SPATIAL STRUCTURES TO SUPPORT AUTOMATIC GENERALISATION

Nicolas Regnauld
Research & Innovation, Ordnance Survey®, Great Britain. E-mail: nicolas.regnauld@ordnancesurvey.co.uk.

ABSTRACT

Automating the cartographic generalisation process has been the focus of much research. It has repeatedly been reported that one critical task is to model the spatial relationships between map features to understand their meaning and importance for the target map. This paper will review the different spatial structures that are usually used by researchers to support their automatic generalisation solutions.

This paper then proposes a model for building and storing a proximity graph between features. This model has been designed for supporting generalisation. This means that the graph works as a logical structure that links the features together by references, allowing fast analysis. Different types of graphs are presented. An efficient technique is described to build a triangulation on a set of features, as well as the tools to derive the other types of graphs from it. These tools have been implemented, the results are presented in the paper.

1. INTRODUCTION

Automatic generalisation for creating cartographic products from geographic databases has proved a difficult task. One of the reasons is that the geographic databases usually describe geographic features, but not the spatial relationships between them. These spatial relationships are critical for understanding the role or meaning of each feature in the map, and therefore determine how it should be generalised. These spatial relationships need therefore to be modelled and extracted, and possibly stored for use during the generalisation process.

Most of the current versions of the commercial GIS do not provide good facilities to represent and compute these spatial relationships. The research team at Ordnance Survey has therefore designed and implemented a model for storing these relationships, and the tools to extract them and query them. The aim is to have a generic model that can support our research in automatic generalisation and can be plugged to different development platforms. This paper is also intended to show what functionalities we would like to see available in commercial GIS to support the development of dedicated applications (generalisation being only one of them).

This paper starts by reviewing the spatial structures which have been used for research into cartographic generalisation. Then we propose a data model which can support basic graphs and two special types of graphs (proximity graphs between geographic features and triangulations). The third section describes the functionalities which have been added to the model to build the different types of graphs and query them. In the model proposed the geographic features are fully linked to the spatial model, so that the spatial queries can be addressed to the geographic features, then the answer is computed by exploring the underlying spatial structure. The model has been implemented in Java and connected to Clarity (Neuffer, Hopewell, and Woodsford 2004), a platform developed by Laser-Scan (UK) and dedicated to support generalisation. Results are discussed in the last section.

2. IMPORTANCE OF SPATIAL STRUCTURE FOR GENERALISATION

It has been acknowledged for a long time that automating the process of cartographic generalisation requires a phase of analysis to detect the geographic structures or phenomena which are relevant to the target map. One of the early conceptual frameworks for generalisation proposed by (Brassel and Weibel 1988) already includes a phase of “structure recognition”. These structures are identifiable by semantic information attached to the features, and the spatial relationships between them. Spatial relationships are usually not explicitly stored in the initial database and need to be computed. They usually include some elements of proximity between features, and can be quite computationally expensive to derive. If we consider the fact that this information is heavily used during the generalisation process, it is obvious that these relationships should be stored, at least temporarily, into what we call spatial structures. Spatial structures are data structures that allow to represent spatial relationships between geographic features. They have been heavily used in automatic generalisation, and take a lot of different forms. Most of them rely on a graph structures, but several types of graphs are used for different purposes. We list below a few of them, with examples of how they have been used.
- **The Delaunay Triangulation**

A Delaunay triangulation is a graph, i.e. a set of nodes connected by edges. Edges of a triangulation do not cross and no new edge can be added to the graph. In addition, the triangles formed by the graph have the property that the inside of their circum circle does not contain any other node of the graph. An example is shown in Figure 1, where the red circles are the initial nodes, and the dark edges the Delaunay triangulation are built on them. The Delaunay triangulation is very useful to represent proximity relationships (distance and direction).

![Figure 1: Triangulation and Voronoi diagrams, created using the applet](http://www.cs.ubc.ca/~ssueda/applets/delaunay/delaunayApplet.html)

In the context of cartographic generalisation, we are not only interested in the proximity between points, but also between features. This often means that we do not want the edges of the triangulation to cross the outline of the geographic features. For example, Figure 2a shows a case where an edge crosses the middle building. This in effect creates a direct neighbour relationship between the top and bottom buildings, while there is an obstructing feature between them. This can be avoided by constructing either a conforming triangulation (Figure 2b), where vertices are added to the outline of the features until no edge of the triangulation crosses a feature outline, or a constrained triangulation (Figure 2c), where all the edges of the feature outlines are forced to be edges of the triangulation. Note that in the constrained triangulation, the circum circle property has been lost.

