A triangulation-based approach to automatically repair GIS polygons

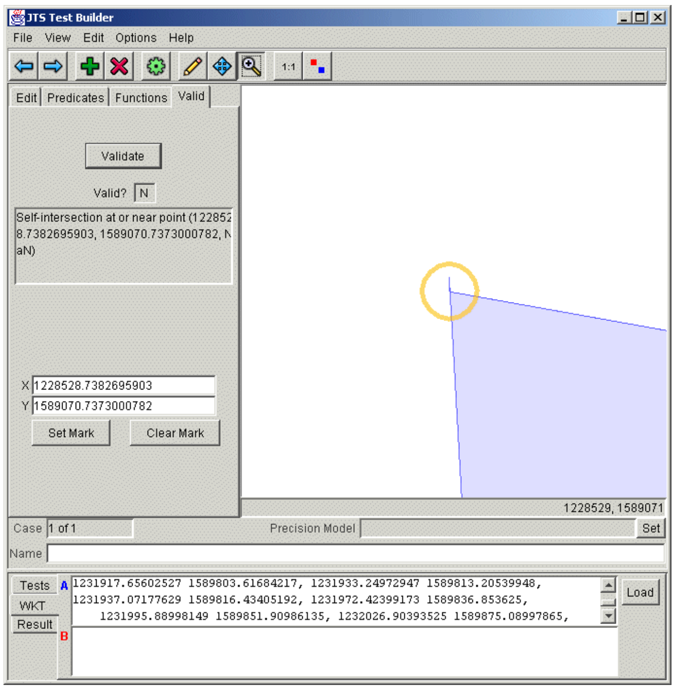
Hugo Ledoux
h.ledoux@tudelft.nl

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

Martijn Meijers
b.m.meijers@tudelft.nl

This is the draft author's version of the paper. The definitive version is published in the journal *Computers & Geosciences*, volume 66, pages 121–131, 2014. (http://dx.doi.org/10.1016/j.cageo.2014.01.009)
http://dx.doi.org/10.1016/j.cageo.2014.01.009
The code of the project is available at:
https://github.com/tudelft-gist/prepair

Although the validation of a single GIS polygon can be considered as a solved issue, the repair of an invalid polygon has not received much attention and is still in practice a semi-manual and time-consuming task. We investigate in this paper algorithms to *automatically* repair a single polygon. Automated repair algorithms can be considered as interpreting ambiguous or ill-defined polygons and returning a coherent and clearly defined output (the definition of the international standards in our case). We present a novel approach, based on the use of a constrained triangulation, to automatically repair invalid polygons. Our approach is conceptually simple and easy to implement as it is mostly based on labelling triangles. It is also flexible: it permits us to implement different repair paradigms (we describe two in the paper). We have implemented our algorithms, and we report on experiments made with large real-world polygons that are often used by practitioners in different disciplines. We show that our approach is faster and more scalable than alternative tools.



Valid? N

```
Self-intersection at or near point (122852
8.7382695903, 1589070.7373000782, N
aN)
```

X 1228528.7382695903
Y 1589070.7373000782

1228529, 1589071

Case 1 of 1 Precision Model

Name

```
A 1231917.65602527 1589803.61684217, 1231933.24972947 1589813.20539948,
  1231937.07177629 1589816.43405192, 1231972.42399173 1589836.853625,
      1231995.88998149 1589851.90986135, 1232026.90393525 1589875.08997865,
B
```

Figure 1: The Java Topology Suite (JTS) interface that helps users locate errors in invalid polygons.

1 Introduction

While there are different definitions for a polygon, most geographical information systems (GISs) use that of the Open Geospatial Consortium (OGC) and the International Organization for Standardization (ISO)[1] (OGC, 2011; ISO, TC211), and provide validation functions to ensure that a given polygon conforms to the definition. There are small variations between different implementations (van Oosterom et al., 2004), but we can consider the validation of a two-dimensional polygon a solved problem. Having one definition together with validation tools ensures that practitioners can exchange datasets and use spatial analysis operations in their downstream applications. Validity is indeed a prerequisite for many GIS operations—invalid polygons will either yield wrong results or, even worse, could make the software crash.

When a polygon is invalid—that is, it does not respect a given definition—then one has to repair it. While most validation tools give users a list of errors and locations (see for instance Figure 1), they usually still have to *manually* fix them. This can become in practice a very tedious and time-consuming task for large polygons.

We investigate in this paper *automatic* methods for repairing GIS polygons. Surprisingly, it is a topic that so far has received little attention. As we discuss in Section 3, most GIS packages perform some form of implicit cleaning/repairing (eg deleting "unwanted parts" for display purposes) when

[1]These are almost identical, see Section 2.

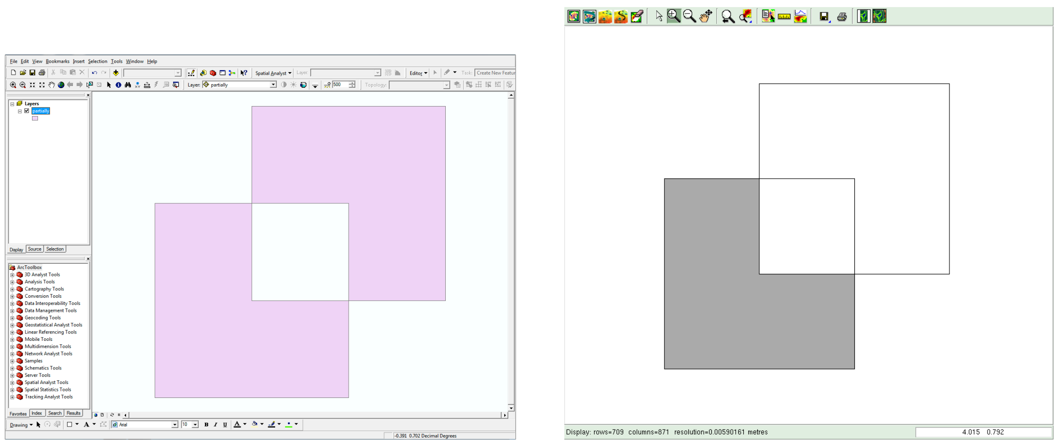

(a) (b)

Figure 2: Different interpretations of the polygon p_3, as shown in Figure 3. (a) ArcGIS considers the overlapping region as a hole, but the non-overlapping part of the hole as a new polygon (QGIS and FME do this as well). (b) GRASS removes the overlapping part from the polygon, becoming a new polygon with a different shape.

reading invalid input, but how this is done is (often) not documented and the user has little control over it. An example of this cleaning, and of how the interpretation of the input can differ, is shown in Figure 2 for two well-known packages. To our knowledge, the only fully automatic repair tool available is the one in PostGIS (the `ST_MakeValid` function). In our context, a repair tool explicitly takes a polygon as input, repairs it, and gives it back to the user; this is in contrast with the cleaning functions that are automatically used on the input as a means to enable its conversion to another (internal) representation. As explained in Section 3, `ST_MakeValid` is not documented (one has to read the code and try with different inputs) and does not perform well for polygons having a large number of boundaries. It should be said that the repair of polygons is not an exact science, ie different persons could repair one invalid polygon in different ways. As a consequence, we describe in Section 3 different algorithms and paradigms that can be used, each one of these has pros and cons. We believe that the most suitable paradigm is application-dependent.

