Topology extension for OpenJUMP

v 0.8 (2014-05-26) by Michaël Michaud

1.1 History

0.8.0 (2014-05-26) : complete **rewrite of ProjectPointsOnLines plugin** which was buggy

0.7.0 (2013-01-27) : **add CoverageCleanerPlugIn**

0.6.1 (2013-01-09) : add es/it/fi language files (from G. Aruta and J. Rahkonen)

0.6.0 (2012-09-15) : complete rewrite of ProjectPointsOnLines to make multi-projections possible

0.5.1 (2012-08-30) : fix a problem in UI

0.5.0 (2012-06-25) : **add ProjectPointsOnLinesPlugIn**

0.4.0 (2012-05-20) : NetworkTopologyCleaning : fix a bug in node degree computation and add an option to check attribute equality before snapping

0.3.4 (2012-05-17) : NetworkTopologyCleaning : nodes inserted in wrong segment (ref layer)

      deactivate CoverageCleaner (not robust enough for 1.5.2 release)

0.3.3 (2011-11-22) : **add CoverageCleaner**

0.3.2 (2011-11-08) : fix a typo in fr language file

0.3 (2011-04-22) : creates a single topology extension from available QA and Topology plugins

0.2 (2011-04-11) : upgrade to new MultiInputDialog coming in OpenJUMP version 1.4.1

0.1 (2010-04-22) : initial version

1.2 General concepts

Topology Extension is a set of plugins bundled as a jar file to be placed in OpenJUMP plugin directory (default plugin directory is lib/ext).

All plugins from Topology extension are related to topology error detection and correction.

Important : this extension works on geometries and heavily uses JTS capabilities. On the other hand, graph extension uses JGraphT capabilities and works on graphs built from feature relations.

This extension has no external dependency, but JTS and OpenJUMP.

1.2.1 Topology extension plugins and common options

Every plugin from topology extension is installed in the Plugins menu of OpenJUMP, in a submenu called... Topology.

Most options have a tooltip giving more details about option signification.

2   Remove micro-segments PlugIn

This plugin removes micro-segments from LineStrings or Polygons.

This process is often needed before a cleaning process involving topology.

Indeed, most cleaning process use a distance threshold to decide if two features (resp. nodes, segments) should be snapped together, and presence of micro-segments less than this threshold inside single geometries often disrupt the process.

What the plugin do and what it doesn't do :

- "Remove micro-segment" carefully chooses the point to remove from the geometry so that the deformation is minimal : micro-segments have two vertices. Vertices located on the geometry boundary are never removed, and if both vertices are strictly inside the linestring, the plugin removes the vertex where interior angle is the closest to flat angle (180°);

- "Remove micro-segment" processes only LineStrings and Polygons yet. GeometryCollections are just ignored.

- "Remove micro-segment" does not check if the vertex it will remove is used by another geometry. The process, which is purely geometric, may break topology consistency. Hopefully, other topologic correction tools should be able to repare what has been broken.

Special use-cases :

"Remove micro-segments" is able to fix invalid polygons with a very small self-intersection :
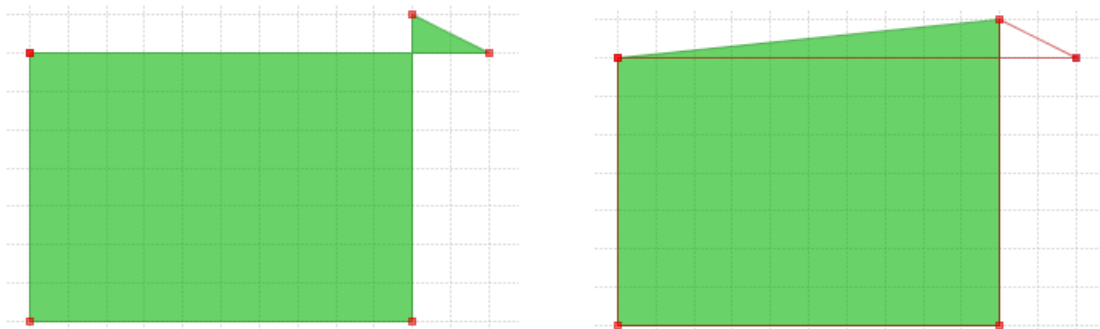

Note that the problem is fixed by the removal of a point, which only works if self-intersection is made of a single micro-segment.