![Figure 2: default Delaunay triangulation (a), conforming (b), constrained (c)]

All these types of triangulation have proved useful for different types of application. (Ruas 1998) uses a standard Delaunay Triangulation to support a displacement method. It is computed only on the centroids of the buildings, so there is no need to refine the graph. The triangulation is used to detect the neighbours, and then the true direction and distance between neighbouring features is computed from the full geometry of these features. Constrained triangulation is more commonly used. (Jones, Bundy, and Ware 1995) use it to support several generalisation operators: collapse (by generating the skeleton of ribbon shaped objects), amalgamation and displacement. They also use it to detect conflicts and monitor the topological consistency of the features. Constrained triangulation is also used to support the displacement methods developed by (Sester 2001) and (Højholt 2000). (Bader and Weibel 1997) have opted for a conforming triangulation to support their generalisation operators for polygonal maps. They have used it to detect narrow sections of polygon and dissolve them with the neighbouring polygons. They also use it to support amalgamation and displacement. They argue that the conforming triangulation provides more hooks for their algorithms to use.

- **The Voronoi diagram**
The Voronoi diagram is the dual graph from the Delaunay triangulation. A Voronoi diagram constructed for a set of objects is a tessellation where each cell delimits the space which is closer to the contained object than any other object. Figure 1 shows an example of a Voronoi diagram built on a set of points (in light grey). Voronoi diagram has been less widely used than the Delaunay triangulation in the context of cartographic generalisation studies, despite being well adapted to spatial analysis purposes. (Gold and Yang 1996) present a structure called the Voronoi Map Object Tree, which allows to describe a map at several levels of details. It provides an efficient framework for doing spatial analysis operations. (Hangouët and Djadri 1997) also use the Voronoi diagram for spatial analysis purposes to support generalisation. They show, for example, how it can be used to characterise conflicts or evaluate local density. Voronoi diagrams have also been used directly to support generalisation operations. (Ai and van Oosterom 2002) presents a displacement method which is based on the Voronoi diagram. Voronoi diagrams and Delaunay triangulations can be derived from each other. The triangulation records more directly the neighbourhood relationships between objects, while the Voronoi diagram provides explicitly the “influence” zone of each object. (Li, Yan, Ai, and Chen 2004) makes a combined use of both: they use a triangulation to detect clusters of buildings, and the Voronoi diagram to give an spatial extent to each cluster. It is then used to perform amalgamation.

- Transport graphs
We distinguish transport graphs from the other types of graphs (any node-edge base structure, like for example a triangulation), because their edges are comprised exclusively of geographic features, like roads, paths, rivers, railways. In these graphs, the topology of the graph directly relates to the connectivity between the initial map features. These graphs are used to support the generalisation of networks. For example, the graph can be used to prevent the simplification of the network creating disconnected sub networks, or increasing dramatically the cost of travelling from one place to another. The graph structure also allows to specify this cost as required for the target map (it can be based on distance, travelling time, cost, width of the road, etc, or any combination of these). The potential of graph structures to support generalisation has been discussed in (Mackaness and Beard 1993), and used to support road network generalisation in numerous studies, for example (Mackaness 1995; Thomson and Richardson 1999). It has also been used on a more specific problem to simplify complex junctions (Mackaness and Mackechnie 1999). Graphs are also very useful for qualifying hydrology networks (Horton 1945), or the structuring lines of the relief (Weibel 1992).

- Other graphs
Graphs can be used for modelling the proximity relationships between any features. They do not have to form a triangulation, and they do not have to follow existing linear features. (Regnauld 2001) builds a minimum spanning tree (subgraph without cycles, minimising costs) to model the proximity relationships between buildings and uses it to detect groups aligned along roads. They are then used to support a typification operation.

Other spatial structures can be used (such as those based on space partitions other than the Voronoi one), but from the literature review, it seems that the graph structure is behind lots of algorithms and spatial analysis tools. The Delaunay triangulation, which is a particular type of graph, is especially widely used. Although the value of such structures is very well known, they are absent from most of the commercial GIS, or at least not generic enough or not rich enough to be exploited to their full potential. In the next sections, we propose a data model and a prototype that provides a rich tool to construct and exploit a Delaunay triangulation, and proximity graphs.