We present in this paper a novel approach to automatically repair invalid GIS polygons. As described in Section 4, it is conceptually simple and is based on the properties of a constrained triangulation (CT) of the input polygon. Our CT-based approach permits us to implement efficiently different repair paradigms, and adding new ones is easily done. We also discuss in Section 5 a pre-processing step to our approach to snap points and lines to each other if they are within a tolerance. Doing so can destroy (modify and invalidate) the topology of the input, but we show that with our approach we can recover from these errors, and that the repaired polygons are free of spikes and more *robust*. We have implemented our approach and we report in Section 6 on experiments we ran with large and complex real-world GIS polygons used by practitioners in different disciplines related to the geosciences. It can be seen that our implementation is efficient in practice, and that it scales better than a graph-based approach (that of PostGIS's `ST_MakeValid`) for very large poly-

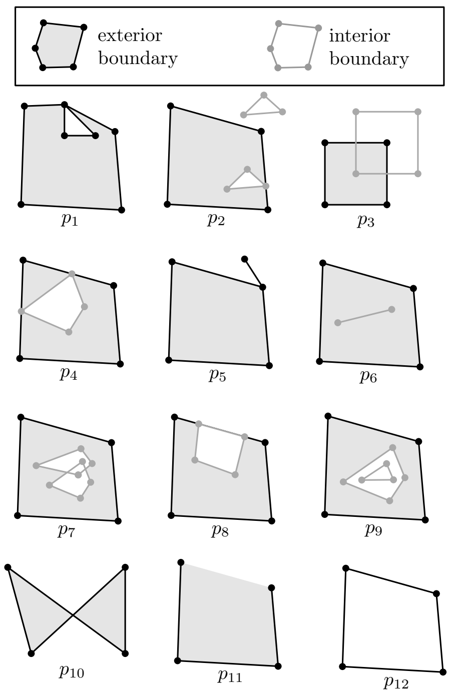


Figure 3: Several invalid polygons; this is not a complete list of all problematic polygons, but rather an overview of common cases. Polygon p_{12} has its exterior and interior rings defined by the same geometry.

gons. Finally, in Section 7, we elaborate on the advantages of our method and discuss how other paradigms can be implemented on it.

2 What is a (robust) polygon?

We use the definition as found in the Simple Features specifications (SFS) (OGC, 2011):

> *[A] planar Surface defined by 1 exterior boundary and 0 or more interior boundaries. Each interior boundary defines a hole in the Polygon.*

A boundary is defined by a (clockwise or counterclockwise) oriented ring. Different rules are provided, the most relevant being the following (examples of polygons breaking the rules are given between brackets, they refer to those in Figure 3):

1. Each ring defining the exterior and interior boundaries should be *simple*, ie non-self-intersecting (p_1 and p_{10}). Notice that this prevents the existence of rings with zero-area (p_6), and of rings having two consecutive points at the same location. It should be observed that the polygon p_1 is not allowed by the SFS (in a valid representation of the polygon, the triangle should be

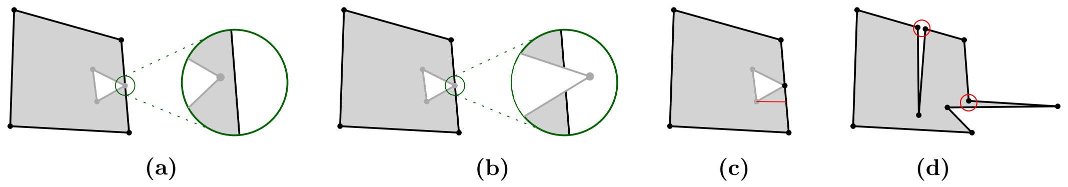


Figure 4: **(a)(b)** Two polygons, appearing to be identical, having an interior ring touching the exterior ring. However, because of the use of a finite-precision representation, the vertex cannot be located directly on the segment, and will thus be either slightly **(a)** inside or **(b)** outside the exterior ring. **(c)** A more robust representation of the polygon. Its robustness is equal to the length of the red line. **(d)** If a given minimum tolerance for a valid polygon is given (red circles), then the polygon becomes invalid.

represented as an interior boundary touching the exterior boundary), but some implementations do allow it (eg ESRI's Shapefile).

2. Each ring should be closed (p_{11}): its first and its last points should be the same.
3. The rings of a polygon should not cross (p_3, p_7, p_8 and p_{12}) but may intersect at one tangent point (the interior ring of p_2 is a valid case, although p_2 as a whole is not since the other interior ring is located *outside* the interior one).
4. A polygon may not have cut lines, spikes or punctures (p_5 or p_6); removing these is known as the *regularisation* of a polygon (a standard point-set topology operation).
5. The interior of every polygon is a connected point set (p_4).
6. Each interior ring creates a new area that is disconnected from the exterior. Thus, an interior ring cannot be located outside the exterior ring (p_2) or inside other interior rings (p_9).

Furthermore, the exterior boundary of a polygon must be oriented counterclockwise, and the interior boundaries clockwise.

The ISO/OGC definition of a polygon assumes an implementation of the rules with an arbitrary-precision representation (real numbers), while most commonly these are done using floating-point, which offers only an approximation (Hoffmann, 1989). The coordinates of the vertices of a polygon are therefore most often rounded to the closest possible value in the computer. This can have serious consequences as the topology of a polygon can be modified and a valid polygon can become invalid; Figure 4 shows an example. In the ISO/OGC rules, if two rings touch at location q, only one of the rings is required to have a vertex at location q. If both rings had a vertex at location q (if a fifth vertex was added to the exterior ring of the polygon in Figure 4a), this problem would be avoided.

To facilitate operations (including validation) on polygons when finite-precision representations is used, Van Oosterom et al. (2004) define the concept of *robustness* of a polygon. Each vertex of a polygon is assigned a tolerance: the maximum distance this vertex can be moved (in any direction) while the polygon is guaranteed to remain valid. As an example, the polygon in Figure 4a is not

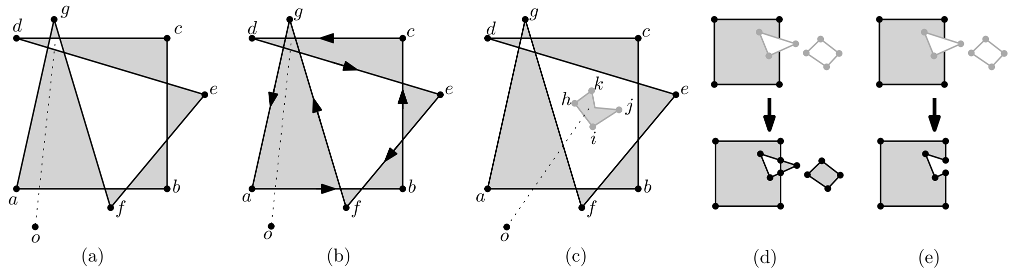


Figure 5: Three non-simple polygons; grey represents the interior of the polygon. **(a)** The odd-even rule. **(b)** The non-zero winding rule. **(c)** The odd-even rule when applied to a polygon having an interior ring. **(d)** Top: input polygon. Bottom: result of the odd-even rule algorithm is formed by 3 polygons (having no interior rings). **(e)** Top: input polygon. Bottom: result of the non-zero winding rule algorithm is formed by 1 polygon.

very robust, but if, as shown in Figure 4c, the exterior ring explicitly had a vertex where both rings touch than the robustness would be greatly increased. Van Oosterom et al. (2004) use the tolerance for validation: a polygon is valid only if it respects the ISO/OGC rules and if its robustness is greater than a given user-defined tolerance (see Figure 4d). Observe here that this definition has the advantage of not allowing 'spikes' in polygons. As Figures 4c and (d) show, calculating the robustness implies calculating distances between vertices and other vertices, and between vertices and edges. We describe in Section 5 one method to improve the robustness of polygons and to remove spikes from polygons.