Currently, OpenJUMP lacks a user-friendly-tool to fix larger self-intersection problems. A quite complex process could be to create linestrings from polygon boundaries, node those linestrings, polygonize the result, remove polygons representing holes, get attributes back using matching tools...
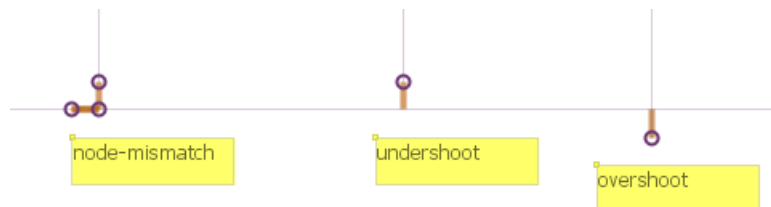
3 Network topology cleaning PlugIn

This plugin can process a dataset representing a network, and detect and/or fix some common topologic errors like

- node mismatches : nodes closed to each other but not exactly the same

- undershoot : a linestring end is close to another linestring but is not snapped on it
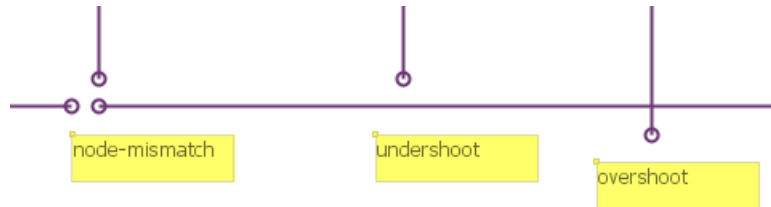
- overshoot : a linestring crosses another one and end just after the intersection

Formal definition : network topology cleaning plugin finds every node of a network (end-points of the network linestrings) which are closed enough to another LineString (closed-enough being user-defined), but not snapped on this LineString (no common vertex).
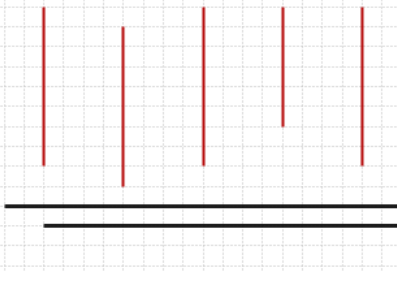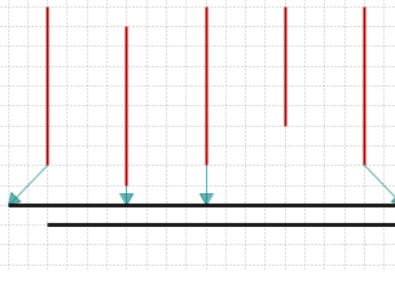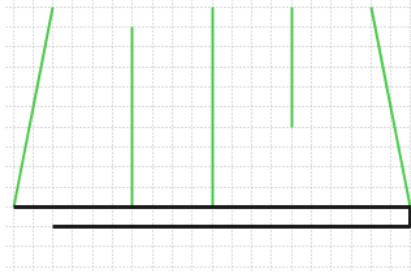
Anomalies can be either detected or corrected.

