Using Rational Numbers and Parallel Computing to Efficiently Avoid Round-off Errors on Map Simplification

Maurício G. Gruppi [1], **Salles V. G. de Magalhães** [1,2], **Marcus V. A. Andrade** [1],
W. Randolph Franklin[2], **Wenli Li**[2]

[1]Departamento de Informática – Universidade Federal de Viçosa (UFV)
Viçosa – MG – Brazil

[2]Rensselaer Polytechnic Institute
Troy – NY – USA

`{mauricio.gruppi,salles,marcus}@ufv.br, mail@wrfranklin.org,`
`liw9@rpi.edu`

Abstract. *This paper presents* EPLSimp, *an algorithm for map generalization that avoids the creation of topological inconsistencies.* EPLSimp *is based on Visvalingam-Whyatt's (VW) algorithm on which least "important" points are removed first. Unlike VW's algorithm, when a point is deleted a verification is performed in order to check if this deletion would create topological inconsistencies. This was done by using arbitrary precision rational numbers to completely avoid errors caused by floating-point arithmetic.* EPLSimp *was carefully implemented to be efficient, although using rational numbers adds an overhead to the computation. This efficiency was achieved by using a uniform grid for indexing the geometric data and parallel computing to speedup the process.*

1. Introduction

The map simplification process, also known as map generalization, allows the production of maps with different levels of details [Jiang et al. 2013]. It consists of removing information that is not relevant to the viewer, while preserving essential features on the map. Generalization is inherent to every geographical data since every map consists of generalized representations of reality, and the more generalized a map is, the more distant it becomes from the real world [João 1998]. The output of this process is a map with more desirable properties than those from the input map. An example of generalization is scaling a map of a single town which contains detailed information about streets and buildings. When scaling this map to show nearby towns it may be necessary to simplify it so that it is not overburden by unimportant data.

A challenge in generalization is to find a balance between simplification and reality. Map simplification can produce inappropriate results as it may affect topological relationships. These results are said to be *topologically inconsistent* and they may present relationships that are conflicting with reality. For example, the simplification can create self-intersecting lines, improper intersections between lines and polygons, etc.

Another kind of topological inconsistency is the sidedness change, that is, after performing simplification, a feature can be on a different side regarding other feature on the map. For example, after the simplification of a line, a point which was originally on the right side of this line now can be on the left side. Thus when designing simplification algorithms it is important to guarantee topologically consistent results.
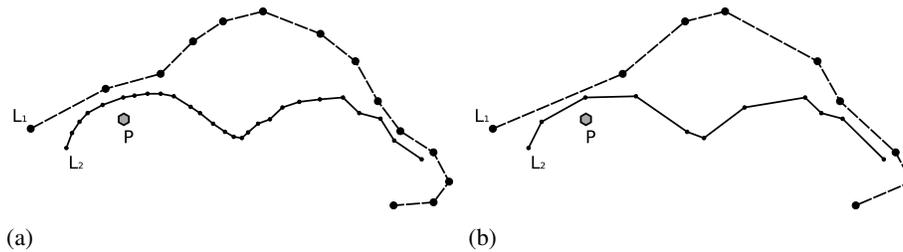
162

Figure 1. (a) Example of polylines L_1 and L_2 and a control point P. (b) Simplification of L_1 and L_2. Notice that topology consistency is preserved: no intersections were created and sidedness is maintained.
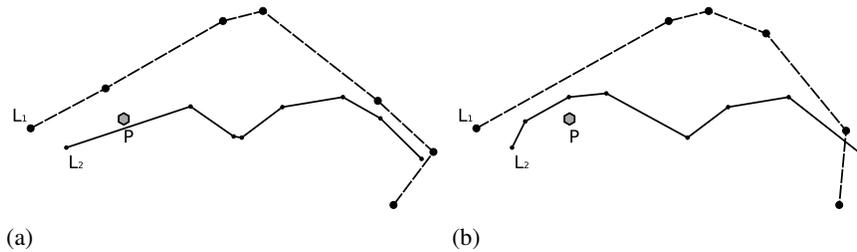

Figure 2. Inconsistent simplification output: (a) P is on the wrong side of line L_2; (b) nonexisting ntersection between lines L_1 and L_2 is created.

2. Polyline Simplification

An approach for performing map simplification is to reduce the complexity of its lines. That means making simpler representation of curves or polygon edges. Usually, lines are represented by polygonal chains or polylines. A polyline is a serie of segments defined by a sequence of n vertices $(v_1, v_2, ..., v_n)$, where each segment consists of two endpoints and adjacent segments share a common endpoint. Figure 1(a) shows an example of two polygonal chains L_1 and L_2, and also a control point P (gray hexagon) that does not belong to a polyline but is considered relevant or meaningful.