3 Related work

3.1 Algorithms to identify the interior of a polygon

The automatic repair of a polygon is akin to the identification of the interior of a polygon. Given one simple and closed boundary in the plane, finding its interior is straightforward since, as the Jordan curve theorem states, the boundary divides the plane into two distinct regions: the interior and the exterior (Jordan, 1887).

If the boundary is non-simple then there are two commonly used algorithms in vector-based graphic software (Foley et al., 1996): (i) the odd-even rule; (ii) the non-zero winding rule. In brief, both approaches first require us to construct the planar graph G of the boundary to identify faces, and then two similar rules are used to determine whether a face is an interior or an exterior region of the boundary. Figure 5 illustrates both approaches. With the odd-even rule, a face F is an interior region if for any point p inside F the segment po (where o is a distant point located outside the boundary) intersects an odd number of edges in G. The non-zero winding rule counts the number of times the boundary makes a full revolution around a point p in a given direction (changing direction can cancel out previous rotations), let us assume counter-clockwise. If the count is non-zero, p is located inside. It is implemented by adding 1 when the segment po, as above, intersects an edge of G that is oriented from left to right, and subtracting 1 when the edge is in the other direction; p is an interior point when the count is non-zero.

Both algorithms can be generalised to GIS polygons (ie having interior rings). Figure 5c shows one polygon having one interior ring (light grey boundary) whose interior has been defined as the interior of the polygon with the odd-even rule (the input polygon has actually been split into several polygons). While this behaviour is predictable (so a practitioner can predict easily how her polygons will be repaired), it is perhaps not suited for all applications, especially when rings overlap. As shown in Figure 5d, with the odd-even rule all rings are considered equal and interior rings become new polygons when located outside the exterior ring. We describe in Section 4 an implementation of this paradigm where degeneracies (eg when two rings are sharing an edge, or when they overlap) are handled.

We also propose in Section 4 an alternative approach to odd-even in which the interior of interior rings is always considered as the exterior of the polygon. This offers another option for practitioners, one where the information of whether a ring is inner or outer is deemed to be correct. As shown in Figure 5e, it follows a point-set topology approach in which a polygon p having an exterior ring r and n interior rings r_i (where $0 \leq i \leq n$) is defined as $p = r \setminus (r_0 \cup r_1 \cup \ldots \cup r_n)$. We are not aware of other software implementing this approach explicitly.

3.2 How practitioners repair their invalid polygons

As seen in the Introduction, most GIS packages have algorithms to automatically clean invalid polygons. These algorithms are usually used implicitly as soon as an invalid geometry is read by the software so that the geometries can be stored in a specific data structure and then drawn on the screen. As shown in Figure 2, the rules for the handling of extreme cases differ greatly from one package to the other. While there exist also specific functions to repair invalid polygons (eg in ArcGIS one can define rules based on a tolerance for the allowed distances between rings and how they can interact), these are usually applied *after* the invalid geometry has been cleaned automatically. Therefore, the user has no direct control over these (some parts of a polygon could be unpredictably deleted), and perhaps worse, these are not documented.

To *explicitly* repair polygons automatically, ie in a manner where the output can be controlled and/or predicted, practitioners often resort to *ad hoc* solutions and tricks. Ramsey (2010) gives an excellent overview of these, his examples are PostGIS-related only, but since it uses other open-source libraries such as GEOS we believe this is representative of what practitioners do. The most known is the “buffer-by-0” operation: a buffered geometry is built, constructed by offsetting lines from the original geometry by nothing (zero). To construct a buffer, the planar graph of the input is built; in other words the topology is built, which will be structurally identical to the original input. While this trick works fine for solving a few simple cases (polygon p_1 in Figure 3 for instance), parts of a polygon can disappear for some input polygons: it removes half of the bow-tie of p_{10}. Repairing it correctly (ie with two polygons) requires using three functions in PostGIS[2]. All these functions are based on the construction of a planar graph of the input, and on identifying loops in

[2]`ST_ExteriorRing` + `ST_Union` + `ST_BuildArea`

this graph to form rings. Some of them reconstruct all the possible loops, while others stop after one loop has been found.

The script `cleanGeometry.sql`[3] was the first attempt to formalise the decision tree based on a given input. Unfortunately, polygons with interior rings are not properly handled.

The PostGIS function `ST_MakeValid`[4] is an attempt to build a high-level function to repair any input polygon. It uses the functions of GEOS and PostGIS, and depending on the topological and geometrical configuration of the input rings, different functions are used to repair. Basically, first a planar graph of the input is built, and then one face in the graph is found and a ring is built (at this point it is unknown if it is an exterior or an interior ring). Then, for all the other faces in the graph the resulting polygon is obtained by the symmetric difference of this ring and the one already found. Each symmetric difference requires building a new independent graph where the topological relations of the rings are extracted (to detect which ring is the exterior and which are the interior). As a consequence, `ST_MakeValid` is inefficient for input containing a large number of points and/or interior rings, as Section 6 demonstrates with real-world large polygons. Even if the function is not documented, after reading the code and testing it we can conclude that it operates according to the odd-even rule, as explained above. The main difference is that it attempts to create a valid representation of a given invalid geometry without losing any of the input vertices, ie if a ring collapses to a line segment, this line segment is also returned to the user as a separate geometry.

4 Repairing a polygon with a constrained triangulation

We present in this section a triangulation-based approach to implement the two automatic repair paradigms described in Section 3:

odd-even paradigm: the odd-even rule when interior rings are present.

setdiff paradigm: the polygon $p = r \setminus (r_0 \cup r_1 \cup \ldots \cup r_n)$, where r is the exterior ring and r_i are interior rings.

We demonstrate that both paradigms can be implemented simply and efficiently using a constrained triangulation (CT) as a supporting data structure. Because the input polygons can be invalid and thus contain special cases (eg two rings sharing an edge or partly overlapping), we have generalised the two paradigms so that they have a consistent behaviour. The overall layout of the two algorithms is very similar, and in brief has the following three steps (Figure 6 illustrates the steps for a polygon having two errors):

1. construction of the CT of the segments of the input polygon;
2. labelling of each triangle as either *outside* or *inside*;
3. reconstruction of the repaired polygon according to the SFS.

[3]Available at: `trac.osgeo.org/postgis/wiki/UsersWikiCleanPolygons`
[4]Since PostGIS version 2.0

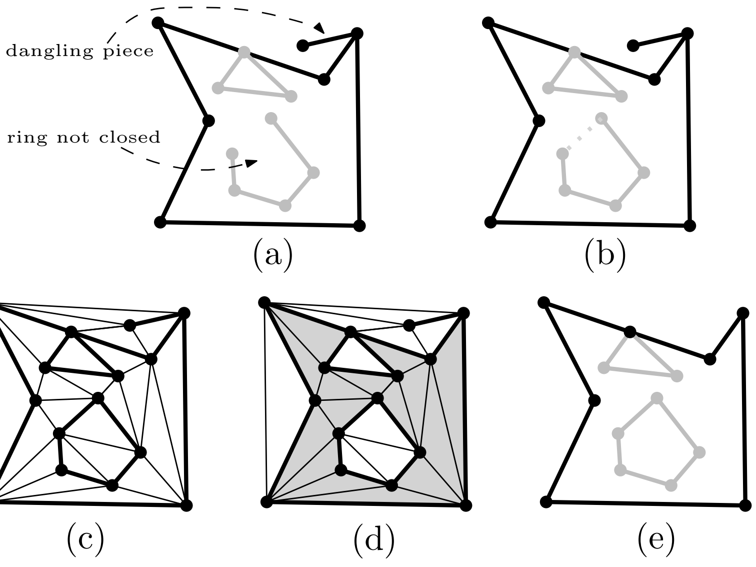


Figure 6: Workflow of our approach for repairing a polygon. In (a) the input polygon has 2 problems; (b) the interior ring is closed; (c) the CT is constructed; (d) triangles are labelled as inside (grey) or outside (white); (e) the repaired polygon.

