EDGECRACK: a parallel Divide-and-Conquer algorithm for building a topological data structure

Martijn Meijers & Hugo Ledoux
Delft University of Technology (OTB, section GIS technology), The Netherlands

ABSTRACT: In this paper we consider the problem of converting a large database of 2D polygons into a topological data structure (a data structure with nodes, edges and faces). We present EDGECRACK, an algorithm to obtain the topological data structure (which is based on a known algorithm for segment intersection) and performs small geometric corrections of the input by snapping to avoid problems. We further show how we have extended this algorithm to a Divide-and-Conquer approach, which is also suited for parallel processing. We present experimental results based on our implementation and show that we have been able to convert a large database of 5.3 millions polygons into a topological data structure.

1 INTRODUCTION

This paper presents our ongoing investigations to convert a large database of 2D polygons, modelled according to the Simple Features specification (OGC, 2006), into a topological data structure (a data structure with nodes, edges and faces and explicit adjacency links between them). Although this important operation has been investigated in the past (van Roessel, 1991; van Oosterom, 1994), it is still problematic to perform with standard GIS tools an initial conversion to such topological data structures for larger data sets (i.e. performing the initial bulkload conversion in a reasonable amount of time, cf. Arroyo Ohori et al., 2012). Furthermore, snapping of geometry to solve small geometric errors in the input, e.g. caused by truncating coordinate digits to an exchange format, is a prerequisite for us.

We have investigated and extended a known algorithm for segment intersection (Sugihara, 1991, 1992) and here we first present our extended algorithm, which we call EDGECRACK. EDGECRACK performs the geometry to topology conversion, but can at the same time also ensure that coordinates are rounded to a grid and that features are snapped to each other (based on various tolerances, e.g. segment-segment and segment-point distance thresholds), while maintaining a valid topology structure and transferring the thematic attributes of the original polygons to the topology structure. This is, however, at the cost of added points.

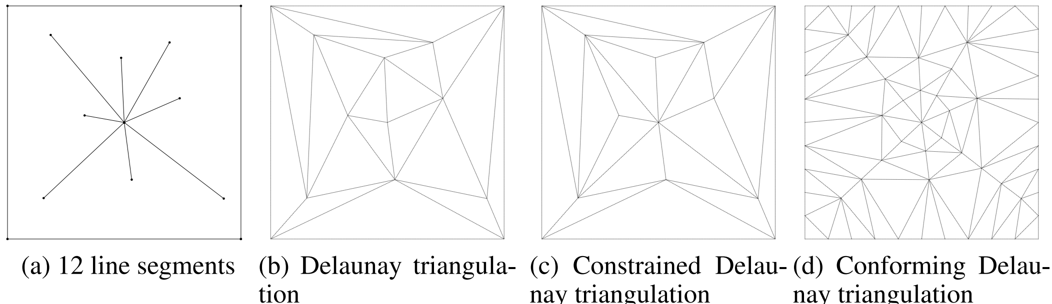
More importantly, based on EDGECRACK, we have devised a divide-and-conquer algorithm so that we are able to obtain *in parallel* an explicit topological data structure for the input, so that we can process datasets with millions of input polygons. We show details of the implementation and the tests that we conducted with EDGECRACK on a subset of the polygons of a large topographic dataset. We have been able to process 5.3 million polygons into an explicit topology data structure with our parallel, Python based implementation.

The remainder of this paper is organized as follows. Section 2 gives relevant definitions. Section 3 introduces EDGECRACK and explains the Divide-and-Conquer algorithm. Section 4 presents the main results that we have obtained, while running the algorithm on a large input data set. Section 5 offers concluding remarks and provides some suggestions for future work.

2 DEFINITIONS

Our topology builder is based on using a Conforming Delaunay Triangulation, this section gives some relevant definitions of this and related concepts.

A triangulation is a Delaunay Triangulation (DT) if the circumscribing circle of any triangle of the triangulation contains no vertex in its interior. A triangulation edge (a side of a triangle) is said to be Delaunay if it is inscribed in an empty circle (containing no vertex in its interior).