3.1 The network topology cleaning plugin main parameters
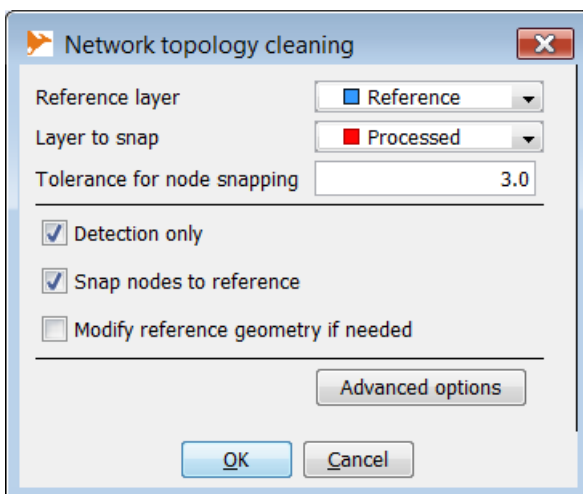
The main parameter is the distance threshold between a node of the layer to process and a feature of the reference layer :
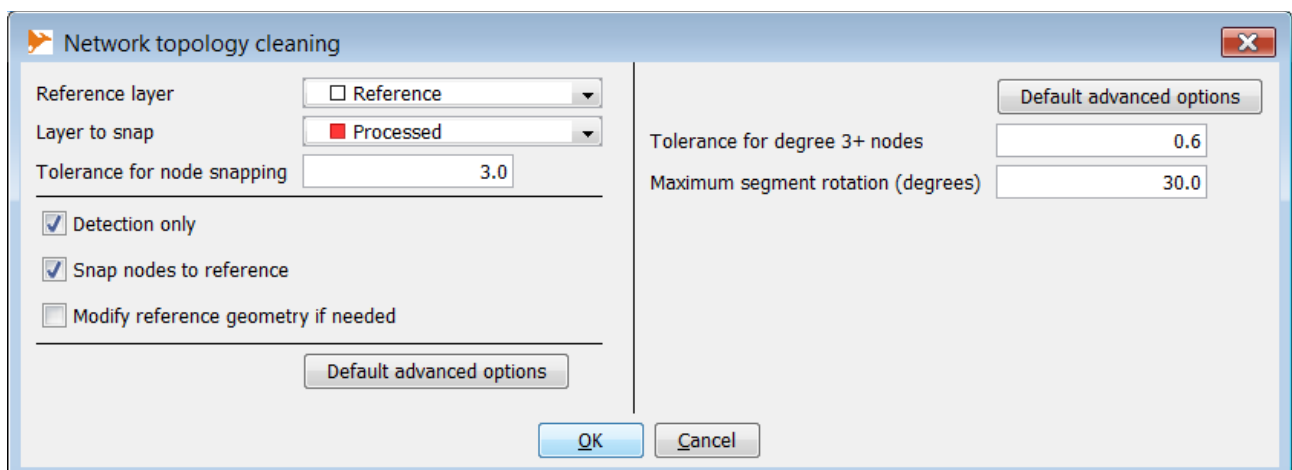
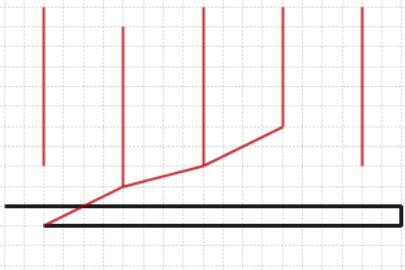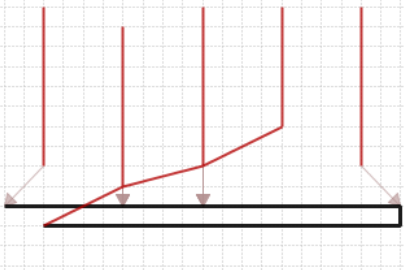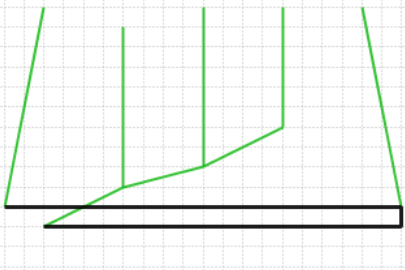|		
Tolerance is 3 meters. The 1st, 2nd, 3rd and 5th feature are under the tolerancere.	Detection : 1st feature is at less than 3 m from a reference node 2nd and 3rd features are at 1m from a reference segment, at 2 meters from another one and at more than 3m from any vertex 4th feature is at more than 3 meter from the reference feature 5th feature is at less than 3 m from a reference vertex	Correction : 1st and last features are properly snapped to reference 2nd and 3rd features are snapped, but a vertex needed to be inserted into the reference layer.

3.2 Advanced options : degree 3 nodes

An option is available to deal with degree 3+ nodes of the processed layer. It maybe useful to decrease the tolerance parameter for these nodes, because if they are not snapped on the reference layer, there is a higher probability that it should not be snapped.

Default value of degree 3 nodes snap tolerance is 1/5 base tolerance.

