Validation and Automatic Repair of Planar Partitions Using a Constrained Triangulation

KEN ARROYO OHORI, HUGO LEDOUX & MARTIJN MEIJERS, Delft, The Netherlands

Keywords: triangulation, repair, planar partition, topological data structure, polygonal coverage

Summary: Planar partitions are frequently used to model, among others, land cover, cadastral parcels and administrative boundaries. In practice, they are often stored as a set of individual polygons to which attributes are attached, e.g. with the Simple Features paradigm, causing different errors and inconsistencies, e.g. gaps, overlaps and disconnected polygons, which are introduced during their creation, manipulation and exchange. These errors severely hamper the use of planar partitions in other software, e.g. due to false assumptions causing erroneous calculations. Existing approaches to validate planar partitions involve first building a planar graph of the polygons and enforcing constraints, then repair is done by snapping vertices and edges of this graph. We argue that these approaches have many shortcomings in terms of complexity, numerical robustness and difficulty of implementation, and do not guarantee valid results. Furthermore, they are semi-automatic, requiring manual user intervention. We propose in this paper a novel method to validate and automatically repair planar partitions. It uses a constrained triangulation of the polygons as a base – which by definition is a planar graph – and only simple operations are needed, i.e., labelling of triangles, to both validate and repair. Perhaps the biggest advantage of our method is that we can guarantee that a planar partition is valid after repair. In the paper we describe the details of our method, our implementation, and the experiments we have done with real-world datasets. We show that our implementation scales to big datasets and that it offers better capabilities and overall performance than existing solutions.
1 Introduction

Planar partitions are frequently used in GIS to model concepts such as land cover, the cadastre, or the administrative boundaries of a given country. As shown in Fig. 1, a planar partition is a subdivision of a polygonal subset of the plane into non-overlapping polygons. In practice, planar partitions are often represented, and stored in a computer, as a set of individual polygons to which one or more attributes are attached, and the topological relationships between polygons are not explicitly stored, i.e., shared boundaries are thus represented and stored twice. The preferred method of practitioners is representing polygons according to the Simple Features specifications (OGC 2006), for instance as an Esri shapefile (ESRI 1998) or in a database, such as PostGIS (POSTGIS 2012).

If a planar partition is stored as a set of individual polygons, then in practice errors, mistakes and inconsistencies will often be introduced when the planar partition is built, updated or exchanged. Examples of common errors are: overlapping polygons, gaps between polygons, and polygons not connected to the others. This can be, among others, due to human error, the use of floating-point arithmetic, or limited precision (Schirra 1997). These errors can have catastrophic consequences for practitioners since most software and algorithms using planar partitions as input assume that this input is valid. At best erroneous results are returned, at worst it causes a software failure, often without any warning to the user. Moreover, such problems are often not visible at the scale that the data is usually viewed, exacerbating the problem (Laurini & Millere-
raffort 1994).

Fig. 1: Part of the Corine Land Cover dataset for the region around Delft, The Netherlands.
pare our method and its implementation to alternatives, both for validation and for repair. Finally, we discuss the advantages and disadvantages of using our method and the conclusions drawn from this in section 5.

2  Related Work

Since a planar partition is formed by a set of individual polygons, we first discuss what a valid polygon is in our context, and then we review existing methods to validate and repair planar partitions.

2.1  Simple Features and Validity of Simple Polygons

While there are several definitions of what constitutes a valid polygon (van Oosterom et al. 2004), we use in the following the standard Simple Features (OGC 2006), with the addition of the ISO 19107 Spatial schema (ISO 2003) polygon orientation rules. Simple Features defines a polygon as follows: “A Polygon is a planar Surface defined by 1 exterior boundary and 0 or more interior boundaries. Each interior boundary defines a hole in the Polygon.”. In the specification, six assertions are given that together define a valid polygon. Essential for a valid polygon is that the boundaries of the polygon must define one connected area. Additionally, a polygon can contain holes. We say that the exterior boundary of the polygon is the outer ring, and a hole is an inner ring. These holes can be filled by one or more polygons, which can recursively contain holes, which are filled by other polygons. Observe also that holes are allowed to interact with each other and the outer boundary under certain conditions, e.g. they are allowed to touch at one point, as long as the interior of the polygon stays one connected area. Each polygon is stored independently from other polygons, and it is not possible to store topological relationships between the polygons. The ISO 19107 standard (ISO 2003) is more ambiguously defined, but it does establish orientation rules (counter clockwise for the outer ring, clockwise for the inner ones), which we use in our output.