(a) 12 line segments (b) Delaunay triangulation (c) Constrained Delaunay triangulation (d) Conforming Delaunay triangulation

Figure 1: Different ways to embed a set of segments in a triangulation: (a) shows the line segments (b) gives the Delaunay triangulation of the end points of the segments, note that not all segments are present (c) illustrates the Constrained Delaunay triangulation, all line segments are represented, but not all triangles are fully Delaunay (d) shows a Conforming Delaunay triangulation, all line segments are represented and all triangles are Delaunay.

Figure 1 illustrates that a set of given line segments may initially be present in a DT – or only partly. To be sure that all segments are represented in the triangulation two approaches exist: the first is the Constrained Delaunay approach that loosens the Delaunay criterion by introducing a visibility criterion, which then means that a circumscribing circle of any triangle of the DT will contain no vertex 'visible' from this triangle. In this case, a constraint edge acts as a visibility blocker and this way input edges ('constraints') are embedded in the triangulation. A second approach to represent segments in a DT is by adding vertices to the initial segments, until all are recovered and the triangulation *conforms* to the given segments, hence the name: Conforming Delaunay triangulation. An advantage is that each triangle is conforming to the Delaunay criterion, without visibility criterion. The difficulty with this approach is that to recover one segment it is likely that another (earlier already represented) segment will be destroyed and in worst case quite many vertices (order $O(n^2)$) might have to be added.

Theoretical algorithms, and several implementations, that robustly construct a Constrained Delaunay triangulation, assume that the input segments form a Planar Straight Line Graph (PSLG). A PSLG is a set of vertices and straight line segments that satisfies two conditions: 1. the PSLG must contain for every segment the two vertices that are its endpoints, 2. segments are only allowed to intersect at their endpoints. In our case, this is a too strong requirement for our input, as topological structuring exactly has to produce these new intersection points for touching and (partially) overlapping polygon boundaries (these can not exist by definition for a PSLG) and at the same time we like to snap segments together that are very close. Therefore, the line intersection algorithm of Sugihara (1992) that we employ in EDGECRACK is based on the second approach: constructing a Conforming Delaunay triangulation.

3 OUR APPROACH

This section presents in § 3.1 how we extend Sugihara's segment intersection algorithm to a topology builder. We give details on our Divide-and-Conquer algorithm for building topology for large data sets with polygons in § 3.2.

3.1 EDGECRACK

We start with input polygons, that are valid according to the Simple Feature specification definition (OGC, 2006). If this would not have been the case, we perform a cleaning step, in which we make all input polygons individually valid (Ledoux et al., 2012). A polygonal area is bounded by one (or more) ring(s): every polygon consists of at least one outer ring and zero or more inner rings. A ring is a simple, closed polyline. Next to the

validity according to the specification (where rings are allowed to touch each other in at most one point), we assume that each ring is oriented in such a way that the interior of the polygon is at the left hand side of this ring, which means that the outer ring is oriented counter clockwise and all inner rings are oriented clockwise. Every segment of the ring is formed by two consecutive vertices. Figure 2 illustrates that based on the orientation rules we can annotate the left side of all segments with the identifier of the input polygon (at the left).

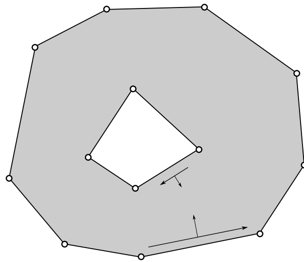


Figure 2: Based on known orientation (outer ring counterclockwise, inner ring clockwise), we can annotate the left side of every segment with the identifier of the input polygon.

EDGECRACK is for a large part based on the segment intersection algorithm given by Sugihara (1992). The segments of all input polygons form the input. EDGECRACK starts by first realizing (cracking) the segments of the polygon boundaries (that become edges) into the conforming Delaunay triangulation. Realizing the segments into the triangulation also takes care of snapping. After realizing all the input segments in the triangulation the structure conforms to the input, while for all triangles the Delaunay criterion still holds. After all segments have been realized in the triangulation, the nodes, edges and faces can be obtained from the triangulation, and because segments were annotated with the identifier of the original polygon, also the connection to the original polygon attributes is maintained.