Changing the tolerance for degree 3+ nodes or setting it back to default is quite easy :
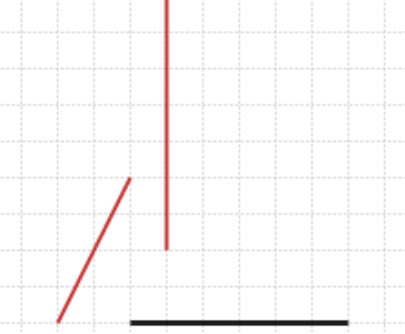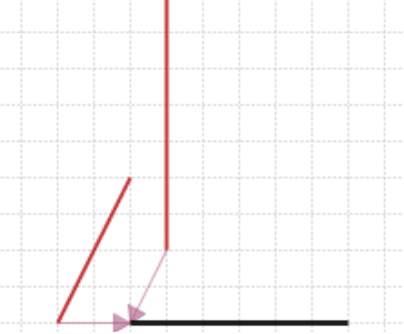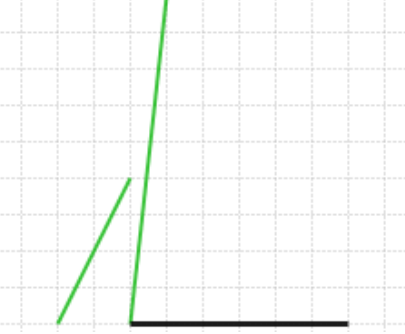
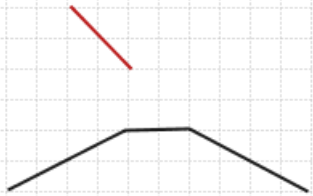|		
Degree of the lower node of vertical features 2, 3 and 4 = 3 (this means that feature extremities not only snap another feature, they share their node with another feature node)	Mismatch indicators are created for every node at less than 3 m from the reference feature but...	only the degree 1 nodes are snapped. Degree 3 nodes would have been snapped if they were under the default tolerance (here, 3/5 = 0,6 m)
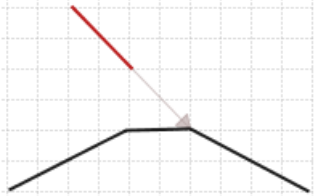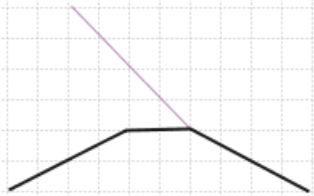
3.3 Advanced options : rotation parameter

There is a second editable parameter which is a rotation tolerance.

Rotation tolerance is available for pendant nodes only (degree 1). Indeed, using a rotation tolerance on degree 2+ nodes would have bad side effects as it could break many connected features just because of their segment orientation.
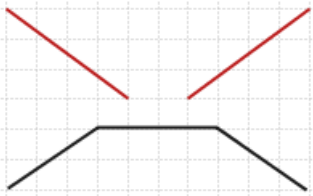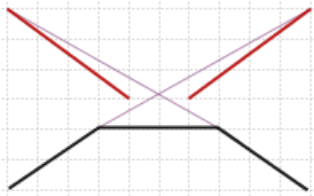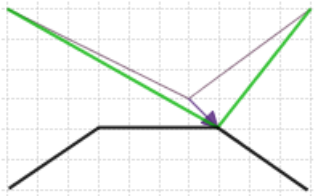
Here, we have two candidate features at the same distance from the reference layer : 2 m	Both could be snapped to the reference, but for the first one, the snap will rotate the segment much more than for the second one	Finally, with the rotation tolerance set to 15°, the first feature is not snapped while the second one is.

When several vertices are found within the distance tolerance rotation criteria is used to determine which vertex to snap to
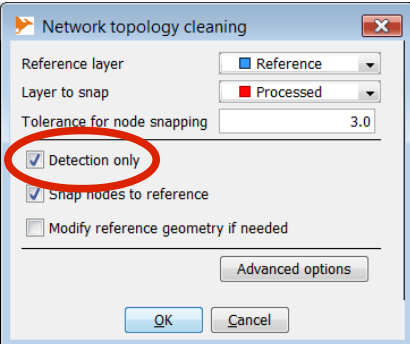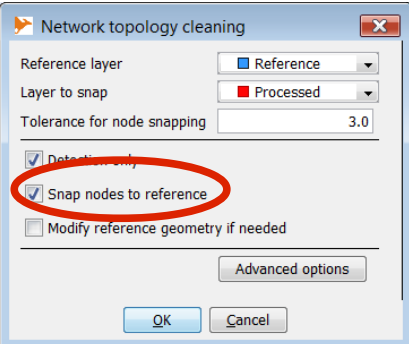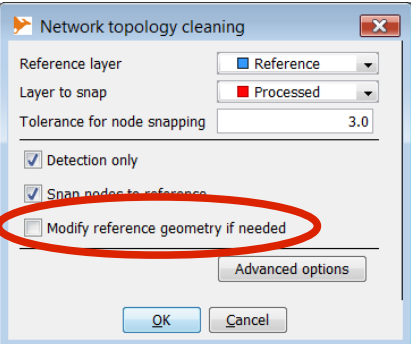
The red LineString is within the 3 m tolerance and both vertices of the nearest reference segment are in the 3 m tolerance.	The choosen snap vertex is not the nearest one...	but the one introducing the smallest segment rotation.