The validation of a single polygon is possible with different libraries, GEOS (GEOS 2012) and JTS (Vidid Solutions 2012) being two widely used open-source examples.

The repair of single polygons is a less documented topic than their validation. Different software vendors offer tools to help identify and semi-automatically repair broken polygons. Examples are ST_MakeValid() from PostGIS and the constraints in 1Spatial Radius Topology. The method we present in this paper has been adapted to automatically repair common errors in individual polygons, e.g. wrong ring orientation, or holes that split the interior of a polygon (Ledoux et al. 2012). However, we focus in this paper on the validation and repair of planar partitions only and we assume that the input polygons are individually valid.

2.2  Validation of a Planar Partition Using a Planar Graph

Assuming that individual polygons have been deemed to be valid, it is possible to test the validity of a planar partition by identifying the two types of invalid configurations: overlaps and gaps.

If individual polygons are checked without building a planar graph or an indexing structure, finding overlaps involves checking whether any possible pair of polygons overlap. This is a computationally expensive operation to make because of its quadratic behaviour, even when heuristics to speed up the process are used (Badawy & Aref 1999, Kirkpatrick et al. 2002). Additionally, robustness issues are significant in polygon intersection tests (Hoffmann et al. 1988). Finding the potential gaps in a planar partition is even more problematic. For this, computing the union of the entire set of polygons is required, which is also computationally expensive (Margalit & Knott 1989, Rivero & Feito 2000).

The validation process can be sped up by first building a planar graph of the input polygons, which is afterwards checked for consistency. It should first be noticed that while different approaches are available to construct a planar graph (Shamos & Hoey 1976, van Rensburg 1991), it is still sometimes difficult, especially if the polygon contains holes. The graph
of the boundary can then be unconnected and extra machinery is necessary to represent the
knowledge of holes in the graph structure. The
fact that holes are also allowed to touch com-
plies the task of validation even further, since holes cannot be assumed to form an un-
connected planar graph.

Based on this graph, Plümer & Gröger
(1997) specify a list of minimal mathemati-
cal axioms that can be used to check the va-
Aria of a planar partition: no dangling edges,
no zero-length edges, planarity, no holes, no
self-intersections, no overlaps, and having a
connected graph. It is important to note that
Plümer & Gröger base their axioms on con-
cepts from graph theory, but they also high-
light the fact that a graph-based approach
alone is not enough: the graph has to be aug-
mented with geometrical knowledge (each
vertex has geometry attached, i.e., the coor-
dinates of points have to be stored). Validation
is thus underpinned by both geometrical and
topological concepts and systems thus have to
deal with those two concepts at the same time.
The method we propose in this paper — using
a constrained triangulation — permits us to do
exactly this: to embed both geometry and to-
ology in the same structure.

2.3 Repair Using Point and Edge
Snapping and Splitting

The most common method for planar par-
tition repair is based on the assumption that
polygons approximately match each other at
their common boundaries. This implies that
they should be within a certain distance of
each other along those edges. If, addi-
tionally, all parts further apart than this value are
known not to be common boundaries, it is
possible to “snap” together polygons that are
closer to each other than this threshold, while
keeping the rest untouched. This method of
planar partition repair is available in many
GIS packages, including ArcGIS (ArcGIS
2012), FME (FME 2012), GRASS (GRASS
2012) and Radius Topology (ISpatial 2012).