A segment for EDGECRACK is in our algorithm part of a hierarchical data structure (a binary tree) to keep track of additional insertions of points to represent the segment in the conforming triangulation. Figure 3 gives an illustration of how segments are represented in the triangulation. Based on the input polygons a list of segments is created. The place of a segment in the list is what we call the 'segment handle'. We represent the segment in the triangulation by keeping a list of segment handles on every triangulation vertex.

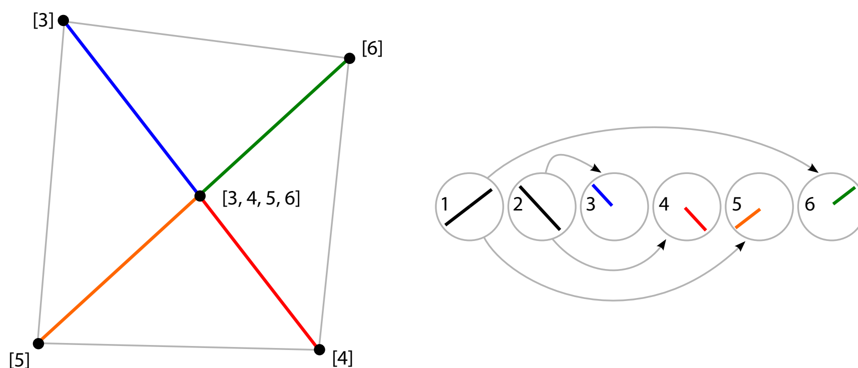


Figure 3: Segments are represented in the triangulation by their place in the segment list, the 'segment handle'. The original segments 1 and 2 have been split by insertion of their intersection point. This vertex in the triangulation representing the intersection point has a list of all related segment handles (3, 4, 5 and 6). Segments are part of a binary tree, which tracks how segments are split (illustrated by the circles and arrows). Note that segment 1 and 2 (and their respective handles) are not present any more in the triangulation at this stage.

Following the terminology of Sugihara, we call a segment *realized* if it is present as a side of a triangle in the triangulation. The original segments will be split to be represented in the triangulation. Splitting thus leads to subsegments, which are descendants of the original segment. If the length of a subsegment in the segment tree goes under a defined threshold for a minimal segment length, we call the segment *saturated* – this also means that we are not allowed to split this segment any further.

EDGECRACK proceeds then as follows, while maintaining a valid Delaunay Triangulation throughout:

Step 1. Compute the Delaunay Triangulation based on the end points of all the initial segments. In our implementation we use Lawson's incremental insertion algorithm (Lawson, 1977) and we represent the triangulation with a half edge data structure (Mäntylä, 1988).

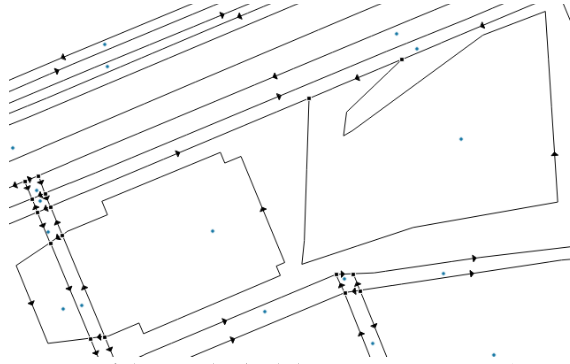


Figure 4: Visualization of the resulting contents of the topological data structure. Note that EDGECRACK also generates a point inside every face (that can be used for example as anchor point for placing a label).

Insertion of a point is based on walking in the triangulation to its definite location. If in this process, based on some tolerance value, a close enough point is detected that already exists in the triangulation, the newly inserted point is snapped to the existing point. A segment is ignored if its two end points collapse to one vertex. This also means that the order of insertion of vertices in the triangulation is important, e.g. polygons that represent 'hard' topography with higher quality, such as houses, should be inserted first.

Step 2. Crack all segments into the triangulation. Cracking means adding additional points to the input segments, until all segments either are realized or saturated.

