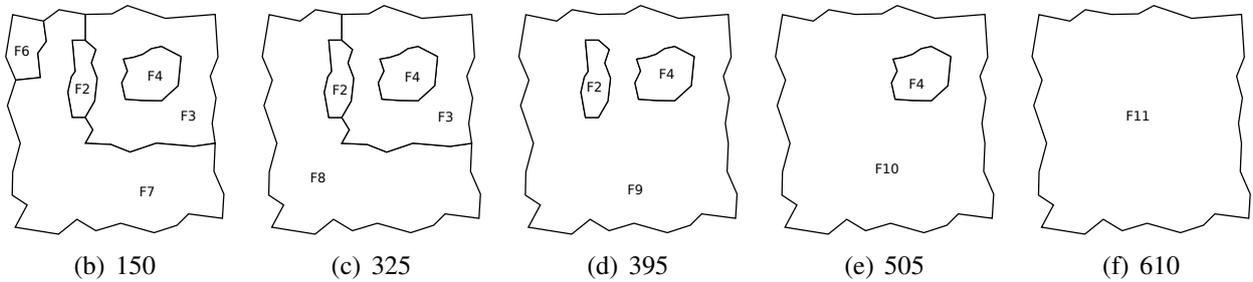


Figure 2: Example map with 6 polygonal regions. Subfigures (b) – (f) show the map at the  $\text{imp\_sel}$  value mentioned in their caption

### 3.2 Filling the node and edge tables

When merging two faces, the life of the edges between the two old faces is ended by setting their  $\text{imp\_high}$  value to the  $\text{imp\_own}$  value of the face that is removed ( $\text{imp\_remove}$ ). The remaining edges are now adjacent to the newly created object, so also these edge versions are terminated (their importance high value is set to  $\text{imp\_remove}$ ) and new, updated versions for those edges are created (with  $\text{imp\_remove}$  as their importance low value). These updated versions get the same identity as before, but with a different left or right face pointer and a new importance low value). In our example edge 10 is removed in the first face merge step, when face 1 is merged to face 5 (the  $\text{imp\_high}$  value of edge 10 is set to 150, see table 3). Edge 11 is an example of an edge that is changed due to the

face_id	parent_face_id	imp_low	imp_high	imp_own	feature_class
1	7	0	150	150	corn
5	7	0	150	750	grass
6	8	0	325	325	grass
3	9	0	395	395	forest
2	10	0	505	505	lake
4	11	0	610	610	town
7	8	150	325	900	grass
8	9	325	395	1225	grass
9	10	395	505	1620	grass
10	11	505	610	2125	grass
11	-1	610	2735	2735	grass

Table 1: The tGAP face table for the sample data set, which is graphically depicted in Figure 2(a) (note there is a bbox and an area value stored, but this is not shown)

node_id	imp_low	imp_high
1	0	2735
2	0	150
3	0	395
4	0	505
5	0	610
6	0	150
7	0	395
8	0	325
9	0	325

Table 2: The tGAP node table. Note that each node has a point geometry, but this is not shown.

change of the neighboring face. This edge was adjacent to face 1, but is after the merge adjacent to face 7. So, a new version of this edge is created.

Furthermore, the nodes that are having only a relationship with two edges after the merge, are as well ended and the incident edges are merged; see the node information from Table 2. A new version for those incident edges is created, with merged geometry based on the geometry of the two old edges. This is shown in our example for the edges 9 and 12 (forming a new edge 14) and the edges 3 and 6 (forming the newly created edge 15). In the classic tGAP structure the edge geometry is represented by a BLG-tree. For leaf edges this is a directly stored version. For non-leaf edges this is a BLG-tree with a new top and references to the two BLG-trees of the child-edges. So no redundancy in the storage of geometry, but the result can be having to trace a lot of references during usage of the structure. An alternative therefore is to create a new (redundant) geometric representation of the merged edge (a non-BLG-tree representation). For this new geometry there are two options: 1. keep all original vertices or 2. keep half of the original vertices (after applying line simplification). Both solutions introduce (controlled) geometric redundancy, but will be easier to use.