3. DATA MODEL TO REPRESENT GRAPHS
We propose a data model that allows any graph to be stored. This model is obviously only one way of representing a graph. We use object-oriented concepts, and have made the choice of privileging process efficiency over storage space. This means that we store a lot of references between the objects of the graphs that could be computed from others. This section only describes the classes used and their attributes. Their functionalities (constructors and methods in an Object-Oriented context) are described in the next section.

We use three classes to store a basic graph (the classes Graph, Node and Edge). Then we specialise these classes to store different types of specific graph. We propose here a proximity graph and a triangulation, but this can be extended at will. The triangulation allows a triangulation on a set of vertices to be stored, and also keeps the links between the nodes and edges of the triangulation and the geographic features on which they lie. The proximity graph allows a graph to be constructed directly on the geographic features (each geographic feature is either represented by a node or an edge in the graph). The proximity graph is therefore useful to represent proximity or neighbourhood relationships between features, and it can be deduced from a triangulation.

Figure 3 shows the classes and the inheritance links between them, used to store the three types of graphs. The basic graph is represented by the classes with a black outline, the proximity graph by those with a red outline, and the triangulation by those with the blue outlines. The function and attributes of these graphs are detailed below.
Basic graph
A basic graph is represented using three classes, detailed below, and illustrated in Figure 4.

- **Graph.** A graph is a list of nodes and edges connected with each others. It has two attributes: **NodeList** contains the list of nodes of the graph and **EdgeList** contains the list of edges. Ideally, these lists would contain direct references to the edge and node objects, but if the model does not allow it, then identifiers can be used instead.

- **Edge.** The class Edge is used to connect nodes of the graph. It provides two attributes: **node1** and **node2** to specify the nodes it connects (ideally using direct references to the node objects). It can support directed graphs, in which case **node1** should be used to store the start node and **node2** the end node. The attribute **FeatureList** is a list of values allowing to identify which geographic features this edge represents. Ideally, it would contain a list of references to these objects. It can be null if the edge of the graph does not relate to any geographic feature.

- **Node.** The class Node is used to store the nodes of the graph. It provides four attributes. **X** and **Y** are used to store the coordinates of the node. **FeatureList** is a list of values identifying which geographic features this node represents. Ideally, it would contain a list of references to these objects. **EdgeList** contains the list of edges that are connected to this node.

Figure 3: hierarchy of object classes used to represent different types of graphs

Proximity graph
A proximity graph is intended to represent clusters of neighbouring features. As a set of features can be decomposed hierarchically (split the group into clusters, then each clusters into sub clusters, etc.), we have enhanced the basic classes so that the hierarchical decomposition can be represented. When a graph is split (by removing some edges), one new graph is created for each disconnected subgraph. The removed edges stay in the original graph, the other ones and the nodes are split between the subgraphs. To split the graph, we use the

Figure 4: Example of a basic graph, and example of attributes for one node and one edge
weight of the edges to decide which ones to remove. So we have added attributes to the edges of a ProxiGraph in order to store a meaningful distance between the two features connected by the edge. Figure 5 and Figure 6 illustrate the additions which have been made to the classes used to store a ProxyGraph. These classes are described below:

- **ProxiGraph.** The ProxiGraph class extends the Graph class, and provides three additional attributes. FeatureList provides the list of geographic features represented by the nodes and edges of the graph. Subgraphs contains the list of subgraphs after segmentation of this graph. Each of them are instances of the class ProxiGraph. The reverse link is kept using the attribute ParentGraph. Both SubGraphs and ParentGraph can be null.

- **ProxiNode.** The class ProxiNode is an extension of the class Node. It contains one additional attribute GraphId which specifies the graph top which the node belong. This had to be added because we are now working with a hierarchy of graphs rather than dependant graphs.

- **ProxiEdge.** The class ProxiEdge is an extension of the class Edge. Like the ProxiNode class, it needs the GraphId attribute to specify which graph it belongs to. It also needs additional information to store the true distance between the features it connects. This is done by storing the two anchor points of the distance we want the edge to represent (usually the minimum distance). This is done by adding four parameters: X1 and Y1 for the coordinates of the anchor on the geometry of the feature represented by node1, and X2, Y2 for the anchor on the feature associated with node2. Finally, an attribute weight has been added to store the weight of each edge, which can be used to decide which edge should be removed during the identification of clusters. The weight can be the Euclidian distance between X1Y1 and X2Y2, but not necessarily.