The loop that adds points to the initial triangulation consists of two steps. Firstly, it tries to add an intersection point between an already realized segment and a not yet unrealized segment (Sugihara terms this: "finding a cross pattern"). Secondly, if segments are knocked out by adding the new intersection point, the longest of these segments is recovered (either by snapping the segment to already existing vertices, or by adding its midpoint).

Snapping a line segment to one or more close points is possible by performing a walk in the triangulation to find close vertices to the not yet represented segment. If there are close vertices (close is defined by a threshold), the original segment is 'bend', so that it is defined by these vertices. This happens frequently in polygonal domains with neighbouring objects (think of two differently sized squares that share the same edge, the longest segment should be bend so that it goes through the shortest segment). This snapping is a modification from the original Sugihara's algorithm.

For both steps the segments are split into subsegments. Splitting means that in general 2 new segments are added to the hierarchical segment tree. For these new segments we register their handles in the triangulation and the old handles of the parent segments are removed from the triangulation.

Step 3. Force unrealized segments.

If unrealized – but saturated – segments do exist after step 2 (i.e. the input segment is still not represented as a collection of triangle sides in the triangulation), we redefine these segments based on a shortest path traversal over the triangulation, i.e. we execute Dijkstra's shortest path algorithm on the relevant subpart of the triangulation (Dijkstra, 1959).

In our experiments we have observed that this step is barely used, but still necessary to ensure a correct result.

Step 4. Generate the topology structure (with nodes, edges and faces).

In this step, we embed the knowledge of which triangle side is a segment directly in the triangulation data structure. It means keeping a boolean flag on every triangulation edge of the triangulation. This simplifies traversal of the triangulation data structure, to 'harvest' nodes, edges and faces from the triangulation. Because all segments have been embedded in the triangulation structure harvesting means traversing every element class of the triangulation once (i.e. triangle vertices, triangle edges and triangles) to obtain nodes, edges and faces. Topology primitives are harvested as follows:

- Nodes are the vertices where there are more than 2 original input segments incident. Additionally, for a ring of segments that forms a hole, one arbitrary vertex has to be selected as node (cf. the hole of polygon in Figure 2).

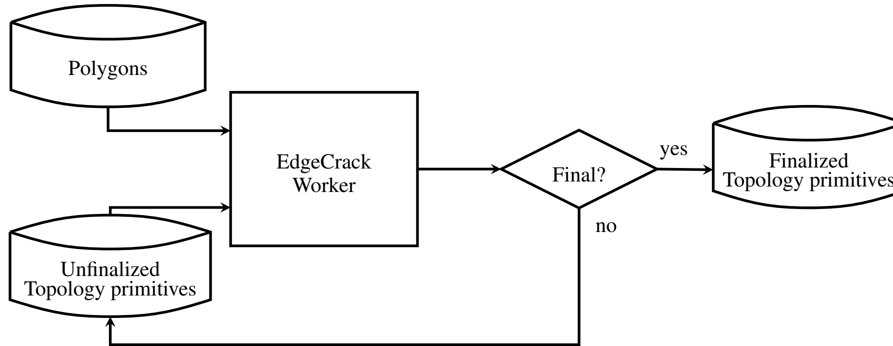


Figure 5: Flow of data for our Divide-and-Conquer algorithm.

- Edges are formed by making the longest chains possible between triangle vertices that are nodes.
- Faces are created by forming groups of triangles (a connected component) that are surrounded by a set of segments. Per face also a point inside the face is generated – the barycenter of the largest triangle of the group.

Figure 4 gives an example of the resulting nodes, edges and faces.

3.2 *A Divide-and-Conquer algorithm suited for parallel processing*

EDGECRACK functions well if the number of vertices of all input polygons and the new points that will be created, together with the triangulation data structure, fit in main memory. When this is not the case, the operating system will have to start swapping and this causes performance problems (very long run times). A solution to this is to divide the input into smaller and more manageable chunks.