Notice that the valid representation of an invalid input polygon can be either:

- nothing (eg the only ring of a polygon is a line segment);
- one polygon (potentially with interior boundaries);
- several polygons (eg polygons p_2, p_4, p_9 and p_{10} in Figure 3).

4.1 Properties of a constrained triangulation

Given a set S of points and (straight-line) segments in the plane (such as that in Figure 6b), a constrained triangulation (CT) decomposes the convex hull of S into triangles that are non-overlapping, and every input segment appears as an edge of CT(S). If S contains segments forming a face (which defines one boundary of a polygon in our case), it permits us to triangulate the interior of this face (ie a triangulation of the polygon). Notice here that for the sake of repairing a polygon, we cannot use algorithms to triangulate a single polygon (eg Chazelle (1982)) as these often assume that the input is simple and forms only one polygon.

Observe that while the shape of the triangles constructed is important for many applications (Shewchuk, 1997), here any CT can be used (the constrained *Delaunay* triangulation can be used but is not necessary). A CT can be built efficiently with a variety of approaches (Guibas and Stolfi, 1985; Clarkson et al., 1992). Once the CT is constructed, it can be used for solving quickly the point-location problem (eventually with an extremely light auxiliary data structure, cf. Mücke et al., 1999), which is useful to identify double vertices and intersections of segments.

4.2 Advantages of a constrained triangulation for the automatic repair

The two algorithms described in this section can in theory be implemented with a planar graph approach where each input segment becomes one edge in the graph. While this would decrease the memory usage (since the CT contains several additional unconstrained edges), there are in practice several advantages to using a CT. First, we can exploit the properties of the CT to perform some cleaning that is otherwise rather cumbersome to implement. One example is that if two input segments intersect, they are split into two sub-segments and thus a new vertex is added at their intersection. This step is performed efficiently since the CT is used as a spatial index to identify the candidate segments intersecting; no brute-force computations or auxiliary spatial index structures are thus necessary. Another example is that no two vertices or edges of a CT can be at the same location, which means that if two identical segments are in the input, only one will be kept in the CT. Also, the handling of rings collapsed to points or lines is trivial as these have no area and are thus not labelled. Second, the CT permits us to embed together in the same structure both the geometry and the topology of the input polygons, which allows us to perform less operations when repairing. For instance, `ST_MakeValid`, an implementation of a planar graph approach, needs to perform extra geometrical operations to detect topological relationships between rings, while with a CT this is not necessary because the extra edges of the CT ensures that the graph is always connected (even when a polygon has interior rings). Third, implementation-wise, several stable and fast constrained triangulation libraries exist (including CGAL (CGAL, 2011), Triangle (Shewchuk, 1997) and GTS (GTS, 2006)) and we can simply build over them as the approach involves mostly the labelling of triangles.

4.3 Odd-even paradigm

The algorithm for the odd-even paradigm is shown in Algorithm 1. Its main steps are described in the following.

Algorithm 1 The OddEven algorithm.

```
Input:  an invalid polygon $p$ having an exterior ring $r$ and $n$ interior rings $r_i$ (where $0 \leq i \leq n$)
Output:  $p$ is valid (potentially formed by none or several valid polygons)
1: for each $r_i$ do
2:     if first vertex != last vertex then
3:         add first vertex as last vertex
4:     end if
5: end for
6: $\mathcal{T} \leftarrow$ construct CT of all segments of $r$ and the $r_i$
7: label each triangle in $\mathcal{T}$ as either *outside* or *inside*
8: reconstruct $p$ as a SFS polygon
```

Closing each ring The SFS require that the first and the last point of a ring be the same (a triangle has thus 4 points). This is in practice often ignored, and most GIS packages will recover from that (small) error by adding the missing point. We believe that this is consequent with the intention of the user: if a shape was defined as a ring, it is probably a mistake that it is not closed.

Labelling triangles To label each triangle as either outside or inside, we start at one triangle located *outside* any input ring, we label it as outside and we expand to all triangles reachable from it without passing through a constrained edge of the CT (this is akin to performing a breadth-first search (BFS) on the dual graph of the triangulation). When these are exhausted, all remaining triangles reachable by passing once through a constrained edge are known to be in its interior. From the remaining triangles, those that can be reached by passing through two constrained edges are in its exterior, and so on.

The fact that we start from the outside is key to ensuring that the algorithm performs correctly. To find a triangle located outside any ring, we exploit the "far-away point" (also called the "big triangle") that is used by several CT implementations (Liu and Snoeyink, 2005; Facello, 1995). In brief, every edge on the border of the convex hull has a triangle incident to it, and this triangle is formed by the edge and a special "infinite" point.

Thus, to label the triangles, the algorithm performs several passes. First the triangles incident to the infinite point and the reachable ones are labelled as outside. Then this operation is expanded to triangles further in the interior of the polygon (labelling them as inside). If all the triangles have been flagged (if there are no interior rings) then the process if finished, otherwise the labelling continues the same way, alternating between outside and inside, until all triangles have been labelled.

From the CT to a polygon To reconstruct the polygon(s)—according to the SFS—from the labelled CT, we need to remove all the edges (both constrained and non-constrained) whose left and right labels are the same. If we performed that operation, then the reconstruction of the polygon(s) (and the identification of the exterior and interior rings of each) would be computationally expensive. We use a more efficient alternative: we construct a path (a polyline) that runs along the boundary segments of the polygon, on the inside of it. In a nutshell, as Figure 7 illustrates, we traverse one area formed by several triangles labelled as interior (or exterior) in a depth-first search order, always going counter-clockwise. In this process, so-called 'bridges' are generated to connect the exterior and interior rings. These are later removed in a rather complex procedure that also ensures that inner and outer rings are generated and nested correctly. More details can be found in Arroyo Ohori et al. (2012).

4.4 The setdiff paradigm

The algorithm to repair a polygon according to the setdiff paradigm is shown in Algorithm 2. It is conceptually similar to OddEven, the two major differences are that: (i) each input ring must

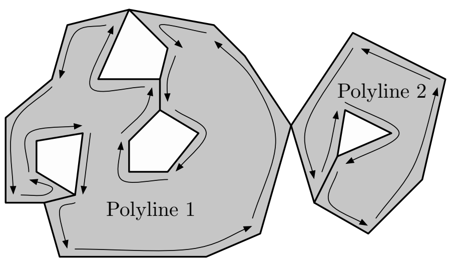


Figure 7: The polyline generated from a given triangle in the interior of the ring joins all holes with the external boundary, always while keeping the interior connected and on the same side of the line (left in this case). A separate polyline is always generated for each different interior connected component. Note that the 'bridges' generated involve passing through them twice in the polyline.

Algorithm 2 The SetDiff algorithm.