3.4 Corner cases

Corner cases and situations where the plugin may give weird results

Because of the angle criteria preference, in this case where both vertices are within the distance tolerance (4 m) and where segment rotations are minimized...	the process resuls in two crossing lines.	A work-around is to pre-process the input layer to eliminate the node-mismatch, then to process the result.

3.5 Output options

Detection : a layer will be created with translation vectors needed to snap nodes to reference features. See below this layer attributes	Correction : a copy of the processed layer is done and nodes of this copy located near but not snapped to the reference layer will be	Reference correction : this option is only available if the reference layer is editable. If checked, missing vertices will be inserted in the reference

	snapped.	layer just where the processed feature has been snapped to.

3.6 Mismatch vector layer

SNAP attribute shows if the node has finally is been snapped on a reference node, on a reference vertex, on a reference segment, or not snapped at all :

Snap to node : the node has been snapped on a reference node

Snap to vertex : the node has been snapped to a reference vertex

Snap to segment : node has been snapped in the middle of a reference segment

Not snapped : the node has not been snapped

Not snapped (D > xx) : the node has not been snapped because it is over the distance tolerance (the degree 3+ one, otherwise, there is no reason to have a mismatch vector)

Not snapped (A > xx) : the node has not been snapped because the result would have been over the rotation angle tolerance

For snapped nodes, **attribute "Rotation"** gives the angle the node segment has been rotated to snap the reference feature (in degrees).

4 Project points on lines PlugIn

This plugin project points of a point layer onto lines of a line layer. Optionnaly, projection points can be inserted into lines or used to split the lines.

4.1 Inputs

Point layer must contain at least one point.

Line layer must contain at least one LineString. MultiLineStrings will not be used. If you have MultiLineStrings in your linear network, decompose them into simple LineStrings before starting the process.

4.2 The PlugIn parameters
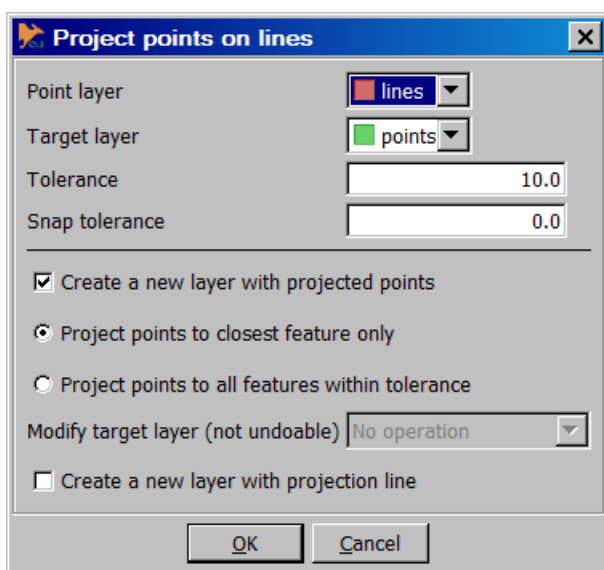
Tolerance : This is the main parameter. If the distance to the closest feature is greater than tolerance, the point will not be projected. By default, the point is projected to the closest location (which generally means at right angles to the closest segment). See above for snap option.