For chunking a dataset in smaller pieces two possibilities exist: 1. cut the input geometries into pieces by introducing cut lines into the input – this leads to extra intersection points and segments, which need to be administrated, so that they can be removed from the definite output, 2. process whole input geometries – with this option it is necessary to keep an administration (which polygons are being processed) as processing the same polygon at the exact same time could lead to contradictory results that then have to be integrated again (i.e. locking is necessary). Although option 1 could initally lead to a more parallel solution (as no locking is needed for obtaining the topology primitives) embedding and then later removing the cut lines is unattractive. This option would mean that the topological identifiers, that were handed out by the parallel processes, should be unified in a separate step (i.e. to unify faces and edges that were split by the cut line and to remove the unwanted nodes on the cut lines). As snapping will happen at both sides of the cut line, and this unification step will have to use the geometry of the resulting topology primitives, it might be quite problematic. Therefore, we have opted for the second alternative.

To be able to chunk the input dataset into smaller pieces, we construct a quadtree based on the input polygons. A quad in the quadtree is split in 4 if too many vertices would have to be processed in one go. Once the quadtree has been constructed, all leaf nodes of the quadtree form a full partitioning of the input dataset, which we will call *tiles*. Figure 6 gives an example of a set of tiles that we have used in our experiments.

After constructing the tile set, we process the polygonal input data per tile, with multiple tiles in parallel. One instance of EDGECRACK (which we call a worker) selects for one of the tiles all related input polygons, i.e. every polygon whose bounding box overlaps the tile. Based on the the union of all the bounding boxes the worker locks all neighbouring tiles that are overlapping this unioned bounding box. Another worker can be run in parallel, except that it is not allowed to process any locked tiles. A worker produces a set of topology primitives and when a worker is finished, it unlocks the tiles it did lock and finally marks the tile as processed.

The main algorithm, as described in § 3.1, has to be extended to be able to function correctly in parallel with other workers. Next to the polygonal input geometries, a worker also is given the outline of which tiles have been processed already, so that it can be determined by a worker when a topology primitive that is generated is final (*finalized*) or has to be further processed (*unfinalized*). Finalized elements are those topology primitives for which it is clear that they will not need to be processed in any other of the remaining tiles. A worker also produces unfinalized elements: these topology primitives will have to be loaded when an adjacent (and previously locked) tile is processed. Although we present not all special cases here (e.g. adjacency to the universe

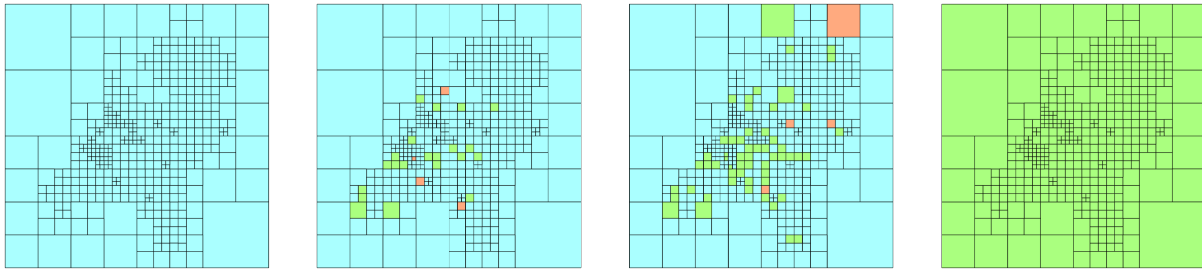


Figure 6: A run of EdgeCrack with 4 parallel workers. Each tile contains no more than 500 000 vertices. Colours represent the status of a tile: *blue*: not yet processed, *orange*: being processed by a worker, *green*: processed.

face), determining whether a primitive is finalized or not can be realized by comparing a primitive its related bounding boxes to the union of the already processed tiles. Which bounding boxes to consider depends on the type of topology primitive:

- A node is finalized if the node itself (and hence its 'degenerate' bounding box) is located in the interior of the union of the processed tiles (i.e. it is fully inside and not on the boundary).
- A face is finalized if its bounding box is fully inside the union of the processed tiles.
- An edge is finalized if its bounding box *and* the two bounding boxes of its adjacent faces are fully inside of the union of the processed tiles.