Input: an invalid polygon p having an exterior ring r and m interior rings r_i (where $0 \leq i \leq m$)
Output: p is valid (potentially formed by none or several valid polygons)

```
1: $r' \leftarrow$ OddEven
2: for each $r_i$ do
3:     $r'_i \leftarrow$ OddEven
4: end for
5: $\mathcal{T} \leftarrow$ construct CT of all segments of $r'$ and the $r'_i$
6: label each triangle in $\mathcal{T}$ as either outside or inside, taking the orientation of rings into account
7: reconstruct $p$ as a SFS polygon
```

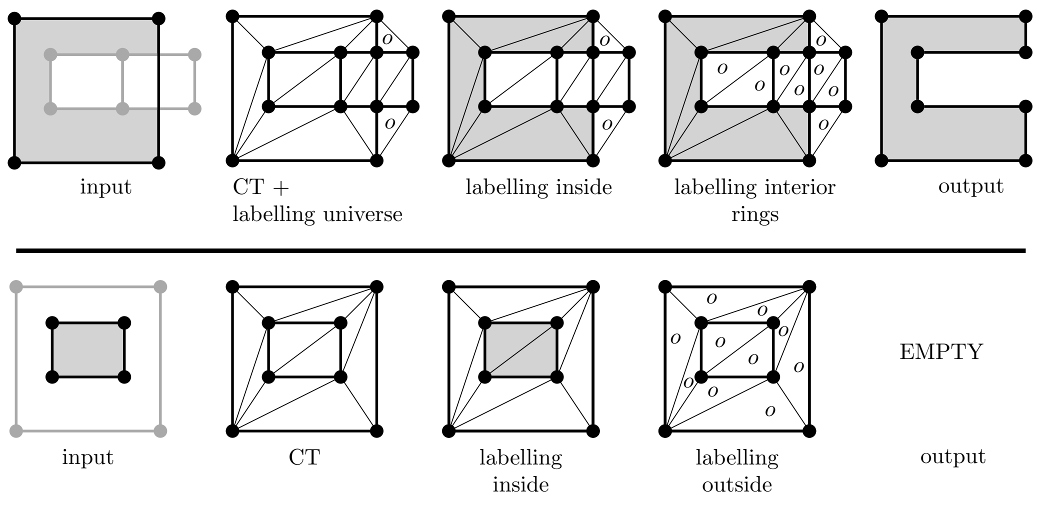


Figure 8: The steps of the SetDiff algorithm for two invalid polygons.

be repaired individually and properly identified (exterior or interior); (ii) the labelling step is performed differently. The other steps (the construction of the CT and the reconstruction of the polygon in SFS) are exactly the same as in OddEven.

Repairing each ring To ensure that every input ring is valid, the algorithm OddEven is used separately for every ring (a ring becomes a polygon for this step). If an input ring is non-simple (eg p_{10} in Figure 3) then it is split into simple rings.

Labelling triangles while considering the orientation As is the case for OddEven, one CT is constructed with the segments of all the repaired rings. Since the exterior and interior rings have to be handled differently, the constrained edges of the CT are oriented and have an attribute for the type of ring. If two rings share an edge, the information is kept for both directions of the edge.

The labelling is performed in three steps, as shown in Figure 8 for two polygons. First the triangles incident the "far-away point" are labelled as *outside*, with exactly the same procedure as with OddEven. Second, the interior triangles of the exterior ring are labelled as *inside*; during the labelling procedure the constraints in the CT are never crossed. Finally, the interior of every interior ring is labelled as *outside*; during this step, constrained edges representing the exterior ring can be crossed and triangles already labelled as *inside* can be re-labelled as *outside*. This is to properly handle the special cases such as p_{12} in Figure 3 or when an interior ring surrounds the exterior ring, as Figure 8 shows.

4.5 Time complexity

The time complexity of OddEven is defined by the complexity of constructing the CT, $O(v \log v)$, and by the reconstruction of the polygon(s), $O(vr \log r)$, with v the number of vertices in all the rings and r the number of rings. It should be noticed that constructing the CT can take $O(v^2)$ in the worst case since a quadratic number of edge-edge intersections are possible (eg in certain star polygons). However, for single polygons commonly found in GIS applications the number of intersections is usually much smaller than v. As an example, the 100 real-world GIS polygons used for the experiments in Section 6.2 contain no intersections. The other operations (ie closing rings and labelling) are performed in linear time or lower. Therefore, the total running time is $O(v \log v + vr \log r)$. If v is several orders of magnitude larger than r (as it is most always the case with polygons used in practice), the algorithm is dominated by the triangulation time, $O(v \log v)$.

For SetDiff, each ring is similarly repaired in $O(n \log n)$, where n is the number of points in a ring. Since every input vertex is only repaired once (vertices belonging to more than one ring appear in the input once per ring), this is equivalent to $O(\sum^r n \log n) \leq O(v \log v)$. The SetDiff repair process is therefore also performed in $O(v \log v)$.

This matches the experimental results in Section 6, where SetDiff is slower by a factor of about 2. This is explained by the fact that the OddEven steps have to be performed roughly twice: once for each ring, and once for all the rings together.

4.6 Example of repaired polygons

Figure 9 shows examples of invalid polygons that were repaired with the approach described in this section. The following should be noticed:

Dangling pieces These are ignored because the labels on the left and the right are the same.

Disconnected interior This is handled properly and one new polygon is created per interior-connected part.

Collapsed area These areas are simply ignored in the output (same labels on left and right). However, if they intersected another boundary, then the point(s) added during the construction of the CT is present in the output. It is possible to post-process segments and merge two consecutive collinear ones, but we have not implemented it.

Overlapping boundaries Such boundaries are merged / dissolved together.

Self-intersections Self-intersections, such as p_1 in Figure 3, are repaired as an interior boundary is constructed.

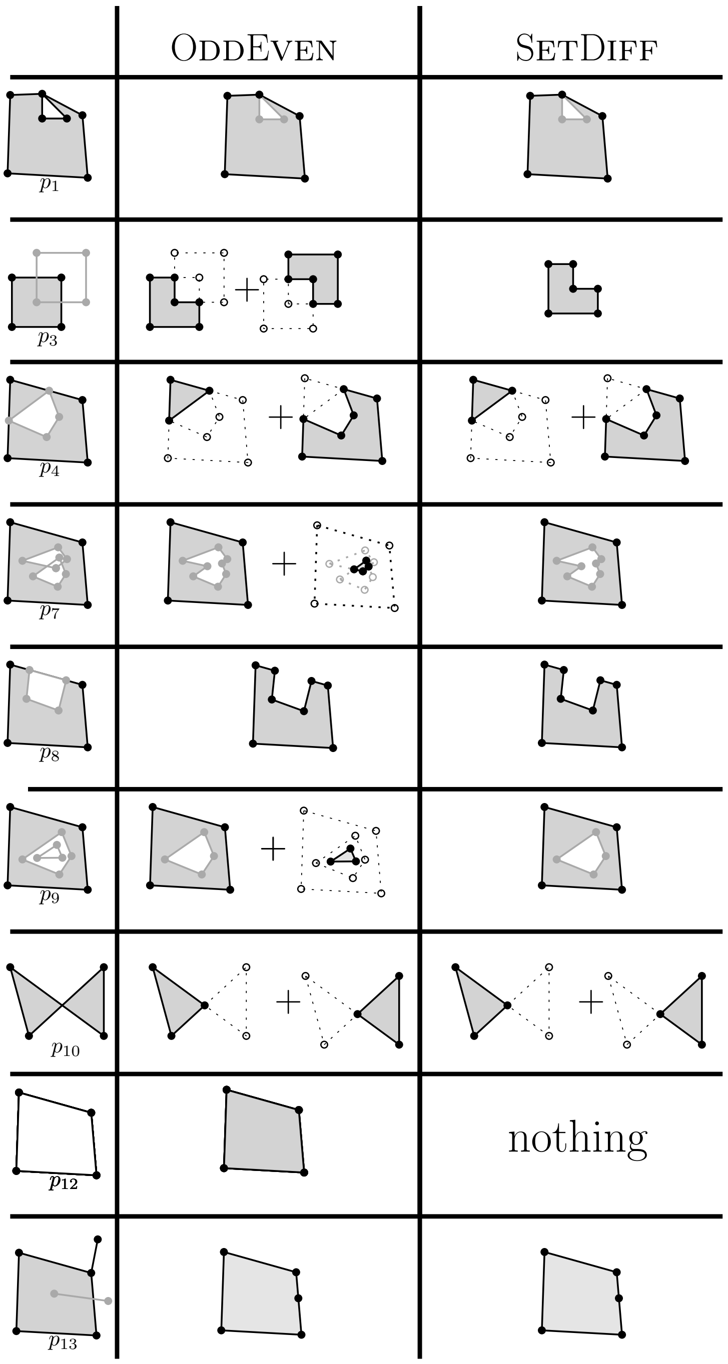


Figure 9: Some polygons from Figure 3 and how OddEven and SetDiff repair them; polygon p_{13} is new.