Since thresholds are central to this meth-
od, it is of utmost importance to select a good
threshold value, something that is completely
different in each dataset. For planar partition
repair to be successful using this method, such
a threshold should be chosen in a careful man-
ner, and always comply with a few conditions.
These have been summarised as follows:

- Adjacent polygons should not be further
  apart than this threshold along any part of
  their common boundaries (shown as the
  maximum threshold in Fig. 2 (a)). Other-
  wise, gaps cannot be fixed.
- Adjacent polygons should not overlap each
  other in areas which are further inwards

![Diagram](image-url)  
**Fig. 2:** Defining a threshold for vertex, edge and face snapping. The threshold to be used should be larger than the largest minimum distance between the matching boundaries, and smaller than the minimum distance between vertices.
than this threshold from their common boundaries shown as the maximum threshold in Fig. 2 (b). Otherwise, overlaps cannot be fixed.

- None of the vertices of a polygon should be closer to each other than this threshold, including non-consecutive vertices shown as the minimum thresholds in Fig. 2. Otherwise, they might be snapped together, creating repeated vertices, disjoint regions, or various topological problems.
- None of the vertices of a polygon should be closer than this threshold to any non-incident edge. Otherwise, they might be snapped together, creating disjoint regions or various topological problems.

This threshold value is usually manually determined, either by trial and error, or by analysing certain properties of the dataset(s) involved, e.g. point spacing, precision, or map scale. However, it is often hard to find an optimal threshold for a certain dataset, since ensuring that it works well for every part of a dataset is unrealistic. Moreover, sometimes such a threshold does not even exist, e.g. because point spacing in some places might be smaller than the width of the gaps and overlaps present.

Since the aforementioned conditions are frequently not met or are not checked beforehand, and since it is still necessary to perform repair of a dataset, snapping is often per-

---

**Fig. 3:** Spikes and punctures can be created by snapping, since the bases of these elongated forms (encircled) might be narrower than the threshold, but its length is not.

**Fig. 4:** Polygons can be split by snapping, since some parts of them might be narrower than the threshold (encircled). While this result does not create an invalid planar partition, it can change the number of polygons present and their topological relations, and can therefore be undesirable.
formed nevertheless, possibly creating invalid polygons and/or planar partitions, or significantly changing the topology of the existing features. Two examples of this phenomenon are shown in Figs. 3 and 4.

While these examples show that snapping is not problem-free, it is important to note that commercial GIS packages often implement more complex snapping options such as point-to-edge, edge-to-edge, or using a reference dataset. These options can help to solve a problematic case, but can also have undesired consequences, such as changing the topology of the polygons. Another problem is that post-processing operations to clean resulting polygons might be required, e.g. disposing of polygons with small areas, removing redundant lines, thresholds for minimum angles etc., which could again create invalid configurations, requiring iterative validation or repair processes.

### 2.4 Repair Using Topological Information

A different approach for planar partition repair, based on topological information, is available in some software.

GRASS also creates a graph, using edges as a base structure instead of triangles, and could be used to (manually) detect overlaps and gaps based on the number of labels using the functions v.what and d.what.vect. However, there is no simple automated procedure to get the number of labels at a certain point, which makes it very cumbersome and time consuming to use GRASS for this purpose.

Meanwhile, ArcGIS provides a more complete solution, using a method similar in some ways to the one developed and described in section 3. It involves using the Geodatabase feature of the software with some combined validation rules, e.g. must not overlap and must not have gaps. However, fixing every problematic area in an appropriate manner requires extensive user intervention (Fig. 5), since the best choice for each case depends on the specific configuration of the error.

Since both the aforementioned programs do not offer an automated process to correctly solve this problem and GRASS lacks the ability to visualise problem areas, they are not really comparable to our solution. A planar partition can easily contain tens of thousands of polygons, possibly generating thousands of errors, which need to be checked and repaired semi-automatically.

