Automatically repairing polygons and planar partitions with *prepair* and *pprepair*

Ken Arroyo Ohori  Hugo Ledoux  Martijn Meijers

OSGIS 2012
September 5, 2012
Planar partition = no gaps, no overlaps
Real data = problems
Real data = problems
What about polygons?

OGC Simple Features + ISO19107:

1. no self-intersection
2. closed boundaries
3. rings can touch but not overlap
4. no duplicate points
5. no dangling edges
6. connected interior
Real data = problems
Real data = problems
Real data = problems
Standards/definitions tell us what is valid, but... what to do with invalid data?

- Planar partitions: snapping / topology rules / manual work

- Polygons: “buffer-by-0” / PostGIS 2.0’s ST_MakeValid() / “visual repair”
Snapping

- Tolerance (threshold) is used for snapping vertices
- Tolerance based on scale of datasets
- Works fine for simple problems
Snapping

- Tolerance (\textit{threshold}) is used for \textit{snapping} vertices
- Tolerance based on scale of datasets
- Works fine for \textit{simple} problems
Snapping

- Tolerance (threshold) is used for snapping vertices
- Tolerance based on scale of datasets
- Works fine for simple problems
Snapping

Spikes and punctures can create invalid polygons
Snapping

Spikes and punctures can create invalid polygons
Snapping

Splitting of polygons into several polygons
Snapping

Splitting of polygons into several polygons
Snapping

High resolution data

Low resolution data

Topologically invalid result
Topology rules
“Buffer-by-0”
“Buffer-by-0”
A planar graph is constructed.
PostGIS 2.0’s ST_MakeValid()

- high-level automatic repair function
- diff functions called depending on geometric and topological configurations of rings
- based on construction of planar graph (GEOS is used)
- not documented (“read the code”) = predicting behaviour is difficult
- very slow for big polygons
“Visual repair”

GRASS

ArcGIS
Our solution = constrained triangulation (CT)

1. Construct CT of input polygons
2. Flag each triangle with its polygon/interior & exterior
3. Make each triangle have exactly one tag
4. Reconstruct polygons
Our solution = constrained triangulation (CT)

1. Construct CT of input polygons
2. Flag each triangle with its polygon / interior & exterior
3. Make each triangle have exactly one tag
4. Reconstruct polygons
Our solution = constrained triangulation (CT)

1. Construct CT of input polygons
2. Flag each triangle with its polygon / interior & exterior
3. Make each triangle have exactly one tag
4. Reconstruct polygons
Our solution = constrained triangulation (CT)

1. Construct CT of input polygons
2. Flag each triangle with its polygon / interior & exterior
3. Make each triangle have exactly one tag
4. Reconstruct polygons

<table>
<thead>
<tr>
<th>Triangle</th>
<th>Repair</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>&quot;red&quot;</td>
</tr>
<tr>
<td>B</td>
<td>&quot;red&quot;</td>
</tr>
<tr>
<td>C</td>
<td>&quot;red&quot;</td>
</tr>
<tr>
<td>D</td>
<td>&quot;red&quot;</td>
</tr>
</tbody>
</table>
Our solution = constrained triangulation (CT)

1. Construct CT of input polygons
2. Flag each triangle with its polygon / interior & exterior
3. Make each triangle have exactly one tag
4. Reconstruct polygons
Our solution = constrained triangulation (CT)

1. Construct CT of input polygons
2. Flag each triangle with its polygon / interior & exterior
3. Make each triangle have exactly one tag
4. Reconstruct polygons
Local control: 6 different operators

<table>
<thead>
<tr>
<th>Repair operation</th>
<th>Type</th>
<th>Criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td>Triangle by number of neighbours</td>
<td>Focal</td>
<td>The label present in the largest number of adjacent faces, overlaps included.</td>
</tr>
<tr>
<td>Triangle by absolute majority</td>
<td>Focal</td>
<td>Label present in two or more valid adjacent faces</td>
</tr>
<tr>
<td>Triangle by longest boundary</td>
<td>Focal</td>
<td>Label present along the longest portion of the boundary of the adjacent faces</td>
</tr>
<tr>
<td>Regions by longest boundary</td>
<td>Focal of zonal</td>
<td>Label present along the longest portion of the boundary of the adjacent faces</td>
</tr>
<tr>
<td>Regions by random neighbour</td>
<td>Focal of zonal</td>
<td>Random label from the adjacent faces</td>
</tr>
<tr>
<td>Triangle by priority list</td>
<td>Varies</td>
<td>Label that has the highest priority according to a predefined priority list</td>
</tr>
</tbody>
</table>
Local control: one concrete example
Local control: one concrete example
Local control: one concrete example
Local control: one concrete example
Local control: one concrete example
Experiments with big polygons: CORINE2006