Figure 5 illustrates that a worker, that processes an earlier locked tile, will first load the unfinalized topology primitives and that it will not load the original input polygons which generated these topology primitives. By using the outline of the processed tiles, it can be determined which polygons were already processed into (unfinalized) topological primitives – their bounding box overlaps with the outline of the processed tiles.

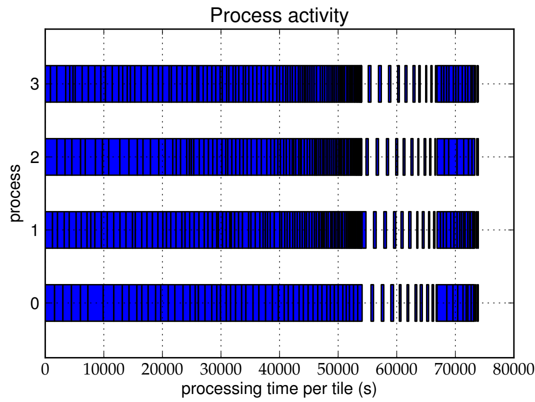
Note that for the unfinalized topology primitives the segments need to carry information that is relevant for the topological data structure – similar to the identifier of the polygon we keep information (pointers, e.g. from edge to face) that is important for the topology structure. The reason for this is that an edge that is unfinalized can point to a face that is finalized already and for which a topological identifier was handed out, so this information should be propagated from a processed tile to a not yet processed tile.

An important remaining question is the order in which to process the tiles. Determining an exact schedule beforehand seems not viable: although overlaps of tiles are known beforehand, it is not known a priori how long a task will run. For the decision of which tiles to process first, we therefore turn to reasonable simple heuristics to be able to derive dynamically at run time an execution schedule. We aim at processing first the tiles that both have a lot of overlap with other tiles and contain a lot of input segments. This way we try to prevent a scenario where we can not process tiles in parallel. In our experiments, we have implemented these heuristics as follows: Per tile we query the database to get a unioned bounding box of all input polygons that spatially interact with the tile (their bounding box overlaps the tile) and get the size of the area that is covered by this box. Furthermore, we count the total number of vertices in the related input polygons as this gives a good estimate how many segments will need to be processed. By multiplying the two numbers per tile we can get an ordered list of tiles, where tiles that should be processed first have a high associated number.

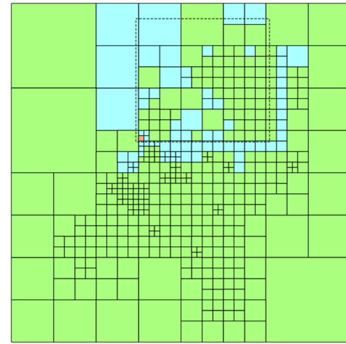
Our experiments (§ 4) show that we did not fully succeed with running in parallel all the time with this approach – congestion takes place due to the geometric configuration of the input. However, the heuristics seem to be good enough for practical use. In the worst case EdgeCrack would only be able to run on all tiles sequentially and no parallel speed up can then be obtained, which seems very unlikely for practical GIS data sets.

4 EXPERIMENTAL RESULTS

We have implemented EdgeCrack and the parallel Divide-and-Conquer algorithm. For the implementation we used the Python programming language and we have used Cython for optimizing some important parts of the implementation to shorten the runtime (e.g. the triangulation). For the triangulation we have created a wrapper around the exact predicates of Shewchuk (1997) for orientation and in-circle tests, to be able to robustly build the triangulation (we have observed errors when not using the exact predicates, while these did not occur when the exact predicates were in use).



(a) Activity of workers



(b) One large polygon makes that all remaining tiles are locked and only one worker can process data. The dashed line shows the unioned bounding box of all polygons that are related to the orange (currently active) tile.

Figure 7: Activity of workers of a run of EDGECRACK with 4 parallel workers. The two subfigures illustrate that congestion takes place: only one worker at a time can process data (after ±55k seconds) due to the geometric configuration of the input. After the related tiles totally have been processed, the workers can again process data in parallel (after ±68k seconds).

Table 1: Statistics on input – geometry – and output – the generated topology. Note that: a. the number of faces is less than the number of input polygons, because slivers were cleaned by the performed snapping and b. the number of coordinates in the resulting topology is quite high due to the use of EDGECRACK).