---

(a) Viewing a topology error in ArcGIS

(b) Assigning an overlapping region to one of the polygons involved

**Fig. 5:** Planar partition repair in ArcGIS. The user is expected to zoom in to a particular error, analyse the situation, e.g. by looking at the properties from the surrounding polygons, and make a decision to assign the problematic region to a certain polygon. More than 11,000 errors were detected in this tile of the Corine dataset.
3 Validation and Automatic Repair Using a Constrained Triangulation

Our approach to validation and automatic repair of planar partitions uses a constrained triangulation (CT) as a supporting structure because it has many good properties, including the following:
- It is by definition a planar partition. Therefore, as long as we keep the information about which polygon each triangle belongs to, the reconstructed polygons will be either a valid planar partition, or multiple ones.
- It can be built quickly, in $O(n \log n)$ with a variety of approaches (Clarkson et al. 1992, Guibas & Stolfi 1985, Mucke et al. 1999). The actual computational complexity can be $O(n \log n) + k$, with $k$ being the number of edge-edge intersections, which could conceivably even be $n^2$. However, $k \ll n$ for most GIS datasets.
- Changes to the triangulation, e.g. adding a new constrained edge is local, and therefore fast.
- Constrained edges can usually be added in constant time, being only significantly slower and more complex when there is an intersection with an existing constrained edge (Shewchuk 1997b).
- Implementation-wise, several stable and fast triangulation libraries exist, including CGAL (2012), Triangle (Shewchuk 1997a) and GTS (2006).

The general workflow of our approach to both validate and repair a planar partition is as follows:
- the CT of the input segments forming the polygons is constructed;
- each triangle in the CT is labelled with the label of the polygon inside which it is located;
- problems are detected by identifying triangles having no or multiple labels, and by verifying the connectivity between triangles;
- repairing of the problems is made by re-labeling triangles to ensure that each triangle has exactly one label;
- extracting the polygons from the triangulation (polygons modelled with the Simple Features specifications).

As mentioned previously, for this workflow we assume that each input polygon is individually valid. We describe in the following section the concepts needed and we give a detailed description of the different steps.

3.1 Triangulation of a Polygon and Constrained Triangulation

A triangulation subdivides an area into non-overlapping triangles. Using a constrained triangulation, every line segment that defines the boundary of a polygon, is ensured to appear as an edge in the triangulation. It is known that

![Fig. 6: (a) A polygon with 4 holes; (b) the constrained triangulation of the segments of this polygon.](image)
any polygon (also with holes) can be triangulated without adding extra vertices (de Berg et al. 2008, Shewchuk 1997a). Fig. 6 shows an example.

In our approach, the triangulation is performed by constructing incrementally a CT of all the segments representing the boundaries (outer + inner) of each polygon. If the set of input polygons forms a planar partition, then each segment will be inserted twice except those forming the outer boundary of the set of input polygons. This is usually not a problem for triangulation libraries because they ignore points and segments at the same location as is the case with the solution we use, see section 4. When segments are found to intersect, they are split with a new point created at the intersection. This is the only situation in which the generation of new points is required. Notice that our approach requires only a constrained triangulation, which not necessarily includes the Delaunay criterion (Shewchuk 1997a). However, having well-shaped triangles is useful for repair purposes and does not significantly increase the processing time. Therefore our implementation actually constructs and uses a constrained Delaunay triangulation.

3.2 Labelling the Triangles of a Planar Partition

Labelling a triangle means assigning to it the label(s) of the polygon(s) that it belongs to. If two input polygons overlap, each triangle in the overlapping region should have the labels of the two polygons.

In our previous work on validation (Ledoux & Meijers 2010), we used the centroid of a polygon to start the labelling process, but this method is prone to errors if for instance the calculated centroid is outside or on the boundary of the polygon and does not allow us to differentiate between gaps and overlaps.

To solve these problems, we store information about the constrained edges of the CT. Since it is known that the input rings are closed and have a known orientation according to the ISO 19107 orientation rules, it is also known on which side of a certain line segment the interior of the polygon lies. This property is used for robust labelling of each polygon. Triangles adjacent to the outer ring of the polygons are labelled first, and this is later expanded to triangles further in the interior of the polygon, recursively labelling adjacent unlabelled faces as long as no constrained edges are crossed. After this operation, all triangles that are part of any polygon have been labelled, with overlapping regions having multiply labelled triangles. However, holes are then indistinguishable from triangles outside the planar partition, since both have zero labels. Therefore, a special label is created for all triangles outside the planar partition, referred to as the “universe” label, which are labelled by recursively labelling adjacent triangles from any triangle known to lie in the exterior of the planar partition. To achieve this we exploit the concept of the “far-away point” (Liu & Snoeyink 2006); which is used by several implementations and is also known as the “big triangle” (Facello 1995).