### 3.3 Using the structure dynamically

The structure is used dynamically by providing a spatial extent (for the view port) and an importance value (for the LoD). The importance value can be derived from a given extent: A smaller extent means more detail to show and finally a lower importance value for querying the data structures with (imagine a user zooming in, more detail can be shown for all objects). Contrary, if a larger

edge_id	imp_low	imp_high	left_face	right_face	start_node	end_node
1	0	325	-1	6	8	9
2	0	395	3	-1	7	1
3	0	150	3	5	2	7
4	0	150	3	1	1	3
4	150	325	3	7	1	3
4	325	395	3	8	1	3
5	0	395	3	2	3	4
6	0	150	1	3	2	4
7	0	150	1	2	4	3
7	150	325	7	2	4	3
7	325	395	8	2	4	3
8	0	395	4	3	5	5
8	395	505	4	9	5	5
8	505	610	4	10	5	5
9	0	150	5	-1	6	7
10	0	150	5	1	2	6
11	0	150	6	1	8	9
11	150	325	6	7	8	9
12	0	150	-1	1	6	8
13	0	150	-1	1	9	1
13	150	325	-1	7	9	1
14	150	325	7	3	7	4
14	325	395	8	3	7	4
15	150	325	7	-1	8	7
16	325	395	-1	8	7	1
17	395	505	9	-1	1	1
17	505	610	10	-1	1	1
17	610	2735	11	-1	1	1
18	395	505	9	2	4	4

Table 3: The classic tGAP edge table with the example content (Note: a. the repeated versions of edges, due to the left/right reference changes, b. The geometry of the edges is stored, but this is again not shown)

extent needs to be shown, due to a user zooming out, a higher importance value needs to be used for selecting less objects. The mapping between importance and spatial extent is currently done in such a way that it honors the rule of ‘a fixed number of objects’ to be retrieved and shown on the screen. For this mapping the Radical law could have been applied, by which the best number of objects for a certain scale can be calculated (cf. Töpfer and Pillewizer, 1966).

Note that the topological data structures used give more degrees of freedom for modelling what information to store and thus allow us to take more different design decisions than when we would have used plain geometry (e.g. simple feature polygons). It must be noted that this paper focuses on saving storage space. For large data sets this also implies saving time as a more compact storage structure requires less disk pages to be read and less communication between server and client (assuming that the structure itself still supports the most important actions). These considerations are the subject of the next section.

## 4 Design alternatives for a lean tGAP structure

During the design of a more data storage (and transfer) efficient version of the tGAP structure a number of different alternatives were explored, they were labeled with the following symbolic names: `no_lr`, `abox`, `use_tree`. In Van Oosterom (2005) it was already mentioned that less columns in the table structure directly implies less storage (a column less to store), but also indirectly implies less storage – if scale changes are reflected only in a column that is removed then there is no need for a new row with the new value. This was explained by showing how the tGAP edge table which has four edge-to-edge references could be reduced in size by removing two edge-to-edge references and only keeping two edge references (`edge_lr` and `edge_fl`). In the example data sets this resulted in both less columns and less rows. In the implementation reported in (Van Oosterom et al., 2006) all edge-to-edge references were removed, but even in that case the tGAP edge table for a realistic data set still did have up to 15 times more rows than the original edge table (and the theoretic worst case is even  $O(n^2)$  with  $n$  the number of edges at the largest scale). This was mainly due to the changing references to the left and right faces after merging two neighbor faces (and not so much due to merging to existing edges into one new edge). One of the approaches followed was splitting the edge table into two parts: one part with attributes that do not change in the tGAP structure (e.g. geometry, and references to start and end node, if stored) and attributes that do change for different scales/importance values (e.g. left and right face references). However, for the changing part of the edges the number of rows is still the same factor higher, only the fixed part is not repeated, saving some storage space. So the aim is to further reduce the required storage, but without losing performance during the most relevant operations. The most important operation is selecting and visualizing a part of the data set at a certain scale. Another important operation is selecting refinement differences between two scales (for a given part of the map). Further, in the future update operations should be supported (at the most detailed level and then propagated upwards, but this is outside the scope of the current paper).

The selection and visualization of a part of the map at a certain scale, called `imp_sel`, functions as follows: select all faces and edges that overlap the selection rectangle and that have their `imp_low-imp_high` range containing `imp_sel`. Note that these are efficient queries (assuming proper 3D spatial clustering and indexing) and this is all the interaction needed with the database server. Then at the client side some topology processing is done: for every face the relevant edges are selected (based on the left/right face references they contain) and rings are created (and if needed inner-rings are properly included in the outer ring). Due to the fact that the edges are selected based on their `bbox` overlap, not all edges needed to complete the rings of faces partly included in the search rectangle may be present. This is solved by first clipping the selected edges against the selection rectangle (and also splitting the selection rectangle at the intersection points and creating temporary edges). Together, the clipped edges and the temporary edges created from the selection rectangle are sufficient for forming closed loops, which together cover the whole selected area. For sure every ring contains at least a part of an original edge. The left/right information of such an edge provides a reference to the face which can then be colored according to its classification. This is the setting of the use of the tGAP structure and it is clear that the left/right information is needed (for classifying and coloring the faces) despite the fact that it is storage expensive; the ‘row explosion of edges’. Now we are going to discuss our three alternatives, `no_lr`, `abox` and `use_tree`, to make the structure more storage efficient.