4.7 Repairing a MultiPolygon

A MultiPolygon is a collection of m polygons in which no two polygons overlap or are edge-adjacent (OGC, 2011). The first obvious approach to repairing a MultiPolygon is to use either OddEven or SetDiff separately for each polygon; this doesn't necessitate any changes to the algorithms. An second approach is to use one of the two algorithms with all the rings of the polygons together when labelling. Special cases such as when an interior ring of a polygon p_1 is located outside the exterior ring of p_1, but inside the exterior ring of another polygon p_2, will yield totally different outcomes. In our implementation of the two algorithms (presented in Section 6), we have favoured the latter. Our main motivation is that if one MultiPolygon was created by a user (and not several separate polygons), this is because the polygons "belong together" (and share the same set of attributes), and thus it makes more sense to repair them in an integrated manner.

5 Computing the robustness and improving it with snapping

The use of a CT as a base to automatically repair polygons can also help us to efficiently compute the robustness of a given polygon, before and after repair. Indeed, the triangulation itself serves as a spatial index, and no auxiliary spatial index structures (such as an R-tree) need to be used. The algorithm works by doing a breadth-first search (BFS) from every vertex v belonging to a polygon, visiting up to the closest vertex c or edge e also belonging to a polygon or until the robustness (initialised at infinity) has been reached. The robustness is updated to the smallest distance between v and c or e if it smaller than the current robustness. Notice that these vertices or edges are found as constraints in the input polygon, but after repair (labelling) they are instead defined as being incident to two triangles having different labels.

If the robustness of a given polygon is not high enough, it is possible to improve it by preprocessing the polygon with a well-known method to convert its segments from an arbitrary-precision representation to a finite-precision one: snap rounding (Goodrich et al., 1997). As shown in Figure 10, the method is based on the subdivision of the plane into a grid of a resolution s. Each vertex, and each intersection between two or more segments, are moved to the centre of the grid cell they are located in (these grid cells are labelled as 'hot'). While this ensures that the distance between two vertices is at least s, the distance between a vertex and a segment not incident to that vertex can be very small. Iterated snap rounding (ISR) solves that problem and ensures that any vertex is at least $s/2$ from a segment. As Figure 10c shows, it splits segments overlapping a hot cell by adding a new vertex at the centre of the cell; as a result, the segment is not straight anymore. The details of the algorithm are out-of-scope for this paper, the reader is referred to Halperin and Packer (2002).

While the ISR algorithm allows us to increase the robustness of a polygon, its topology can be significantly changed. For instance, the polygon in Figure 10 is now split into two polygons, but other cases such as the collapsing of a small area into a line can also arise. One example is the polygon in Figure 4d where the two spikes would potentially become segments, and thus be removed by our triangulation-based approach. ISR can thus be used not only to improve the robustness of polygons, but also to remove spikes since these collapses to lines, which are deleted.

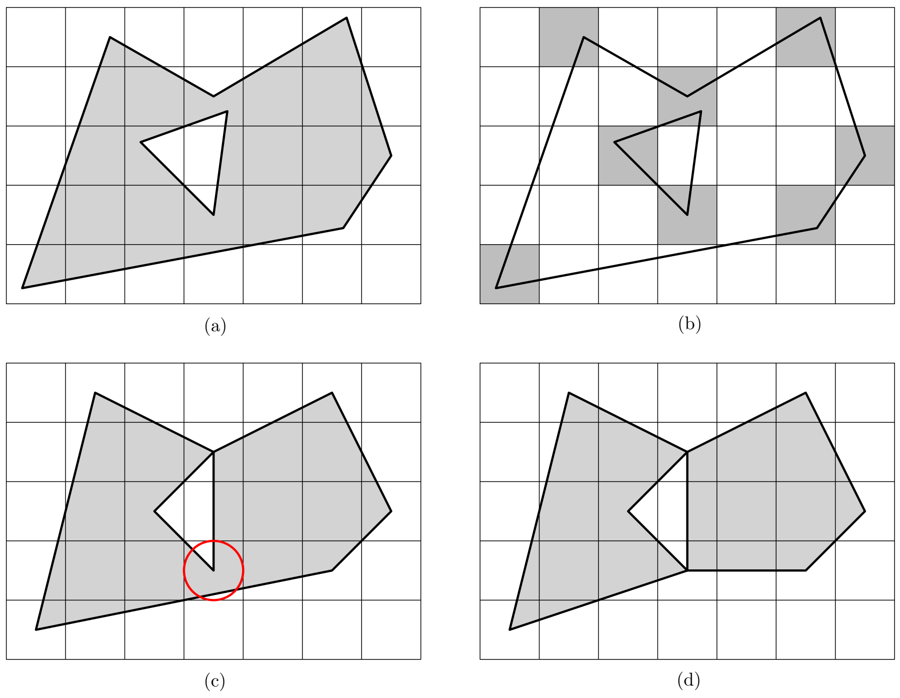


Figure 10: (a) Input polygon with one interior ring. (b) 'Hot' cells are in grey. (c) The polygon after vertices have been moved to the centre of the grid cells. Observe that one hot cell contains a line segment. (d) The result of the ISR algorithm.

Notice that the origin of the virtual grid used for ISR influences the outcome of Figure 4d: it is possible that the vertex and the segment be located in different grid cells and the spike would stay in the repaired polygon. However, the 'base' of the spike would be larger (the interior angle of the vertex at the right would be greater).

6 Experiments and comparison with other tools

We have implemented in C++ the two algorithms described in Section 4. The prototype, called `prepair`, is open-source and freely available under a GPL licence[5]. Two libraries are used: (1) CGAL[6] (we use its constrained triangulation module and its robust geometric operations) and (2) the OGR Simple Features Library[7] (which allows us to read and write from a large variety of GIS data formats). In its current form, the prototype reads one `Polygon` and returns one valid `MultiPolygon`; different GIS formats are supported for the input/output.

We describe in this section experiments that were run with different datasets. We compare our implementation of OddEven to that of PostGIS (version 2.0.2) and the function `ST_MakeValid`. Both implementations have the same behaviour, and preprocessing with ISR was not used. Since `ST_MakeValid` first validates a polygon (with the function *ST_IsValid()*), we have subtracted from the total running time the validation time. All the experiments were run on a laptop with Mac OS X 10.8, 2.5GHz and 4GB of main memory.