	Geometry		Topology			
	Polygons	Coordinates	Nodes	Edges	Faces	Coordinates
Top10NL	5 326 132	99 212 628	7 887 686	13 040 554	5 298 197	72 953 951

With the implementation, we have performed two parallel test runs. In the first run, we set the maximum of vertices for a tile to 500 000; in the second run the maximum was set to 1 000 000. The tests were run on our research server (a SUN V40z server running Solaris 10, 64-bit) equipped with 4 AMD CPU's clocked at 2.2 GHz and 8 GB of main memory.

The input geometry – a subset of the polygons of the Dutch Top10NL topographic dataset (which since January 1, 2012 is available to the general public as open data, see `http://www.kadaster.nl/top10nl/`) – is stored in PostgreSQL 9.1.4, extended with PostGIS 2.0.1. These polygons are supposed to form a polygonal coverage (representing terrain, water and road features).

Figure 7 shows details on the activity of the workers for the parallel process during the first test run. It shows that the total run took about 21 hours (the second run, with more data per tile did not significantly deviate from this). Furthermore, it illustrates that, although we try to avoid it by our heuristics of first processing tiles with a huge number of inputs and having the biggest overlap with other tiles, congestion takes place: only one worker can process data (after ±55k seconds) due to the geometric configuration of the input. In the input there exists one large polygon, which causes that all remaining tiles are locked by one worker and only this worker can process data. However, after the related tiles have been fully processed, the workers again start to process data in parallel (after ±68k seconds).

Table 1 shows the statistics about the number of generated topological primitives and coordinates. It is clear that EDGECRACK could be leaner with respect to added coordinates: if a boundary is stored twice by the polygonal geometry, we would expect to find around 50% of the original input vertices in the topological data structure. This would mean that 99M coordinates should lead to approximately 50M coordinates, however we obtain around 73M coordinates. This higher number is caused by the coordinates that were added by midpoint insertions (to represent the segments in the triangulation).

For both test runs we have collected runtime statistics. Figure 8 shows the runtime versus the size of the input: our tests on the topographic dataset indicate that our approach with this dataset exhibits $O(n \log n)$ runtime performance, which is what theory states (Lawson, 1977).

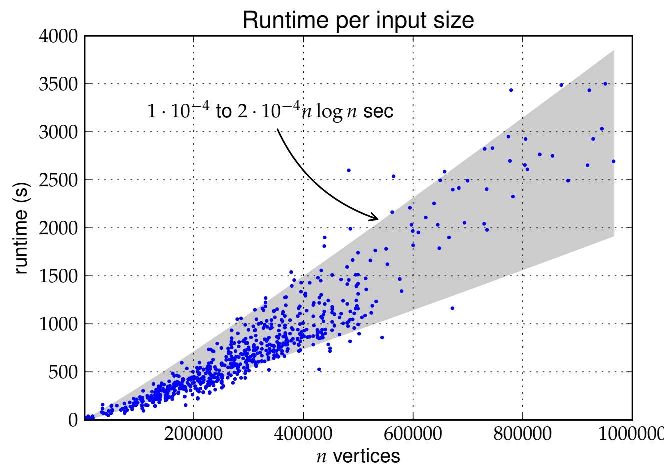


Figure 8: Runtime of EDGECRACK for varying input sizes using the topographic Top10NL dataset. Practical runtime for the tested dataset exhibits $O(n \log n)$ behaviour.

5 DISCUSSION

5.1 *Discussion of results*

The most important contribution of this work is that we have presented a Divide-and-Conquer approach for building a topological data structure for huge polygonal data sets (with millions of polygons), that is suited for running in parallel. Furthermore, we have demonstrated with EDGECRACK that snapping can be easily integrated in Sugihara's algorithm of segment intersection. This snapping is based on using the conforming Delaunay Triangulation as underlying spatial search structure for finding close vertices.