![Figure 5: example of a proxyEdge](image)
Triangulation
The triangulation is a graph which has a few specific properties, one of which is that it creates a partition of the space, where every cell has the shape of a triangle. A class is added to store these triangles, and attributes are added to all classes to store the relationships with the triangles.

- **Triangulation.** The class *Triangulation* extends the class *Graph*. It has two additional attributes: *TriangleList* contains the list of triangles of the triangulation, and *FeatureNodeRef* is a dictionary (list of unique keys, each one associated with a value). The keys represent all the geographic features on which the triangulation is built. For each key (geographic feature), the secondary value contains the list of nodes of the triangulation that lie on the geometry of the feature. This is useful to query the triangulation from the features. If the triangulation does not need to be linked to features, it featureNodeRef can be set to null.

- **TriEdge.** The class *TriEdge* extends the class *Edge*. It contains three additional attributes. *TriRight* is a reference to the triangle lying on the right of the edge (oriented from node1 to node2). *TriLeft* is a reference to the triangle on the left. *Length* is the length of the edge. It could be computed on demand by a method on the class, but as it is used often, we have decided to store it.

- **TriNode.** The class *TriNode* extends the class *Node*, but in fact is similar, because no attribute are added. We just created this class for defining specific methods on it.

- **Triangle.** The Class *Triangle* has nine attributes. *Node1*, *Node2* and *Node3* reference the three nodes forming the summits of a triangle. *Edge1*, *Edge2* and *Edge3* reference the three edges that form the sides of a triangle. *AdjTri1*, *AdjTri2* and *AdjTri3* reference the three adjacent triangles. Note that the triangle should always share *Edgei* with *AdjTrii*. In addition, *Nodei* is always the node opposite to *Edgei* in the triangle. This is illustrated in Figure 7. These rules allow a much more efficient navigation through the triangulation.

Figure 6: Example of a ProxyGraph split into two after removing two edges (e2 and e6)
4. FUNCTIONALITIES FOR OUR GRAPH CLASSES

This section describes basic functionalities required to build and use a proximity graph and a triangulation. The first section proposes some constructors for these two types of graphs, while the second section proposes a set of utilities to modify and query these structures.

4.1 Constructors

We have developed two types of constructors for a triangulation. One builds a triangulation over a set of vertices, the other constructs a triangulation over a set of geographic features. For the proximity graph, we have only one constructor which builds a proximity graph from a triangulation on features. The proximity graph is in fact deduced from the triangulation.

- **Building the triangulation from a set of vertices.**

  The method to build the triangulation over a set of vertices has reused the principles described in (Dwyer 1987) and (Shewchuk 1996). Experiments have shown that the method is one of the most efficient available (Su and Drysdale 1995). The method uses a divide-and-conquer approach which in effect creates a spatial index for the set of vertices, and makes the construction efficient, particularly when the number of vertices is large.

  The sequence of actions is as follows:
  
  - Sort the set of vertices by increasing x-values, and divide it by the middle into two sets. Each set is then sorted by increasing y-values and divided again. Alternative divisions using x and y values are carried out until all sets contain two or three vertices. These divisions can be seen in red lines on Figure 8a.
  
  - A mini triangulation is created for each set, either a single edge or a single triangle. Each edge which is not linked to a triangle on both sides has a “ghost” triangle added to it. A ghost triangle has a floating (not set) summit. Ghost triangles are very useful to efficiently traverse the convex hull of the triangulation. This can be seen in Figure 8a, where the edges of the triangulation are shown in plain black lines, while the ghost triangles are shown in black dashed lines.
  
  - An iterative bottom-up process is used to saw the triangulations two by two, following the division process in the reverse order (last sets divided have their triangulations sawed first). The sawing is done by identifying the two edges that need to be added to form a convex hull around the two triangulations. Then the two triangulations are “sawn” by freezing the ghost triangles lying between these junctions. Freezing is performed by fixing the floating node of the ghost triangle to a node on the convex hull of the other triangulation. This can be seen in Figure 8b and Figure 8c. Pink dashed lines show the convex junction that determine the sawing areas and green arrows show on which nodes the floating summits of ghost triangles move. Note that after each sawing, a ghost triangle needs to be added on each convex junction edge. In some cases (not in the figure), triangles adjacent to the convex hull and inside the sawing area must be removed, as their circumcircle contains a node from the opposite triangulation. In such cases, the edge of the triangle which lies on the convex hull is removed and the two remaining edges are associated with a new ghost triangle. Figure 8d shows the result.
Constructing a triangulation from a set of features.
Constructing a triangulation on a set of features consists of retrieving all the vertices from all the features and using the constructor for a set of vertices as described above. In addition, FeatureNodeRef dictionary of the triangulation is populated, so that each feature is associated with the list of nodes associated with it in the triangulation. This constructor offers an option to build a constrained triangulation. It this is required, the triangulation is first built normally on vertices, then all the edges of the features geometries are forced in the triangulation. This is done by flipping edges of the triangulation when required. The FeatureNodeRef allows the parts of the triangulation which need updating to be accessed very easily.