6.1 Unit tests

All the polygons shown in Figure 3, and other similar ones, were tested. The situations depicted in these purposefully involve many degenerate cases, both with regard to interpretation and implementation. They are meant as a sort of *unit testing* polygons to compare how they fare in different tools (Burns, 2001). We are able to repair all of these, with the behaviour explained in Section 4. `ST_MakeValid` repairs these correctly also, and the results are the same as OddEven, except that the collapsed geometries are also returned to the user. These polygons are very small and the running time is comparable.

6.2 Corine 2006 dataset

To test the efficiency of `prepair`, we have tested it with complex real-world polygons from the CORINE land cover dataset (CLC2006)[8]. Since they are constructed from reclassified raster imagery, they can be very large, both in terms of number of points and of rings. As a test dataset, we used the 100 largest (in terms of number of points) invalid polygons in the CLC2006 dataset. The

[5] `www.github.com/tudelft-gist/prepair`
[6] `www.cgal.org`
[7] `www.gdal.org/ogr`
[8] `www.eea.europa.eu/publications/COR0-landcover`

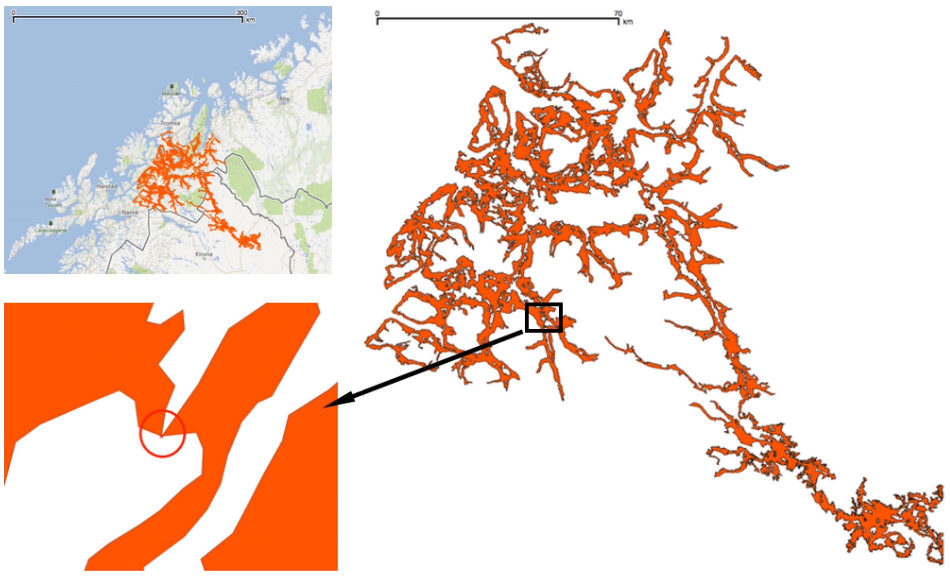


Figure 11: The polygon *EU-180927* from the CLC2006 dataset. It covers an area of about 26 000 km^2 in the North of Norway and Sweden, and contains 102 272 points and 299 rings. The polygon contains a typical error that is found in several polygons of the CLC2006 dataset: self-intersection of the exterior boundary.

smallest of these 100 polygons (ID = EU-2018418) contains 44 051 points and 126 rings; the largest (ID = EU-199949) contains 1 189 903 points and 7 672 rings. The average numbers of points and rings per polygon are respectively 146 478 and 776; the median values are 90 526 and 434. Figure 11 shows one example of a polygon, the errors in the polygons are generally the self-intersection of a boundary and different interior boundaries touching at more than one location. It should be noticed that with these 100 polygons, no new vertices were added during the construction of the CT; in other words there were no intersections between straight-line segments in the input. The number of vertices in the CT is in fact in each case *lower* than the number of points in the input (ranging from 2 to 90 vertices); the main explanation for this is that several points were duplicated, an example is shown in Figure 11.

Figure 12 shows the results of the experiments for the odd-even paradigm. Observe that for polygons with less than 400K points, the running time appears similar, although `prepair` is on average 6 times faster. The bottom plot in Figure 12 shows that `ST_MakeValid` follows roughly a polynomial of degree 2, and `prepair` has a linear behaviour. The results corroborates the theoretical analyses for the two algorithms, as previously explained in Section 3 and Section 4.5. For the polygons having between 500K and 1M points it is about 11 times faster. The exception is the biggest polygon: `ST_MakeValid` takes more than 100 times more time to repair.

The comparison of the running times for OddEven and SetDiff is shown in Figure 13. It can be seen that both algorithms, have a close-to-linear behaviour, and that in practice OddEven runs on average about twice as fast as SetDiff.

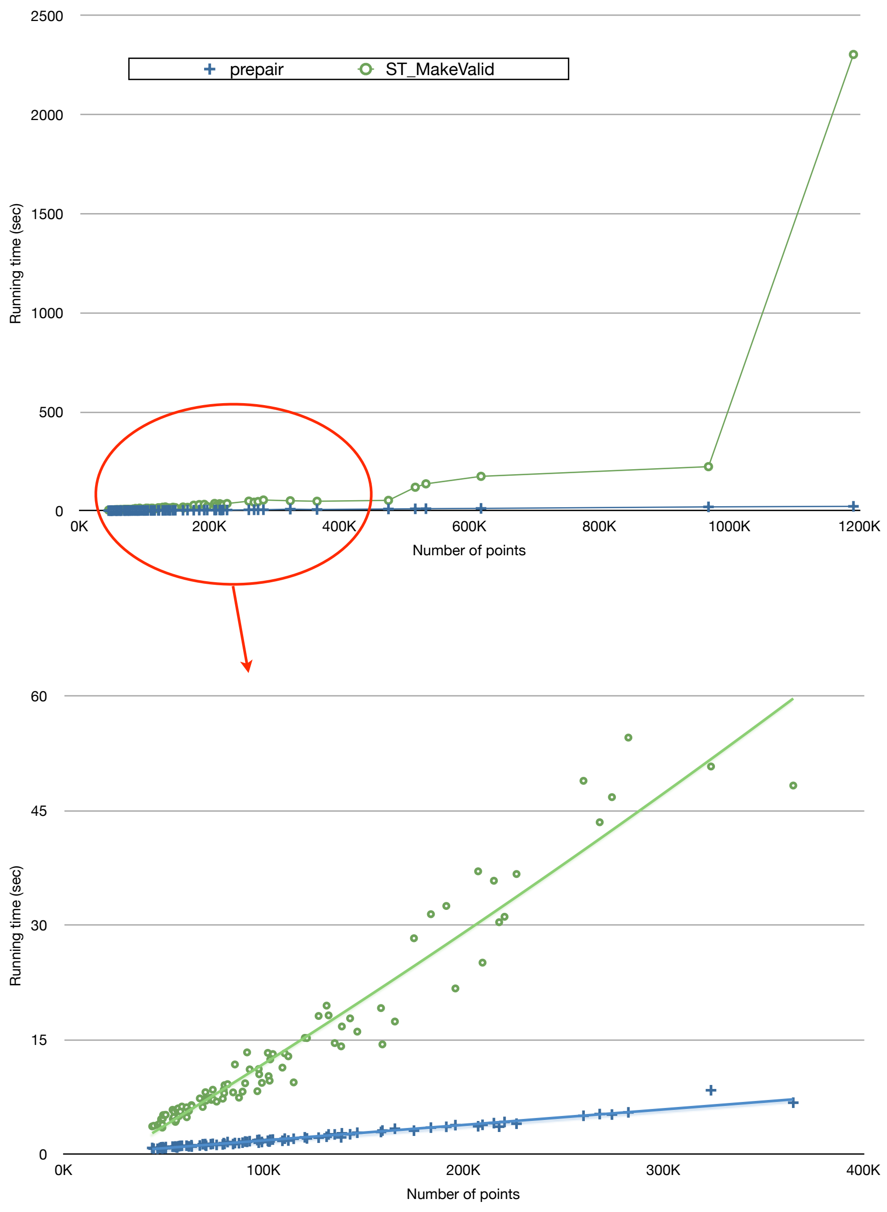


Figure 12: Running time of `prepair` (OddEven algorithm) and `ST_MakeValid` for the 100 largest polygons in the CLC2006 dataset. The bottom plot is for the part in the ellipse at the top.