The beauty of the parallel algorithm is that it is in principle not tied to EDGECRACK. Also other approaches could be used for constructing the topological data structure, as long as a worker correctly splits the output in finalized and unfinalized primitives. For this, the same principles for scaling out – using the locking strategy and the comparision of the bounding box of generated primitives with the already processed area – can be applied.

5.2 *Future work*

One concern is that EDGECRACK adds quite some vertices, also on locations where these do not exist in the input (the crossings initially also do not exist, but these should be made explicit as a requirement from the topological data structure). The addition of points has a negative influence on practical runtime (although for our experiment the algorithm seems to follow an $O(n \log n)$ theoretical curve).

One option is to remove the added extra points (which was already suggested by Sugihara, 1992). Although the points are annotated with the fact that they are added to the triangulation as additional 'midpoints', we have not yet tried this. The reason for this is that removal of these points has to be executed before outputting the topology structure. We think that this can be accomplished by collapsing triangulation edges that are between two of such points (these points are by definition on a straight line) and updating the triangulation accordingly, but we still have to try this.

Another option is to employ a completely different strategy: either using a Constrained Triangulation (where only intersection points are added, that still will have to be computed for intersecting segments), or using a completely different approach (e.g. using a R-tree or sweepline, cf. van Oosterom, 1994; van Roessel, 1991; Hobby, 1999). For selecting an alternative approach we should carefully consider how snapping would fit.

Furthermore, we plan to extend EDGECRACK, so that it also supports point and line object and not only polygons as input. The result then is a scalable bulk loader for topology that can be stored in either Oracle Spatial Topology or PostGIS (which since version 2 supports a topology model based on ISO, 2006, see `http://postgis.refractions.net/docs/Topology.html`).

ACKNOWLEDGEMENTS

This research is supported by the Dutch Technology Foundation STW (project numbers 11300 and 11185), which is part of the Netherlands Organisation for Scientific Research (NWO) and partly funded by the Ministry of Economic Affairs.

REFERENCES

Arroyo Ohori, K., H. Ledoux, and M. Meijers (2012, October). Validation and Automatic Repair of Planar Partitions Using a Constrained Triangulation. *Journal of Photogrammetry, Remote Sensing and Geoinformation Processing 2012*(5), 613–630.

Dijkstra, E. W. (1959). A note on two problems in connection with graphs. *Numerische Mathematik 1*, 269–271.

Hobby, J. D. (1999, October). Practical segment intersection with finite precision output. *Computational Geometry: Theory and Applications 13*(4), 199–214.

ISO (2006). ISO/IEC 13249-3: 2006: Information technology - Database languages - SQL Multimedia and Application Packages - Part 3: Spatial.

Lawson, C. L. (1977, August). Software for C^1 surface interpolation. In J. R. Rice (Ed.), *Mathematical Software III*, New York, pp. 161–194. Academic Press.

Ledoux, H., K. Arroyo Ohori, and M. Meijers (2012). Automatically repairing invalid polygons with a constrained triangulation. In *Proceedings 15th AGILE International Conference on Geographic Information Science*.

Mäntylä, M. (1988). *An introduction to solid modeling*. New York, USA: Computer Science Press.

OGC (2006, October). OpenGIS Implementation Specification for Geographic Information - Simple Feature Access - Part 1: Common Architecture.

Shewchuk, J. R. (1997, October). Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates. *Discrete & Computational Geometry 18*(3), 305–363.

Sugihara, K. (1991, February). A Robust Intersection Algorithm Based on Delaunay Triangulation. Technical Report CSD-TR-91-011, Purdue University.

Sugihara, K. (1992). Application of the Delaunay triangulation to geometric intersection problems. In L. Davisson, A. MacFarlane, H. Kwakernaak, J. Massey, Y. Tsypkin, A. Viterbi, and P. Kall (Eds.), *System Modelling and Optimization*, Lecture Notes in Control and Information Sciences, pp. 112–121. Springer Berlin / Heidelberg.

van Oosterom, P. (1994). An R-tree based map-overlay algorithm. In *Proceedings EGIS'94*, pp. 318–327.

van Roessel, J. W. (1991). A new approach to plane-sweep overlay: Topological structuring and line-segment classification. *Cartography and Geographic Information Science 18*, 49–67.