# Topology Library Architecture

Kittel Austvoll

December 3, 2008

## Contents

## 1 Introduction

The topology module code can be divided into two sections:

- Code used to create layers: *CreateLayer*
- Code used to connect layers: *ConnectLayer*

## 2 Creating layers

A layer is a special kind of subnet model. There are two kinds of layers; fixed grid layers (*layer.h/cpp*) and unrestricted layers (*layer_unrestricted.h/cpp*). A new 3D layer might also be added in the close future (*layer_3d.h/cpp*).

A layer is simply a list of node pointers (inherited from the compound class), and a dictionary describing the layout of the layer.

## 2.1 Translating a spatial position to an id in a node vector

The primary purpose of the layer is to serve as a bridge between a 2D (or in the future 3D) position and the node vector in the Compound class. The mapping between a spatial position and an id in the node vector have to be done at run time whenever needed.

The fixed grid layer solves this problem by wrapping the node vector into a 2D matrix. See *Austvoll*[1] for more about fixed grid layers.

The unrestricted layer uses a quadtree structure to map a position to an id in the node vector. An unrestricted 3D layer would use an oct-tree structure to do the same job.

### 2.1.1 The quadtree structure

A quadtree is a data structure used to simplify the process of doing spatial searchs in a list of spatially distributed geographical data. The structure is created by repeatedly splitting the 2D space where the data points reside into four subregions (quadrants). The splitting is done whenever a quadrant contains more than the maximum number of allowed nodes. A region in space having a high density of nodes would thus contain more quadrants than a region with less density. The number of allowed nodes is kept fixed.

The quadtree only needs to be constructed at the time of performing spatial connections with the layer.

The oct-tree is the 3D equivalent of the quadtree.

#### Inserting layer nodes in a quadtree structure

```
for each node in layer
        find quadrant leaf that overlaps node position
        if quadrant leaf is full
                split quadrant into 4 sub-quadrants
                        re-insert all nodes overlapping the split quadrant
        else
                insert node in quadrant node list
```

#### Making spatial queries to a quadtree structure

```
identify minimum bounding box of search query (1)
find quadrant leaves overlapping minimum bounding box (2)
for all nodes in returned quadrant leaves
        if node overlaps exact region of search query
                add node to result
```

(1) The minimum bounding box (mbb) is the smallest possible rectangular region covering the search region. For a rectangular search region the mbb will equal the search region. (2) More on this below.

**Finding quadrant leaves overlapping minium bounding box**

```
iterate through tree structure
        if upper left corner of mbb have been reached
                if lower right corner of mbb haven't been reached (1)
                       add quadrant to result
```

(1) Also include quadrant overlapping lower right corner. (Comment) The procedure above can in some cases include many quadrants that don't overlap the query region. It is then important to remember that the purpose of the quadtree is not to find the exact data points overlapping the query region but to narrow the number of data points down to a small number that can quickly be tested individually. The algorithm above is described in *Rigaux et al.*[2].

## 3 Connecting layers

The layer connection process can be divided into two parts:

- Initialize the layers and the other connection variables
- Connect the layers

**Creating a topological connection**

```
initialize layers
initialize connection variables

for all nodes in driver layer (1)
        find nodes in pool layer that overlaps scope region (2)
        modify scope selection according to random settings
        for all scope nodes
                determine weight
                determine delay
                connect driver node and scope node
```

(1) The driver layer is the target layer for a receptive field connection and the source layer for a projective field connection. (2) The pool layer is the opposite of the driver layer.

### 3.1 Initializing the layers

A layer may consists of nodes at many different subnet depths and of many different model types. Slicing of the layer based upon these two parameters is done in this phase of the connection process. The desired nodes from the original layer are inserted into a new temporary layer container. The temporary layer container is used in the rest of the connection process. Information about node depth is lost in this process.

**Slicing a connection layer**

```
for all nodes in layer
        if node has desired modeltype and depth
                add node to temporary node list

create copy of original layer with new node list
```

## 3.2 Initializing the other connection variables

The nature of a topological connection depends upon several parameters. Most of these are implemented in the three file pairs *parameters.h/cpp*, *region.h/cpp* and *topologyconnector.h/cpp*.

The *region.h/cpp* classes are used to set up the scope used to retrieve nodes from the pool layer.

The *topologyconnector.h/cpp* classes are used to refine the scope node selection (according to the random conditions), set up the weight and delay of an individual connection and to create the final connection.

The *parameters.h/cpp* classes are used by the *topologyconnector.h/cpp* classes to set up the weight, delay and probability variables.

## 3.3 The scope

A topological connection is created by iterating through one node layer and connecting every node in this layer to a scope or region in another layer.

The scope is defined by the region classes (*region.h/cpp*). There are two kinds of region classes, one for use with fixed grid layers, and one for use with unrestricted layers. We won't discuss the fixed grid layer scope any further here (See *Austvoll*[1] for more information about this region class).

The unrestricted layers currently allow for rectangular, circular and doughnut regions. But additional region types can easily be added. The region objects are both identified by their minimum bounding box and by the exact 2D shape of the objects.

**Find nodes overlapping an unrestricted region object**

```
query quadtree for nodes overlapping scope
        quadtree->find nodes overlapping minimum bounding box
        quadtree->check if nodes also overlap exact region
        quadtree->return nodes
return nodes
```

## 3.4 Extracting nodes based upon probability

Note: This procedure is under discussion and development.

Once an iterating node and a group of scope nodes have been identified the nodes can be furthered processed based upon the parameters related to randomness. There are currently two ways to use random numbers to make refinements to the scope node selection.

- Limiting the set of scope nodes based upon a random divergent/convergent connect scheme.
- Drawing a random number for each scope node to determine if the node should be connected to or not.

The two procedures can also be combined.

**Refining the scope region based upon randomness**

```
if random divergent/convergent connect
        while node limit is not reached
                randomly draw a node from scope
                if probability of connection is used
                        determine probability
                        if random number < probability
connect current scope node
else
                                throw node back in scope pool
else
        for all nodes in scope
                if probability of connection is used
                        determine probability
                        if random number < probability
                                connect current scope node
```

## 3.5 Determining the value of the weight, delay and probability parameters

The parameters classes are used to determine both the probability, the weight and the delay of each individual scope node. The parameters classes usually depend upon the relative position of the iterating and the scope node. Currently three parameters classes exist; a class producing a constant value regardless of node positions, a class producing values based upon a gaussian function of the relative position of the connecting nodes, and a class used by the fixed grid layer to produce a matrix of parameter values.

**Retrieving the weight, delay or probability of a scope node**

```
calculate relative position of iterating and scope node
parameters->calculate parameters value based upon relative position
return parameters value
```

### 3.6 Connect the nodes

The last step of the connection process is to call the *Network::connect(..)* function. This is done by the *TopologyConnector* classes. A connection can either be a receptive field connection or a projective field connection. A receptive field connection will use the iterating node as a target and the scope node as source. The projective field connection will use the iterating node as source and the scope node as target.

#### Calling the right version of the *Network::connect(..)* function

```
if receptive field
        connect scope node to iterating node with
        determined weight and delay
else if projective field
        connect iterating node to scope node with
        determined weight and delay
```

## References

[1] K. Austvoll: *NEST Topology module*, Norwegian University of Life Sciences, Master Thesis (2007)

[2] P. Rigaux, M.O. Scholl, A. Voisard: *Spatial Databases: With Application to GIS*, Morgan Kaufmann