3.3 Validation

If the set of input polygons forms a planar partition then all the triangles will be labelled with one and only one label. The problems are easily detected:

- **Gaps** are detected by finding triangles without any labels.
- **Overlaps** are detected by finding triangles with two or more labels.
- **Disjoint regions** are detected by identifying regions separated by the “universe” label. This is done by starting at a given triangle, and doing a breadth-first search on the dual of the triangulation, without visiting the triangles labelled as “universe”. If all triangles can be reached, no polygons of the planar partition are disjoint.

3.4 Repair Operations

The greatest benefits of using a labelled triangulation for planar partition repair stem from the fact that while repair operations are performed, the validity of the planar partition is always kept, together with the integrity of the data. Unlike snapping, vertices are not re-
(a) A set of overlapping polygons that needs to be repaired.
(b) The polygons are triangulated and each triangle is labelled with the polygons it belongs to.
(c) An invalid region is found, defined as a set of contiguous polygons with an equivalent set of 0 or ≥2 labels.

<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>

(d) The neighbours of the region are analysed to see how it should be re-labelled, and the result is saved in a list.
(e) The triangles are relabelled according to the list.
(f) The output polygons, which now have no gaps or overlaps.

Fig. 7: The steps in a generic repair operation.
Tab. 1: The repair operations currently implemented in our software. The types of operations are defined according to the map algebra classification by Tomlin (1994).

<table>
<thead>
<tr>
<th>Repair operation</th>
<th>Type</th>
<th>Criteria</th>
</tr>
</thead>
<tbody>
<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>
<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>
</tbody>
</table>

Fig. 8: Different repair operations used in the two polygons for the Arribes del Duero Natural Park in Spain (red) and the International Douro Natural Park in Portugal (green). All of them can be considered best by a certain criterion, like (b) preserving the area ratio between the two polygons, (c) smoothness of the boundary, or (d) a balance between the two.
be further developed. For instance, triangles could be split to subdivide an area with problems as in Bader & Weibel (1997), or sliver triangles/regions could be discarded (and then filled during repair).

3.5 Extraction of Polygons from a Triangulation

Starting from a labelled triangulation, it is possible to reconstruct the individual polygons of a planar partition to conform to valid polygons according to the ISO 19107 and the Simple Features specifications, which allows users to incorporate automatic validation and repair in their workflow.

We do this operation polygon by polygon and start at an unprocessed triangle. Then we visit all the connected triangles that have the same label, reconstruct the polygon, and mark the triangles as processed. Note that since all of these triangles are connected, the outer and inner boundaries of a polygon are all simple, i.e., non self-intersecting. We repeat this operation until all triangles have been processed.

For each polygon, we have to recover not only its outer boundary, but also its inner boundaries, which are not connected. Observe that we cannot simply follow the original constrained edges as these do not have any meaning after a planar partition was repaired; the boundaries of the repaired polygons are instead formed by edges incident to two triangles having different labels.

For each polygon, we start at a triangle, and move on to triangles having the same label. As the process goes on, a single polyline that runs along all boundaries of the polygon is generated. This involves a depth-first search (clockwise) that recursively reaches until the boundary of a polygon, returning a long chain of edges in a procedure similar to following the boundary edge by edge. The procedure is shown step by step in Fig. 9.

The polyline created with this method has “bridges”, which allow us to keep all inner boundaries (holes) connected with the outer boundary, in a manner that keeps the interior connected as well. These help to preserve connectivity and the relations between different boundary types (outer and inner), but are re-
Fig. 9: The traversal order when navigating through triangles in a clockwise manner. Starting at \( \triangle A \) and the edge between \( \triangle A \) and \( \triangle B \), the operations occur according to the encircled numbers. Black arrows denote when an unprocessed triangle is found, red arrows when it is not. Notice how the traversal is performed clockwise (shown in dark blue arrows) for both cases, despite starting from different sides.