<table>
<thead>
<tr>
<th></th>
<th>points</th>
<th>rings</th>
<th>prepair</th>
<th>ST_MakeValid()</th>
</tr>
</thead>
<tbody>
<tr>
<td>EU-47552</td>
<td>2,412</td>
<td>10</td>
<td>0.5s</td>
<td>0.8s</td>
</tr>
<tr>
<td>EU-47997</td>
<td>32,473</td>
<td>346</td>
<td>11.4s</td>
<td>314.0s</td>
</tr>
<tr>
<td>EU-180927</td>
<td>102,272</td>
<td>299</td>
<td>52.2s</td>
<td>740.2s</td>
</tr>
</tbody>
</table>
Experiments with large real-world datasets

(a) E41N27

(b) 4tiles

(c) 16tiles

(d) Mexico
## Experiments with large real-world datasets

<table>
<thead>
<tr>
<th></th>
<th># polygons</th>
<th># pts</th>
<th># pts largest polygon</th>
<th>avg # pts per polygon</th>
</tr>
</thead>
<tbody>
<tr>
<td>E41N27</td>
<td>14 969</td>
<td>496 303</td>
<td>26 740</td>
<td>34</td>
</tr>
<tr>
<td>4tiles</td>
<td>4 984</td>
<td>365 702</td>
<td>16 961</td>
<td>75</td>
</tr>
<tr>
<td>16tiles</td>
<td>63 868</td>
<td>6 622 133</td>
<td>95 112</td>
<td>104</td>
</tr>
<tr>
<td>Mexico</td>
<td>26 866</td>
<td>4 181 354</td>
<td>117 736</td>
<td>156</td>
</tr>
</tbody>
</table>
Comparison with other GIS packages

<table>
<thead>
<tr>
<th></th>
<th>pprepair</th>
<th>ArcGIS</th>
<th>FME</th>
<th>GRASS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>memory</td>
<td>time</td>
<td>memory</td>
<td>time</td>
</tr>
<tr>
<td>E41N27</td>
<td>145 MB</td>
<td>19s</td>
<td>145 MB</td>
<td>1m3s</td>
</tr>
<tr>
<td>4tiles</td>
<td>116 MB</td>
<td>17s</td>
<td>113 MB</td>
<td>37s</td>
</tr>
<tr>
<td>16tiles</td>
<td>1.45 GB</td>
<td>4m47s</td>
<td>crashes</td>
<td>–</td>
</tr>
<tr>
<td>Mexico</td>
<td>1.01 GB</td>
<td>3m31s</td>
<td>216 MB</td>
<td>&gt;1d</td>
</tr>
</tbody>
</table>

|          | memory   | time    |
| Mexico   | 216 MB   | >1d     |

> crashes
The code is robust and freely available

- [http://tudelft-gist.github.com/pprepair](http://tudelft-gist.github.com/pprepair)
- Uses OGR and CGAL
- BSD license → soon GPLv3

**prepair**

Automatic repair of single polygons

View the Project on GitHub
[tudelft-gist/prepair](http://tudelft-gist.github.com/prepair)

Download ZIP File  Download TAR Ball  Fork On GitHub

What is prepair?

prepair permits you to easily repair "broken" GIS polygons, and that according to the international standards ISO 19107. In brief, given a polygon stored in WKT, it automatically repairs it and gives you back a valid WKT. Automated repair methods can be considered as interpreting ambiguous or ill-defined polygons and giving a coherent and clearly defined output.

It performs more or less the same as the new PostGIS 2.0's function `ST_MakeValid()`, but is several order of magnitude faster, scales better to massive polygons, and predicting its behaviour is simple (so one can guess how her polygons will be repaired).

prepair is based on a constrained triangulation, and CGAL and OGR are used.
Thanks for your attention

Ken Arroyo Ohori
g.a.k.arroyoohori@tudelft.nl

Hugo Ledoux
h.ledoux@tudelft.nl

Martijn Meijers
b.m.meijers@tudelft.nl