Snap tolerance : if not null, snap tolerance makes it possible to project a point to an existing vertex if the distance from this vertex to the othogonal projection of the point is less than snap tolerance. To snap a vertex, the total distance between the vertex and the source point must also be under main "Tolerance" parameter. Note that if the source point is already on the line, but near a vertex point, this option may move the point along the segment to the closest vertex.
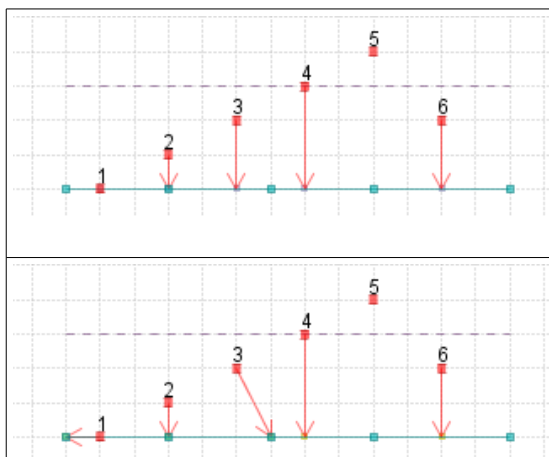
	With a **Tolerance of 3** and a **Snap tolerance of 0**, points are projected at right angle to the closest line, except if their distance to the line is greater than the tolerance.
	Now let see the same operation with a **Tolerance of 3** and a **Snap tolerance of 1** : 1 : project to the closest vertice (which distance is less than the snap tolerance) 2 : project to the closest vertice which happens to be also the closest location on the line. 3 : project to the closest vertice (< tolerance) 4 : project orthogonally (total distance to the closest vertice is > tolerance) 5 : not projected (> tolerance) 6 : project orthogonally (vertices > snap tolerance)

Create a new Layer with Projected points : creates a new layer containing the projection of points on the nearest feature. The layer is called "sourceLayerName - projected". It has the same schema as the source point layer plus one attribute containing distances between source point and closest location. Note that the distance reported in the attribute is the orthogonal distance, even if the final projection is not exactly orthogonal.

Project points to all features within tolerance : can project a single source point several times if several target features are within the tolerance distance.

Modify line layer (not undoable) : to be more safe, this component is only available if line layer is editable. It has three options :

- No operation : do not modify line layer

- Insert vertices in lines : add the projected points to the line geometry

- Split lines (create MultiLineStrings) : split line where points have been projected

Inserting points and splitting lines is performed in the source layer itself and is undoable. This choice has been guided by performance and memory considerations. So, make a copy of your layer before processing it if you're not sure.

Create a new layer with projection lines : creates small line segments between source points and projected points. Moreover, for further analysis, these lines have the same attributes as the points, plus the distance between the point and its (orthogonal) projection (may be different from the segment length if snap tolerance is greater than 0).

5 Coverage cleaning PlugIn (topology-0.7.0)

This plugin cleans a coverage layer with small defects like overlaps and/or gaps.

It is based on Java Conflation Suite (mainly coded by Martin Davis, JTS architect), released by Vividsolutions. Improvements added from the original plugin are :

- performance improvements
- support of polygons with holes
- small bug fixes

5.1 How it works

How the plugin works ? The plugin tries to match each segment of each polygon with segments of neighbour polygons. Segments match when :

- they are not topologically equals (AB ≠ CD et AB ≠ DC)
- the minimum distance between segments is less than the user-defined tolerance,
- the angle between both segment is less than the user-defined tolerance
- the orthogonal projection of each segment on the matched one is non null

If matched segments are found on a contour, matching contours are merged by :

- snapping vertices of matched segments when they are close enough
- inserting vertices from ring A missing in ring B and the other way.
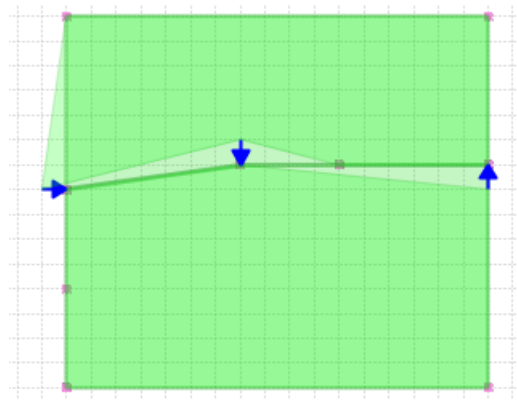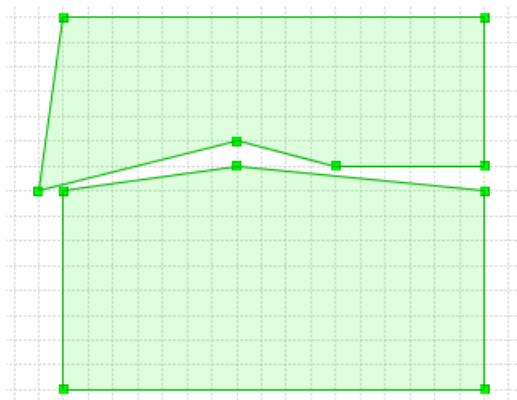
5.2 Options

The following options are available in the coverage cleaning plugin.