### 4.1 `no_lr`

We started out with a very lean topology data structure: no left/right references (as these caused most of the storage overhead), only edge geometry and a point inside a face region (‘spaghetti with

meatballs'-approach); Tables: Nodes (id, location, imp\_low, imp\_high), Edges (id, geometry, imp\_low, imp\_high), Faces (id, mbr, point\_on\_surface, imp\_low, imp\_high). The rings are formed based on topology processing without left/right information. There are three steps: 1. creating rings, 2. assigning island rings to their parent and 3. association of the right identifier with the area (outer ring). Step 1: The procedure starts with an arbitrary edge and then starts forming rings by finding all edges incident with the end (node) coordinates (using the geometry of edges), sorting all incident edges based on angle and then takes the first edge left (for counter-clockwise orientation), this process is repeated until the start edge is reached again and the ring is closed. This procedure is then repeated with the next unused edge and a new ring is formed. The ring production terminates when all edges are used twice (once in forward and once in backward direction). Step 2: some of the rings do not have the expected counter-clockwise orientation, and these correspond to islands in the face. The parent outer-ring can be found by a point-in-polygon test (use arbitrary point from inner-ring and finding the smallest outer ring that contains this point). Step 3: Now all faces with holes are created and have to be assigned an identifier. This is done again with a point-in-polygon test (the point now being the point on surface from the Faces table). For both step 2 and 3 the use of an R-tree (or other type of spatial index) will speed up the point-in-polygon test, building the R-tree once takes  $O(n \log n)$  time and then the repeated searches take  $O(\log n)$  time.

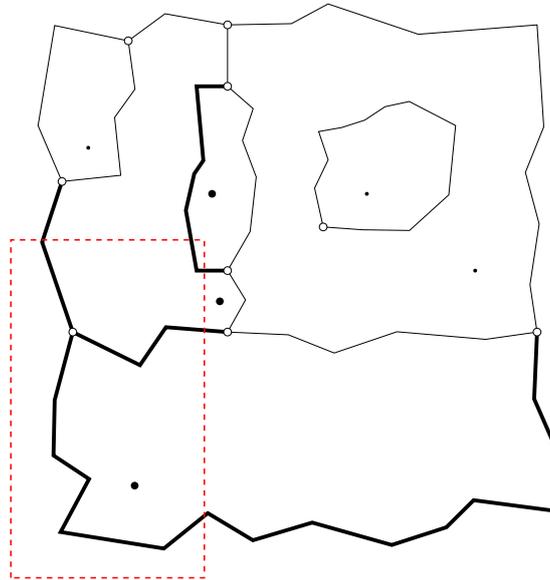


Figure 3: The 'spaghetti with meatballs' approach. The retrieved edges (overlapping with the selection rectangle in dashed lines) are given with the thickest lines. After clipping, 3 rings are formed, but the two rings at the top of the selection rectangle cannot be labelled with the correct face information as the point on surface for these faces is outside the formed ring.

This approach does work for having a complete extent of area partition within the view port while visualizing. It does not work well when clipping the data: areas cannot be reconstructed any more, without having a complete set of edges. An option is to clip the selected edges again (as described above). The result is that now areas can be created covering the selection rectangle. However, faces crossing the boundary might have their point on surface outside the rectangle (and therefore the area can not be identified. There might be some solution to go back to the database server for each unidentified area, but this is both a non-trivial query and time expensive as it has to be repeated for every unidentified area.

## 4.2 abox

In an attempt to solve the identification of the clipped areas, the adjacency box (Van Oosterom and Vijlbrief, 1994), or abox for short, instead of the bbox of edges was proposed for selection. The abox of an edge is the union of the bbox of the faces left and right of the edge. The result is that more edges are selected based on the abox, but for sure these are enough to completely reconstruct all faces in the selected rectangle. However, in order to have the aboxes available in the edge table they have to be maintained (stored). Due to merging of faces in the tGAP structure also the aboxes have to be updated. Actually this is then exactly the same increase in rows as what would be obtained by maintaining the left and right face references. So, no real storage reduction, rather the opposite as the abox will take more storage space than the left and right reference. The advantage of the abox solution is that it allows easier reconstruction of faces at the client side resulting in full unclipped areas. In theory the explicit storage of aboxes might be avoided by introducing them in a view (which uses a function to compute the abox). But again this is non-trivial without the left and right references. Therefore we concluded that this was also not the ideal solution and continued investigating another alternative with fewer drawbacks.