Constructing a proximity graph on a set of features.
To build the proximity graph, the constructor first builds a constrained Delaunay triangulation on the set of features. Then the graph is populated by adding a node for each feature, and an edge between any pair of nodes whose associated features are connected by at least one edge in the triangulation. When several edges of the triangulation connect the two features, we retrieve the triangles that connect these features and use them to find the minimum distance between the features. This is used to populate the attributes X1, Y1, X2, Y2 and length for the edge of the proximity graph.

4.2 Methods
In this section, we present a list of methods which we have implemented in our prototype for adding some flexibility to the creation of the different structures, and also to query them (navigate through them). We only have listed here the most interesting ones. A lot more methods have been implemented for the construction of the different structures. These lists of methods can and should be extended to extract more information from these spatial structures.

The classes Graph, Edge and Node do not currently have interesting methods. They have been implemented in the subclasses, but some are similar in the different subclasses and should be moved to this level, and then inherited.

Methods for the class Triangulation
- InsertNode: The method receives a TriNode as parameter, and inserts it in the triangulation, using the convex hull insertion algorithm (Tsai 1993).
o **InsertFeature**: The method receives a feature and a boolean to specify if the insertion should be done in constrained mode or not. The insertion is then done by successive call of InsertNode and forceEdge (see below) if required (for constrained insertion).

- **Methods for the class TriNode**
  - **GetEdgeToOtherNode**: The method takes a node of the triangulation as parameter. The method returns the edge that connects the current node to the one passed in parameter, or null if there is no such edge in the triangulation.
  - **GetTriangles**: This method returns the list of triangles which have this node as one of their summits.

- **Methods for the class TriEdge**
  - **GetNextNode**: The method takes a node of the triangulation as parameter. The node must be connected to the current edge, and the method returns the other node of the edge.
  - **GetOtherTriangle**: This is given one triangle adjacent to the current edge, and returns the other one.

- **Methods for the class Triangle**
  - **GetCircumCentre**: This returns the centre of the circum circle of the triangle.
  - **GetOppositeEdge**: Given a node of the triangle, this returns the edge of the triangle which does not connect to this node.
  - **GetOppositeNode**: Given an edge of the triangle, this returns the node of the triangle which is not connected to this edge.

- **Methods for the class ProxyGraph**
  - **GetEdgeToOtherNode**: The method takes a node of the graph as parameter. The method returns the edge that connects the current node to the one passed as a parameter, or null if there is no such edge in the graph.
  - **InsertEdge**: The method receives two features and adds an edge between them in the graph.
  - **RemoveEdge**: The method removes an edge passed as a parameter from the graph.
  - **GetLowestEdge**: Returns the edge of the graph with the lowest weight.
  - **ReduceToMST**: Returns the minimum spanning tree for this graph.
  - **SplitBasedOnWeight**: Given a threshold weight, the graph is split by eliminating the edges which have a weight higher than the threshold. This creates a hierarchy of graphs, with as many subgraphs as there are isolated groups of nodes created by the edge elimination. This is used for identifying clusters of features.
  - **GetNodeFromFeature**: Given a feature, this returns the node that represents it in the graph.

- **Methods for the class ProxyEdge**
  - **GetNextNode**: The method takes a node of the graph as parameter. The node must be connected to the current edge, and the method returns the other node of the edge.

- **Methods for the class ProxyNode**
  - **GetAllNeighbours**: Returns the list of nodes that can be directly accessed from the current one (by traversing a single edge).
  - **GetEdgeToOtherNode**: The method takes a node of the graph as parameter. The method returns the edge that connects the current node to the one passed as a parameter, or null if there is no such edge in the proximity graph.