Fig. 10: Processing the polyline shown starting from vertex \( a \) and moving towards polyline \( \mathcal{P} \), the following operations occur: (1) \( \mathcal{P} \) unclosed \( \rightarrow \) push, (2) \( H \) unclosed \( \rightarrow \) push, (3) \( C \) unclosed \( \rightarrow \) push, (4) \( D \) unclosed \( \rightarrow \) push, (5) \( E \) unclosed \( \rightarrow \) push, (6) \( F \) closed \( \rightarrow \) store, (7) pop \( \rightarrow \) EG closed \( \rightarrow \) store, (8) pop \( \rightarrow \) DH degenerate \( \rightarrow \) erase, (9) pop \( \rightarrow \) CI closed \( \rightarrow \) store, (10) pop \( \rightarrow \) BJ degenerate \( \rightarrow \) erase, (11) pop \( \rightarrow \) AK closed \( \rightarrow \) store.
moved later in the process to conform to the Simple Features specifications. Also, its orientation conveys the information of whether a section of it is part of an inner or an outer boundary.

This polyline is processed with a stack-based algorithm that generates separate closed rings for the outer boundary and each of the inner boundaries, collapsing the “bridges” that were generated. In order to do this, the polyline is cut at the positions where more than two edges join, and these are joined in the correct order by keeping track of (yet) unclosed rings. When a new segment is processed, it can be one of three options: one that completes a ring, one that is part of a bridge, or one that is not yet closed. Closed rings are stored and bridges are removed, while unclosed rings are saved in the stack until they can be popped to form a closed ring together with a new segment. This is shown in Fig. 10.

4 Implementation and Experiments

4.1 Implementation

An implementation of the algorithms described in section 3 was written in the C++ programming language, using external libraries for some functionality. The developed software is called pprepair, and is open source and freely available at http://tudelft-gist.github.com/pprepair/. C++ was selected in order to have good control with regards to low level details and to achieve good performance, which makes it possible to compare it with existing solutions. The libraries used are: the OGR Simple Features library (GDAL 2012) which allows input and output from a large variety of data formats common in GIS; and CGAL (CGAL 2012) which has support for many robust spatial data structures and the related operations; we use its constrained triangulation module.

4.2 Experiments with Real-World Planar Partition Datasets

We have made experiments with four freely available real-world datasets, i.e., we have validated and automatically repaired them with our implementation; the overview of these datasets is shown in Fig. 11 and their properties in Tab. 2. The datasets are the following:

- **E41N27** Corine 2000 (Corine 2000) tile E41N27, which contains a shifted polygon (by about 10 cm), creating many small gaps and overlaps in the dataset. The snapping threshold has been set to 1 m.
- **4tiles** Corine 2000 tiles E39N32, E39N33, E40N32 and E40N33, which are known to have long and thin overlapping regions (1 mm) with each other. The snapping threshold has been set to 1 cm.
- **16tiles** 16 adjacent Corine 2000 tiles: E39N30, E39N31, E39N32, E39N33, E40N30, E40N31, E40N32, E40N33, E41N30, E41N31, E41N32, E41N33, E42N30, E42N31, E42N32, E42N33. Between some of them are gaps, others are overlapping. However, all match within a few centimetres. The snapping threshold has been set to 10 cm.
- **Mexico** 1:1,000,000 scale land cover data-set from INEGI consists of over 26,000 polygons. It is mostly valid according to the Shapefile specification, but contains some very large polygons, with tens of thousands of vertices, which presents difficulties to most existing software.