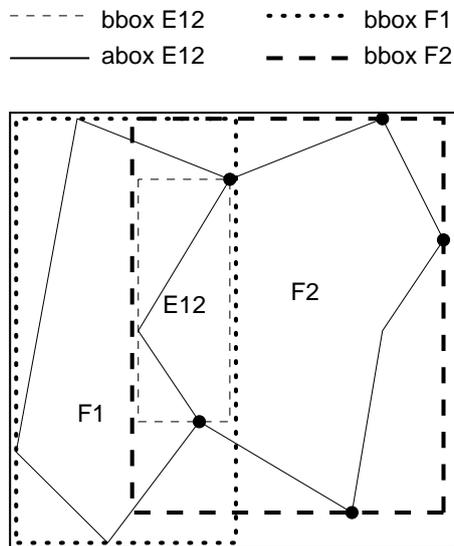


Figure 4: Adjacency box (abox)

## 4.3 use\_tree

Looking at edges that are changing due to changes in the left and right side information (and not in the edge geometry), we might consider merging the rows related to the same edge in one row. This results in no change for the geometry, start and end nodes, and id attributes. The `imp_low` and `imp_high` attributes contain the union of all imp ranges of the edge (which are per definition adjacent ranges). The next question is what to do with the differences in left and right references? Store the left/right reference corresponding to the lowest imp range or to the highest imp range? Take for example edge 4 in Table 3 and 4: storing the right face reference corresponding to the lowest imp range  $[0 - 150)$  would imply a reference to face 1, and storing it related to the highest imp range  $[325 - 395)$  would result in a reference to face 8. It was decided to store the left and right face references related to the lowest imp-range, for reasons that will be explained below when assigning the proper identity to the created areas. Anyhow, just storing only rows for edges that are really new

(because these edges are merged) saves a lot of storage (rows) as will be explained in section 4.4 (the number of rows is for sure always below a factor 2 as edges are merged pair wise). The left/right information and the tGAP face-tree can then be exploited to properly identify the areas at a certain importance level (scale). With this solution we have combined both the requirement to be storage efficient (as the factor 15 of records in the edge table is solved), while still having an efficient solution for the most relevant operation.

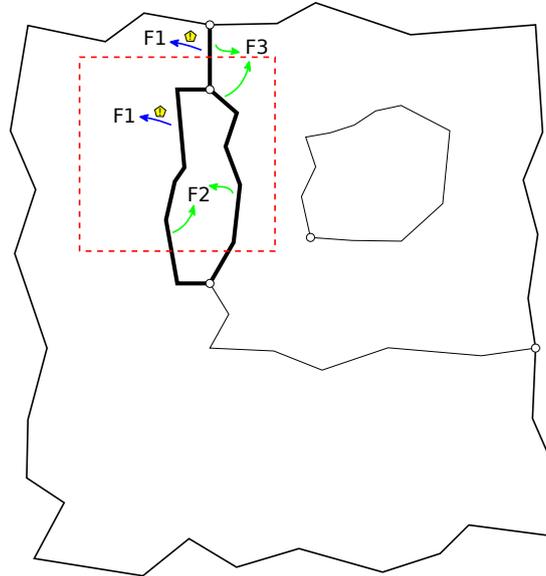


Figure 5: Rewriting of face-id's with the use\_tree variant. Edges are retrieved, at `imp_sel = 330`, based on their bounding box and the selection rectangle (dashed). After clipping, only the thickest lines are used for forming rings. The most-left ring is formed based on edge 4, 7 and temporary edges stemming from the selection rectangle. Both edges 4 and 7 do not point to the correct neighboring face and rewriting has to take place (face 1 is rewritten using the tGAP face tree as face 8).

The identification of areas in a given search rectangle of a specified importance level `imp_sel` proceeds as follows. All edges are retrieved a. based on a selection rectangle and b. having an `imp` range that includes `imp_sel`. The faces are also selected based on these two criteria. Then the clipping is applied to the edges and rings are created as described above and inner-rings are again assigned to outer-rings. During the creation of rings the left/right information is used to find the identity of the face. As the edges carry the left/right information of the lowest `imp`-range (which may be below the requested `imp_sel`) not all edges directly have a pointer to the correct face (that is at the requested `imp_sel` level). In many cases however there will be at least one edge with the proper (w.r.t. `imp_sel`) left/right information and this is then indeed the identity of the area. In some cases this information is not present (1. when this edge is outside the selection rectangle, 2. when an island is not yet merged with its parent). In these cases the referred face (and the corresponding edge) with the highest `imp_low` level is used as start in the tGAP face-tree and the tree is traversed upwards until the face identifier at the right `imp` level is found. The final layout of data structure is (again) based on topology and has the following tables: Nodes (`id`, `geometry`, `imp_low`, `imp_high`), Edges (`id`, `start node`, `end node`, `left face`, `right face`, `geometry`, `imp_low`, `imp_high`), Faces (`id`, `feature class`, `bounding box`, `imp_low`, `imp_high`, `imp_own`); see Table 1, 2 and 4 for the sample data set in Figure 2(a), the sample map, and Figure 6, a visual representation of the tGAP face-tree.