### 5. RESULTS

We have implemented a prototype in Java using the data model and methods described in the previous sections. Our prototypes work as a plug-in for the Clarity platform, developed by Laser-Scan. All the spatial structures classes are implemented in Java, and the geographic features are stored in Gothic classes (object database developed by Laser-Scan, on which Clarity has been built). Figure 9 illustrates some of the capabilities of the prototype developed. We have computed triangulations and graphs on a set of buildings, but we could do the same on a mix of objects of different geographic classes and geometric types (line, area, or point). Figure 9a shows a non-constrained triangulation. You can notice a few edges crossing some building boundaries. This does not occur in Figure 9b which shows the result of constructing a constrained triangulation on the same set of features. Figure 9c shows the proximity graph which has been generated from the triangulation (both generate the same result in this case). Figure 9b shows some clusters obtained by splitting the proximity graph using a threshold value of 10 metres (edges with a weight higher than 10 have
been removed, which means that pairs of features which minimum distance is more than 10 metres are not connected in the graph. Figure 9e shows a minimum spanning tree deduced from the proximity graph.

![Figure 9: Results obtained: a) triangulation, b) constrained triangulation, c) proximity graph, d) clustering, e) Minimum Spanning Tree](image)

The tests have been done on a Laptop with a processor running at 1600MHz, and with 1GB of RAM. The triangulation took 70ms to build, while the constrained triangulation took 100ms. Both triangulations have been built on 12 features, contain 85 nodes and 236 edges. Table 1 shows the execution times required to build simple triangulation, constrained triangulations and proximity graph, over larger sets of features.

<table>
<thead>
<tr>
<th>Features/Nodes/Edges</th>
<th>Triangulation (ms)</th>
<th>Constrained Triangulation (ms)</th>
<th>Proximity Graph (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>12/85/236</td>
<td>70</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>84 / 425 / 1251</td>
<td>360</td>
<td>400</td>
<td>440</td>
</tr>
<tr>
<td>120 / 755 / 2248</td>
<td>920</td>
<td>1060</td>
<td>730</td>
</tr>
<tr>
<td>201 / 1068 / 3181</td>
<td>1350</td>
<td>1500</td>
<td>1500</td>
</tr>
<tr>
<td>225 / 1294 / 3859</td>
<td>1500</td>
<td>1740</td>
<td>1700</td>
</tr>
<tr>
<td>262 / 1707 / 5086</td>
<td>1880</td>
<td>2200</td>
<td>2200</td>
</tr>
<tr>
<td>462 / 2304 / 6886</td>
<td>3200</td>
<td>4000</td>
<td>5000</td>
</tr>
</tbody>
</table>

Table 1: Computation times to build triangulations and deduce proximity graphs

8. CONCLUSIONS

This paper has highlighted the heavy use of spatial structures made by automatic generalisation algorithms. Such structures are rarely available in commercial GIS platforms, and do not provide the flexibility and richness of tools that developers of automatic solutions require. We have proposed a model to store two types of graphs and the methods to construct them and navigate through them. This model and the tools are not new, they have been extracted from different studies, mostly from the computational geometry domain. We hope that such models will soon be available in commercial GIS, which would allow developers to base their algorithms on similar spatial structures, saving development time, and allowing greater interoperability between the generalisation algorithms.

The prototype developed and described in the paper has proved robust and very useful. It has been used to identify clusters of buildings to control their amalgamation, to control the amalgamation of forest polygons, and also to collapse dual carriageways into single centre lines. All these applications are described in more detail in (Revell, Regnauld, and Thom 2005).

REFERENCES

Gold, C. and Yang, W., 1996, Managing spatial objects with the VMO-Tree. Proceedings of The Seventh Symposium on Spatial Data Handling, 711-726.


This article has been prepared for information purposes only. It is not designed to constitute definitive advice on the topics covered and any reliance placed on the contents of this article is at the sole risk of the reader.

BIOGRAPHY

Nicolas Regnauld is in charge of the generalisation team at the research department of Ordnance Survey. His PhD supported by the Institut Géographique National (Paris) focused on the typification of buildings, i.e. the reduction of the number of buildings on a map while preserving their distribution pattern. He has since been working at the University of Edinburgh on the EU funded project AGENT, aiming at developing a generic platform for automated generalisation.

This article has been prepared for information purposes only. It is not designed to constitute definitive advice on the topics covered and any reliance placed on the contents of this article is at the sole risk of the reader.