As a comparison, we have also tried to perform the same operations with other available software. While the capabilities for pla-

<table>
<thead>
<tr>
<th></th>
<th># polygons</th>
<th># pts</th>
<th># pts in 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>33.7</td>
</tr>
<tr>
<td>4tiles</td>
<td>4,984</td>
<td>365,702</td>
<td>16,961</td>
<td>74.7</td>
</tr>
<tr>
<td>16tiles</td>
<td>63,868</td>
<td>6,622,133</td>
<td>95,112</td>
<td>103.7</td>
</tr>
<tr>
<td>Mexico</td>
<td>26,866</td>
<td>4,181,354</td>
<td>117,736</td>
<td>155.6</td>
</tr>
</tbody>
</table>
anchor partition repair among the software tested vary considerably, with full topological repair only available in ArcGIS (using manual operations only), it is also important to consider how different repair implementations scale to large datasets. For this, a few performance tests were made in our implementation, and three planar partition repair tools that perform this process using snapping and splitting. The testing methodology for each tool is as follows:

ArcGIS In ArcCatalog, a multiple feature dataset is created in a Geodatabase, set with XY resolution values equal to the snapping threshold. The features are imported into it and the merge and dissolve operations are used to merge adjacent polygons with the same ID. Topology is then generated to check that the planar partition is valid. Everything is finally exported to a single Shapefile. The individual parts of the process are timed and the total is recorded. Memory usage is calculated as the difference between the just load-
ed ArcCatalog application and its maximum memory usage throughout the process.

**FME**  
A reader is created for each input file, which serves as input to a Snapper transformer; features with the same IDs are then dissolved, and finally they are output into a new Shapefile writer. The topology generator is used to be able to tell whether the result is a valid planar partition. Results are timed and the maximum memory allocation of the fme.exe process is recorded.

**GRASS**  
Input files are imported with v.in.ogr, with all polygon cleaning operations performed and snapping set to the correct values. Boundaries between features with the same IDs are then dissolved using v.dissolve. Files are then exported with v.out.ogr. Times reported by GRASS are added together to give the total, while memory usage by the v.in.ogr.exe, v.dissolve.exe and v.out.ogr.exe are monitored and their maximum is recorded.

**pprepair**  
Files are read and put into the triangulation, the triangulation is labelled, the repair is performed with the longest boundary first (see Tab. 1), with ambiguous cases resolved with a random choice. Polygons are then extracted from the triangulation and output to a single Shapefile. The entire process is timed, and the maximum amount of memory used is recorded.

Notice that we are somewhat comparing apples with oranges here since our implementation is able to repair more cases than other methods, keeps topological consistency, and does not require finding out the appropriate threshold value (if it exists). More importantly, it is able to directly state whether the result is a planar partition, unlike the three other solutions (with these snapping is performed but that does not guarantee that the output is a valid planar partition). The results of the experiments are shown in Tab. 3.

We have made the experiments in order to have an idea of the processing time and the memory usage involved. As Tab. 3 shows, our approach uses somewhat more memory than other solutions. This is explained by the extra (unconstrained) edges that are added to the input over when triangulating them. It should however be noticed that both ArcGIS and GRASS crashed with the biggest dataset 16tiles. Our implementation is the fastest of the four tested, being for instance around three times faster than FME for the biggest datasets. Only for the Mexico dataset is our implementation slower than FME. This is (probably) explained by the fact that its polygons already form a valid planar partition, and therefore very few snapping operations have been performed by FME; the planar graph of the input was basically built, and then the polygons saved back to disk. We believe that ArcGIS and GRASS struggled with the dataset because it contains several very large polygons (with more than 10,000 vertices). Our implementation took 3m31s, but, as Tab. 4 shows, it took CGAL 3m02s to simply triangulate the input edges.

This table also demonstrates the efficiency of our approach: for the four datasets, around 85% of the time of our approach was used – by the CGAL library – to read the input from disk and triangulate the input edges.

During the implementation, we took several engineering decisions to optimise our pro-

<table>
<thead>
<tr>
<th>Dataset</th>
<th>pprepair Memory</th>
<th>FME Memory</th>
<th>GRASS Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>E41N27</td>
<td>145 MB</td>
<td>158 MB</td>
<td>59 MB</td>
</tr>
<tr>
<td>4tiles</td>
<td>145 MB</td>
<td>158 MB</td>
<td>59 MB</td>
</tr>
<tr>
<td>16tiles</td>
<td>1.45 GB</td>
<td>636 MB</td>
<td>crashes</td>
</tr>
<tr>
<td>Mexico</td>
<td>1.01 GB</td>
<td>264 MB</td>
<td>408 MB</td>
</tr>
</tbody>
</table>

This table also demonstrates the efficiency of our approach: for the four datasets, around 85% of the time of our approach was used – by the CGAL library – to read the input from disk and triangulate the input edges.
age compared to a pure graph-based approach – this difference is mainly caused by the unconstrained edges introduced by the triangulation. While it would be possible to use our repair rules together with a (primal/dual) graph-based approach, these additional edges in the triangulation give fine-grained control over the repair operations, and ensures that the graph is connected, which facilitates the reconstruction of polygons.