The drawback of using the tGAP face-tree is that this tree is not present at the client side (after the two face and edge selection queries). An efficient solution is to send one third query to the server

requesting the ‘rewriting’ of the face-id’s which correspond to a too low imp level and get back face-id’s that correspond with imp\_sel. An easier solution is not to draw these faces at all: the drawback is of course that white spots will occur on the map (most often near the boundaries of the selection rectangle).

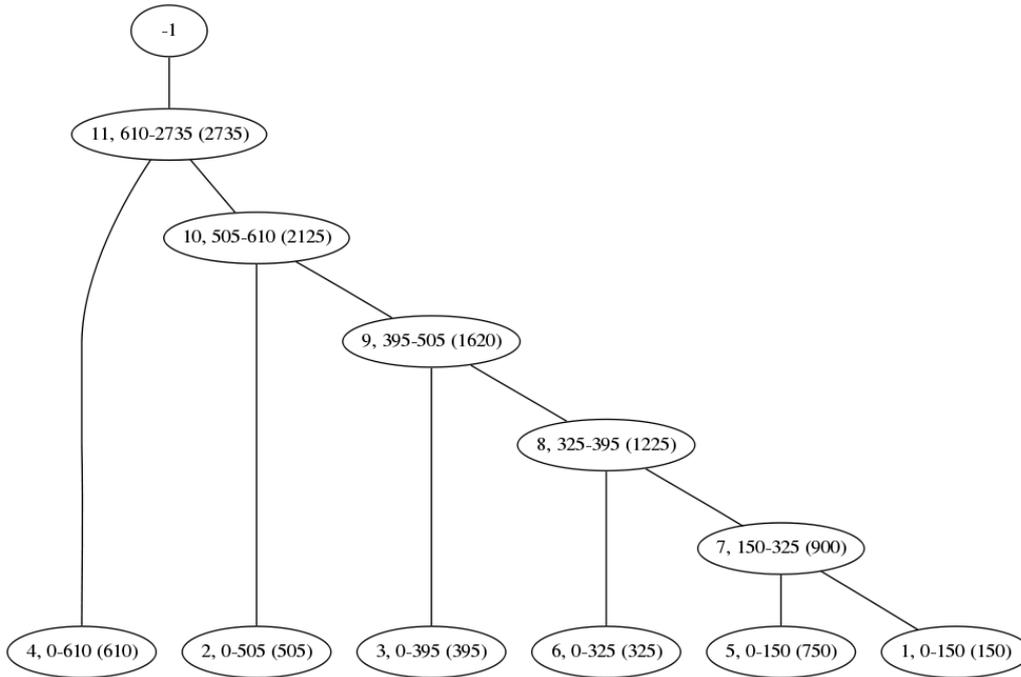


Figure 6: The tGAP face-tree, corresponding to the data set of Figure 2(a)

edge_id	imp_low	imp_high	left_face	right_face	start_node	end_node
1	0	325	-1	6	8	9
2	0	395	3	-1	7	1
3	0	150	3	5	2	7
4	0	395	3	1	1	3
5	0	395	3	2	3	4
6	0	150	1	3	2	4
7	0	395	1	2	4	3
8	0	610	4	3	5	5
9	0	150	5	-1	6	7
10	0	150	5	1	2	6
11	0	325	6	1	8	9
12	0	150	-1	1	6	8
13	0	325	-1	1	9	1
14	150	395	7	3	7	4
15	150	325	7	-1	8	7
16	325	395	-1	8	7	1
17	395	2735	9	-1	1	1
18	395	505	9	2	4	4

Table 4: The lean tGAP edge table with the example content (note the geometry/line is not displayed but present in the structure)

## 4.4 Theoretical numbers for faces and edges

In the previous section we sketched a more optimal solution for storing data in the edge table. Here, we continue our investigations by finding the theoretical upper bounds after filling the data structures for both the classic and the lean variant. These bounds are expressed in numbers of edges ( $e$ ) and faces<sup>1</sup> ( $f$ ) present in the original dataset.

**Lemma 4.1.** *The number of total faces stored in the tGAP structure is, after the generalization process, equal to:*

$$2 \cdot f - 1$$

*Proof.* The generalization process starts with  $f$  original faces. Merging can be executed until we have only one face left. This means we can merge  $u$  times, with  $u = f - 1$ . Each time we merge two faces, we add 1 new face to  $f$ . In total we add  $u$  times a face to  $f$ . The total number of faces will thus be  $u + f$ , or, expressed differently:

$$2 \cdot f - 1$$

□

**Lemma 4.2.** *The total number of edges in the classic tGAP structure, that is, filled with the original method (generating all intermediate edge versions), is at most:*

$$\sum_{i=0}^{f-1} e - i$$

*Proof.* Faces are merged in  $f - 1$  steps. Faces that are neighbors are adjacent in, at least, one edge (due to the planar map criterion). With each merge step thus at least one edge will disappear. The worst case is that in every generalization step all remaining edges will be duplicated due to new left/right references. These observation lead to Lemma 4.2. □