Explode multipolygons first : the plugin cannot clean a multipolygon layer. Tha't why exploding geometries is a default option. For a very large layer, it may be a better idea to make polygon simple out of the plugin in order to save memory used by the cleaning phase.

Normalize polygons first : the plugin cannot clean a coverage if polygon orientation is not normalized. Tha't why the option "normalize geometries first" is the default option. For a very large

layer, it may be a better idea to normalize the layer before running the cleaning plugin in order to save memory used by the cleaning phase. Note : buffer(0) produces normalized geometries.

Distance tolerance : if the distance between segments is more than this parameter, polygons are not adjusted.

Angle tolerance : if the angle between two segments (modulo PI) is more than this parameter, polygons are not adjusted.

Use fence : if this option is checked and a fence is available in the fence layer, only the parts of the polygons located inside the fence are adjusted.

5.3 Internal

The coverage cleaning plugin is quite complex. Here is a few explanation about how it works which can be useful for maintenance purpose.

5.3.1 Main data structures used by the plugin

Main data structures, from the largest to the smallest are :

Coverage : coverage maps each feature of the source dataset to a CoverageFeature (see above). It also incudes a VertexMap (see above) and an array of adjustable coordinates.

CoverageFeature : contains a reference to the parent Feature, a Shell representing the exterior ring, n shells representing holes and some indicators on the matching process (isProcessed and isAdjusted).

- *computeAdjustmentSingle* computes adjustment with shells of nearby features, and
- *getAdjustedGeometry* returns the final geometry for this Feature.

Shell : this is the most important datastructure of the algorithm. It contains a coordinate array with original coordinates, a coordinate array with adjusted coordinates and a array of segments where new coordinates will be inserted. Most interesting methods are :

- *match*(Shell, SegmentMatcher,SegmentIndex) returning a boolean and
- *computeAdjusted*() computing the final coordinates

FeatureSegment : a special extension of JTS *LineSegment* an equals method which makes a topologically equals for indexation purpose (see also *Segment*).

Segment : a GeometryComponent which modelizes a segment of the source geometry which vertices will be adjusted and where vertices from other shells will be inserted.

Vertex : an adjustable point of the coverage. It contains a original coordinate, an adjusted coordinate and a set of pointers to shells sharing this vertex.

VertexMap : a data structure mapping Coordinates to Vertices. Thanks to this data structure, all shells sharing a same coordinate in the source dataset will share the adjusted coordinate in the final dataset.

5.3.2 WorkFlow

The class leading the whole process is CoverageCleaner. The process is organized in the following way :

1 - Get matched segments and matched features

Find matching FeatureSegments (intersecting fence if fence is not null). It uses *InternalMatchedSegmentFinder* to

- retain unique segments (remove segments topologically equals to each other)
- create a spatial index with unique segments
- put matching unique segments in a list (matchedSegments)

Then, it finds Features containing matching segments.

2 - Load matched segment index (and coord set)

Creates a *SegmentIndex* (called *matchedSegmentIndex* in the code) containing each single matching FeatureSegment.

Creates a Set containing all their coordinates (called *matchedSegmentCoordSet* in the code)

3 - SetAdjustableCoordinates (put *matchedSegmentCoordSet* into Coverage)

Associate the set of unique coordinates belonging to a matching segment with the coverage object.

4 - Get all Features involved (from *matchedSegmentCoordSet*)

Get all features with a point included in *matchedSegmentCoordSet*. Note that features with no matching segment but with a coordinate in *matchedSegmentCoordSet* will have to be adjusted.

5 - Creates a new dataset which will be upadated to include new geometries

6 - Adjust all adjustable features

6.1 - Adjust features with nearby features as following.

The process is as follow :

- For each candidate Feature, find nearby features
  - For each nearby feature use *CoverageFeature#computeAdjustmentSingle* to
    - match the shell with the nearby feature shell
    - match each hole shell with the nearby feature

For each shell-shell comparison, compare all pairs of LineSegment.

Eliminate segment pairs if one of them (or both) do not belong to the set of matched segment,

- test if segments match with *SegmentMatcher#isMatch*
- if they match,perform a *Segment#addMatchedSegment*

6.2 - Then update features (Coverage#computeAdjustedFeatureUpdates)