We have implemented our algorithm over a numerically robust triangulator (CGAL) and since repair operations are expressed solely in terms of re-labelling of triangles (no geometric computation is involved), the approach is also fully robust. Since, during our experiments, most of the time was used to compute the constrained triangulation, another library could also be tested to improve the implementation.

For the future, we plan to:

● Improve the scalability of the approach and process datasets with more than 10 million polygons. It is known that using divide-and-conquer techniques triangulation algorithms can handle big datasets (Amenta et al. 2003, Blanford et al. 2005). We will investigate whether it is possible to automatically repair each divided part individually and ‘glue’ the repaired parts together.

● Investigate snap rounding (de Berg et al. 2007, Höf 1999) as a pre-processing step – or embedded directly in the triangulation – to guarantee that repaired planar partitions have no vertices that are closer than a certain threshold. However, snap rounding may change the topology of the input, but the output will nevertheless be a valid partition as the topology will be repaired. Apart from topological changes, snap rounding can also lead to removed polygon parts that are too small to be preserved based on the chosen $\varepsilon$.

● Add more advanced repair operations to our repair toolkit, e.g. repair could take place based on splitting a collection of triangles.

● Extend our work to include the third dimension to validate and repair 3D city models using a constrained tetrahedralisation (Si 2008). Notice that the tetrahedralisation of a given polyhedron does not always exist, and thus extra (Steiner) points might need

---

**Tab. 4:** Timed steps of the planar partition repair procedure, rounded to the nearest second. The percentage is the triangulation time over the total time.

<table>
<thead>
<tr>
<th></th>
<th>E41N27</th>
<th>4tiles</th>
<th>16tiles</th>
<th>Mexico</th>
</tr>
</thead>
<tbody>
<tr>
<td>Triangulate</td>
<td>0:17</td>
<td>0:15</td>
<td>4:00</td>
<td>3:02</td>
</tr>
<tr>
<td>Label</td>
<td>0:01</td>
<td>0:01</td>
<td>0:27</td>
<td>0:16</td>
</tr>
<tr>
<td>Repair</td>
<td>0:00</td>
<td>0:00</td>
<td>0:00</td>
<td>0:00</td>
</tr>
<tr>
<td>Reconstruct</td>
<td>0:01</td>
<td>0:01</td>
<td>0:11</td>
<td>0:07</td>
</tr>
<tr>
<td>Output</td>
<td>0:00</td>
<td>0:00</td>
<td>0:10</td>
<td>0:06</td>
</tr>
<tr>
<td>Total</td>
<td>0:19</td>
<td>0:17</td>
<td>4:47</td>
<td>3:31</td>
</tr>
<tr>
<td>triangulate %</td>
<td>89%</td>
<td>88%</td>
<td>83%</td>
<td>86%</td>
</tr>
</tbody>
</table>
to be added. The main concepts of our approach, (re)labelling and reconstruction, extend naturally to 3D. However, appropriate repair operations for 3D city models still need to be defined, and some application-specific constraints, e.g. right angles at corners are not trivial to implement in our current approach.

References


Address of the Authors:

KEN ARROYO OHORI, Dr. HUGO LEDOUX & Dr. MARTIJN MEIJERS, Section GIS Technology, OTB Research Institute for the Built Environment, Delft University of Technology, Jaffalaan 9, 2628 BX Delft, The Netherlands, e-mail: {g.a.k.arroyoohori} {h.ledoux} {b.m.meijers}@tudelft.nl

Manuskript eingereicht: November 2011
Angenommen: April 2012