**Corollary 4.3.** *The total number of edges in the classic tGAP structure, that is, filled with the original method (generating all intermediate edge versions), can be quadratic:*

$$O(e^2)$$

*Proof.* Assume a configuration (similar to the one shown in Figure 7) with one big face (described by one big edge) containing many small islands (small faces, each one described by one edge). Then in the summation of Lemma 4.2 it is clear that  $f = e$  and this results in a total of  $e \cdot (e + 1)/2 = O(e^2)$  edges. □

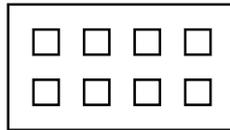


Figure 7: A worst case initial configuration

Our new, lean approach performs significantly better in this respect:

<sup>1</sup>Numbers for faces here do *not* include the concept of a universal face

**Lemma 4.4.** *The total number of edges stored in the tGAP structure, filled with the new ‘use\_tree’ method, is dependent on the number of original edges and faces and is at most:*

$$2 \cdot e - f$$

*Proof.* All original edges will be present once in the output. The merging of edges is what brings new edge versions.

Suppose this edge merging is performed with all start edges as input, as follows: two edges will be merged at a time, until 1 edge is left. The resultant of this process is then one large polyline with self-intersections. The total number of edges in the output will then be at most two times the original number of edges minus 1 (cf. Lemma 4.1).

However, in each generalization step, to merge two faces, at least one edge has to be removed, i.e. the number of edges to be removed is the number of faces minus 1 (as that is the amount of merges that will take place). Taking both steps into account, results in a number of edges that is equal to:

$$(2 \cdot e - 1) - (f - 1) = 2 \cdot e - f$$

This is a worst case estimate, as in each merge step more than one edge might be removed. □

## 5 Experiment and results

To judge whether our theoretical investigations described above would yield valid results in practice, we implemented both variants (classic and lean) of filling the tGAP structures using PostgreSQL<sup>2</sup> extended with PostGIS<sup>3</sup> (for the geometrical attributes) as DBMS. For filling the tGAP structures in the DBMS with our generalization procedure of merging faces and for retrieving and visualizing the data from the DBMS, we wrote some scripts using the Python<sup>4</sup> programming language. Table 5 highlights the number of faces and edges for the original data, the amount of data after using the classic variant and for the lean variant of filling the structures.

To verify the lemma’s from section 4.4, we started by creating two artificial test data sets (1 and 2). It is clear that the number of faces follows Lemma 4.1 in all cases, independently from which filling variant is used. Further, it is also clear that our concerns with respect to the duplication of edge rows are valid: To see whether the upper bound for the number of edges could exist in practice, we created a data set (set 2) consisting of one polygon containing 2500 islands polygons. Each polygon was described with one line, resulting in 2501 faces and 2501 edges. In practice, this data set can occur when an archipelago is mapped and in which all islands are merged to the surrounding ocean. The factor for the classic variant of filling is an abominable result (on average each edge is duplicated 1251 times, that is indeed  $O(e^2)$ ), especially compared to the lean version (in which only the original edge versions are present once).

Besides artificial data sets we also used some data sets containing real world data. That the factors are higher for the sets 5 and 6 compared to the factors for 3 and 4, is explainable by the fact that the last two sets do not contain any island polygons, while set 5 and 6 do contain some polygons with a

---

<sup>2</sup>[www.postgresql.org](http://www.postgresql.org)

<sup>3</sup>[www.postgis.org](http://www.postgis.org)

<sup>4</sup>[www.python.org](http://www.python.org)

Data set	Original Faces	tGAP faces	Original Edges	Edges Classic (increase factor)	Edges Lean (increase factor)
Artificial data					
Set 1	6	11	13	29 (2.2)	18 (1.4)
Set 2	2501	5001	2501	3128751 (1251)	2501 (1.0)
Real world data					
Set 3	525	1049	1984	11091 (5.6)	2975 (1.5)
Set 4	5537	11073	16592	77585 (4.7)	26787 (1.6)
Set 5	50238	100475	178815	2663338 (15)	264950 (1.5)
Set 6	173187	346373	426917	3544232 (8.3)	630944 (1.5)

Table 5: Number of faces and edges for the different test data sets. Numbers are shown for the original data, the data after using the classic variant of filling (i.e. edge version duplication) and for the lean variant (only each first edge version is stored). Both data set 1 and 2 were created artificially. The data sets 3 – 6 contain real world data. Data set 3 and 4 both contain land cover data. Set 5 contains cadastral parcels and data set 6 contains topographical data.