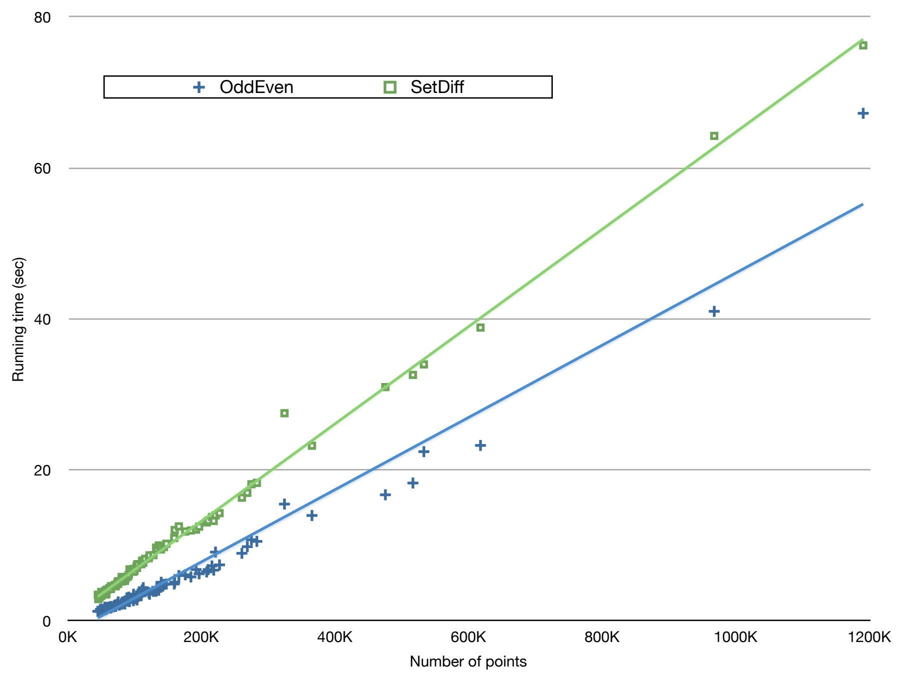


Figure 13: Running time for the CLC2006 polygons for the `prepair` implementations of OddEven and SetDiff. Both display a linear behaviour.

7 Conclusions

We have demonstrated that a triangulation-based approach for automatically repairing GIS polygons yields in practice a fast and scalable implementation (with a behaviour that in practice is linear in terms of the number of points in the polygon), and has several benefits over a graph-based approach. The main benefits are: (i) many of the cleaning operations can be performed locally on the CT; (ii) the graph of the polygons with interior rings is always connected, which facilitates the detection of topological relationships between rings; (iii) it is robust, thanks to the several robust triangulators that have been developed in several disciplines. We can further claim that our implementation is fully robust since we rely on CGAL (which uses exact arithmetic when needed) and our repair operations are expressed solely in terms of labelling of triangles (no complex geometric computations are involved).

While designing our approach we had to make several—often arbitrary—choices for its behaviour when special cases are present in the input polygon. While the way polygons are repaired in `prepair` is perhaps not always consistent with what one might do manually, we believe that the two paradigms we have proposed to automatically repair polygons are consistent and sufficient for most applications. The two repair paradigms can be described in a simple manner, and that permits users to predict easily how their polygons will be repaired. Since the two paradigms can be translated to properties of a CT and labelling of triangles, it is relatively easy for practitioners to modify the code so that different application-specific rules are used.

Acknowledgements

This research was financially supported by (i) the Dutch Technology Foundation STW, which is part of the Netherlands Organisation for Scientific Research (NWO), and which is partly funded by the Ministry of Economic Affairs (project code 11300); (ii) the European Location Framework (ELF) project, EC ICT PSP Grant Agreement no./ 325140.

References

Arroyo Ohori K, Ledoux H, and Meijers M (2012). Validation and automatic repair of planar partitions using a constrained triangulation. *Photogrammetrie, Fernerkundung, Geoinformation (PFG)*, 1(5):613–630.

Burns T (2001). Effective unit testing. *ACM Ubiquity*, 2001, issue January(Article no. 1).

CGAL (2011). *CGAL 3.8 User and Reference Manual.* CGAL Editorial Board.

Chazelle B (1982). A theorem on polygon cutting with applications. In *Proceedings 23rd Annual Symposium on Foundations of Computer Science*, pages 339–349. IEEE Computer Society, Washington, DC, USA.

Clarkson KL, Mehlhorn K, and Seidel R (1992). Four results on randomized incremental constructions. In Finkel A and Jantzen M, editors, *Proceedings of the 9th Annual Symposium on Theoretical Aspects of Computer Science*, volume 577 of *Lecture Notes in Computer Science*, pages 461–474. Springer Berlin / Heidelberg.

Facello MA (1995). Implementation of a randomized algorithm for Delaunay and regular triangulations in three dimensions. *Computer Aided Geometric Design*, 12:349–370.

Foley JD, Dam AV, Feiner SK, and Hughes JF (1996). *Computer Graphics: Principles and Practice.* Addison-Wesley.

Goodrich M, Guibas LJ, Hershberger J, and Tanenbaum P (1997). Snap rounding line segments efficiently in two and three dimensions. In *Proceedings 13th ACM International Symposium on Advances in GIS*, pages 284–293.

GTS (2006). *GTS Library Reference Manual.*

Guibas LJ and Stolfi J (1985). Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Transactions on Graphics*, 4(2):74–123.

Halperin D and Packer E (2002). Iterated snap rounding. *Computational Geometry: Theory and Applications*, 23(2):209–225.

Hoffmann CM (1989). The problems of accuracy and robustness in geometric computation. *Computer—IEEE Computer Society Press*, 22:31–42.

ISO(TC211) (2003). ISO 19107:2003: Geographic information—Spatial schema. International Organization for Standardization.

Jordan MC (1887). *Cours d'analyse de l'École Polytechnique, Paris*, volume Tome troisième. Gauthier-Villairs.

Liu Y and Snoeyink J (2005). The "far away point" for Delaunay diagram computation in $\mathbb{E}^d$. In *Proceedings 2nd International Symposium on Voronoi Diagrams in Science and Engineering*, pages 236–243. Seoul, Korea.

Mücke EP, Saias I, and Zhu B (1999). Fast randomized point location without preprocessing in two- and three-dimensional Delaunay triangulations. *Computational Geometry—Theory and Applications*, 12:63–83.

OGC (2011). OpenGIS implementation specification for geographic information—simple feature access. Open Geospatial Consortium inc. Document 06-103r4.

Ramsey P (2010). PostGIS: Tips for power users. Presentation at the FOSS4G 2010 Conference, Barcelona, Spain. `http://2010.foss4g.org/presentations/3369.pdf`.

Shewchuk JR (1997). *Delaunay Refinement Mesh Generation*. Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburg, USA.

van Oosterom P, Quak W, and Tijssen T (2004). About invalid, valid and clean polygons. In Fisher PF, editor, *Developments in Spatial Data Handling—11th International Symposium on Spatial Data Handling*, pages 1–16. Springer.