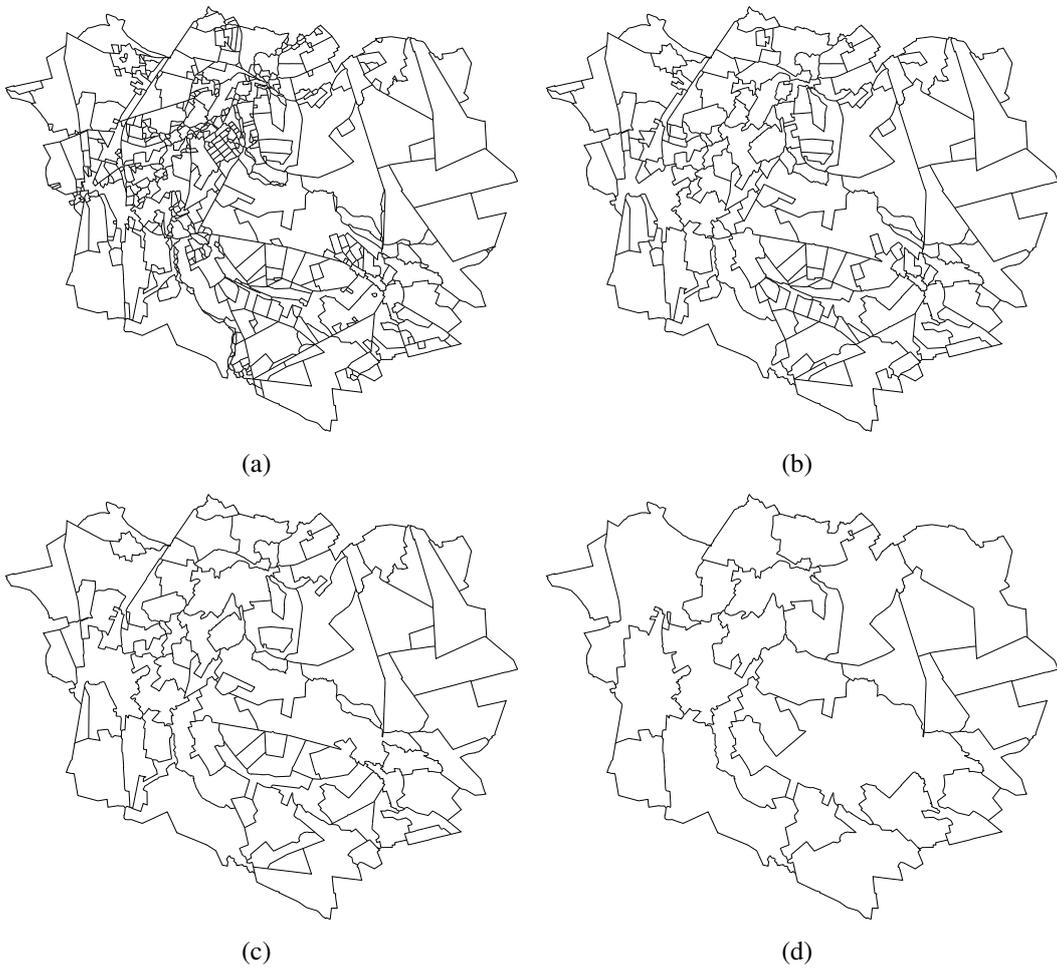


Figure 8: Data set 3, visualized with different `imp_sel` values.

few hundred islands; Filling the structures in the classic way leads then to even more duplicated edge rows. Although the theoretical upper bounds are, by far, not met by these data sets, the factors of the classic filling variant are still high (and we suspect that this will even be worse for larger data sets), while our new variant significantly performs better.

## 6 Conclusion and Discussion

With our design and implementation exercise, we learnt the following lessons: First, our ‘use\_tree’ alternative performs a lot better when looking at the storage part, compared to our initial solution, not only in theory, but certainly also in practice. Reducing storage is not the only achievement here; The reduction in the number of edge rows will be very beneficial for the case when the tGAP structure will be used in a web service environment and data is sent (progressively, using increments) to a client. Second, we designed a structure that now has a better trade off between storage and calculation-when-needed than before (much less data is to be stored and transferred, but with our lean alternative sometimes it is necessary to perform a lookup operation of the correct neighboring face). Third, we support – with less than twice the original dataset size – all intermediate scales for visualization at an arbitrary scale.

Irrespective of these accomplishments, we also realize that our main contribution is currently based on one generalization operation (the merging of objects) and based on the heuristics this operation brings (geometry is always removed and gradually becomes less). The field of map generalization however offers more operations, like line simplification, collapse/splitting of area objects, displacement and typification of groups of objects, to name a few. Some of these operations, by definition, introduce new geometry (e.g. splitting of objects will introduce new boundaries). An optimal solution for a data structure, in terms of data storage, has thus to take into account the heuristics of these generalization operations. Therefore our investigations will continue and topics we would like to focus on in the (near) future include:

- Topological correct line simplification (taking into account the neighborhood, while performing line simplification and preventing (self-)intersections that cause a change in topology, similar to what is described in Saalfeld (1999) and Bertolotto and Zhou (2007)). Creating and using data structures for variable scale access to this geometry (e.g. finding an alternative algorithm for filling the BLG-tree structure, currently Douglas-Peucker is used) is another topic that deserves attention.
- More operations for the (currently simplified) generalization process; As a first step, we would like to, instead of merging, allow splitting, probably based on a triangulation (cf. Bader and Weibel, 1997), and experiment what happens when using such a split operator with weights for all neighbors, until no further splitting is possible.
- Inclusion of more and different semantics in order to take better decisions which generalization operator to choose (instead of the current ‘one fits all’ approach); e.g. apply a different generalization operator for infrastructure type of objects than for other terrain objects.
- Making the structure dynamic: perform updates at the largest scale (and propagate these upwards in tGAP structure).

## References

Bader, M. and Weibel, R. (1997). Detecting and Resolving Size and Proximity Conflicts in the Generalization of Polygonal Maps. In *Proceedings of the 18th International Cartographic Conference.*, pages 1525–1532, Stockholm.

- Bertolotto, M. and Zhou, M. (2007). Efficient and consistent line simplification for web mapping. *International Journal of Web Engineering and Technology*, 3(2):139–156.
- Bobzien, M., Burghardt, D., Petzold, I., Neun, M., and Weibel, R. (2006). Multi-Representation Databases with Explicitly Modelled Intra-Resolution, Inter-Resolution and Update Relations. In *Proceedings Auto-Carto 2006*, Vancouver.
- Buttenfield, B. and Wolf, E. (2007). “The road and the river should cross at the bridge” problem: Establishing internal and relative topology in an mrdb. In *Proceedings of the 10th ICA Workshop on Generalization and Multiple Representation 2-3 August 2007*, Moscow, Russia.
- Cecconi, A. and Galanda, M. (2002). Adaptive Zooming in Web Cartography. In *Computer Graphics Forum*, volume 21, pages 787–799. Blackwell Synergy.
- Ellsiepen, M. (2007). Partial regeneration and its requirements on data structure and generalization functions. In Kremers, H., editor, *Proceedings 2nd ISGI 2007: International CODATA symposium on Generalization of Information*, Lecture Notes in Information Sciences, pages 72 – 84, Germany. CODATA.
- Saalfeld, A. (1999). Topologically Consistent Line Simplification with the Douglas-Peucker Algorithm. *Cartography and Geographic Information Science*, 26(1):7–18.
- Stoter, J., Morales, J., Lemmens, R., Meijers, M., Van Oosterom, P., Quak, W., Uitermark, H., and van den Brink, L. (2008). A data model for multi-scale topographical data. In *Headway in Spatial Data Handling 13th International Symposium on Spatial Data Handling*, pages 233–254.
- Töpfer, F. and Pillewizer, W. (1966). The principles of selection, a means of cartographic generalization. *Cartographic Journal*, 3(1):10–16.
- Van Oosterom, P. (1995). The gap-tree, an approach to “on-the-fly” map generalization of an area partitioning. In Müller, J., Lagrange, J., and Weibel, R., editors, *GIS and Generalization, Methodology and Practice*, page 120–132. Taylor & Francis.
- Van Oosterom, P. (2005). Scaleless topological data structures suitable for progressive transfer: the gap-face tree and gap-edge forest. In *Proceedings Auto-Carto 2005*, Las Vegas, Nevada. Cartography and Geographic Information Society (CaGIS).
- Van Oosterom, P., de Vries, M., and Meijers, M. (2006). Vario-scale data server in a web service context. In Ruas, A. and Mackaness, W., editors, *Proceedings of the ICA Commission on Map Generalisation and Multiple Representation*, pages 1–14, Paris, France. ICA Commission on Map Generalisation and Multiple Representation.
- Van Oosterom, P. and Vijlbrief, T. (1994). Integrating complex spatial analysis functions in an extensible gis. In *Proceedings of the 6th International Symposium on Spatial Data Handling*, pages 277–296, Edinburgh, Scotland.
- Vermeij, M., Van Oosterom, P., Quak, W., and Tijssen, T. (2003). Storing and using scale-less topological data efficiently in a client-server dbms environment. In *7th International Conference on GeoComputation*, Southampton.
- Xinlin, Q. and Xinyana, Z. (2008). Multi-representation geographic data organization method dedicated for vector-based webgis. In *Proceedings of the XXXVI congress of ISPRS*, volume Part B4 Commission IV, pages 815–819.