The basic idea of line simplification consists of removing points and representing the original curve using approximation with fewer vertices. Figure 1(b) presents an example of the simplification of the lines shown in Figure 1(a). Two famous and frequently used line simplification algorithms are the Ramer-Douglas-Peucker's algorithm (RDP) [Douglas and Peucker 1973, Ramer 1972] and Visvalingam-Whyatt (VW) [Visvalingam and Whyatt 1993] algorithm.

The line simplification process can bring inconsistencies to the output if some care is not taken. Figure 2 shows two examples where removing certain points from the polylines in Figure 1(a) would cause topological inconsistency: (a) after simplification, point P is on the other side of the simplified line L_2; (b) a "nonexistent" intersection between lines L_1 and L_2 is created.

Topological inconsistency may be created by some simplification algorithms such as the ones based on the RDP method. But there is another source of error that affects even algorithms that attempt to avoid inconsistencies: round-off errors resulting from floating point arithmetic. These errors occur because real numbers cannot be exactly

represented in computational systems, instead, an approximation of the real number is used [Goldberg 1991]. In order to overcome such problems, the best strategy is to make use of Exact Geometric Computation [Li et al. 2005].

In this paper is presented a method that uses rational numbers and parallel computing to solve the following variation of the generalization problem: given a set of polylines and control points, the goal is to simplify these polylines by removing some of their vertices (except endpoints) such that topological relationships between pairs of polylines and between polylines and control points are maintained. In practice, polylines may represent boundaries of counties or states, and control points may represent cities within these states. The introduction of rational numbers was used to prevent errors introduced by rounding in floating point arithmetic. The use of arbitrary precision numbers is expected to increase the overall execution time of the algorithm since its operations are more complex. In order to compensate this performance drop, parallel computing is used.

3. Related Works

In this section we describe algorithms for line simplification as well as problems that arise from floating-point arithmetic.

3.1. Algorithms for Line Simplification

Many algorithms for line simplification have been developed so far. One of the most famous is the Ramer-Douglas-Peucker's algorithm (RDP) [Douglas and Peucker 1973, Ramer 1972]. Its basic idea is to start with a very rough approximation of the original line (i.e. a straight line connecting the end vertices) and iteractively refine the approximation including, in each step, the vertex that is farthest from the current line. The method stops when the distance between the farthest vertex and the line is greater than a given threshold (the smaller the threshold the less simplified the line is).

The RDP algorithm does not take topological consistency into consideration and may generate inconsistent results. An approach proposed by Saalfeld [Saalfeld 1999] attempts to avoid such inconsistencies. It uses Douglas-Peucker's algorithm to simplify lines and then starts a refining process by adding points to the output line so that the curve no longer presents any inconsistency. Noteworthy to mention that adding points to a curve may eliminate previous inconsistencies but may create new ones.

Another approach based on Douglas-Peucker was proposed by Li et al. [Li et al. 2013]. It intends to avoid topological inconsistencies as well as cracks on polygon shapes using a strategy based on detection-point identification, which are points lying within a minimum boundary rectangle (MBR) of the bounded face formed by a subpolyline and its corresponding simplifying segment. These detection-points are used for consistency verification of the simplification process.

Visvalingam and Whyatt [Visvalingam and Whyatt 1993] proposed a method (called the *VW* algorithm) for line generalization that uses the concept of *effective area* of a point to define the priority of its removal. The *effective area* of a point v_i, for $1 < i < n$, in a polygonal chain $v_1, \cdots, v_n$, is defined as the area of the triangle formed by v_i and its two adjacent vertices, namely, v_{i-1}, v_i, v_{i+1}. The *VW* algorithm considers that the "importance" of the points are proportional to their *effective area* and, therefore, it ranks the points and simplifies the polylines by removing first the points with smaller areas.

Even though the *VW* algorithm performs simplification with good quality, it does not avoid topological problems in the map. To solve this problem, Gruppi et al.[Gruppi et al. 2015] developed *TopoVW*, a variation of the *VW* algorithm that avoids the creation of topological inconsistencies. Similarly to *VW*, *TopoVW* processes the points in an order based on their *effective area* but only removes a point v_i if its removal does not create inconsistencies in topology. When a point is removed the effective areas of its two neighbor points in the line are updated since the triangle associated with them change. *TopoVW* may be configured to stop when the number of points removed reaches a limit or when the smallest effective area of the points is greater than a given threshold.

Although some of the methods previously mentioned have mechanisms to detect and prevent topological inconsistencies created by the simplification process itself, these problems may still happen because of round-off errors related to the use of inexact arithmetic to process the points' coordinates.

3.2. Round-off Errors in Floating Point Arithmetic

The computational representation of a non-integer number is made by adjusting this number to a finite sequence of bits, this possibly causes the number to be an approximation most of the time. Furthermore, even if some numbers can be exactly represented, arithmetic operations applied to these numbers may generate a result that is not exactly correct. In geometric algorithms this is a great issue since they may result in inconsistent outputs.

Kettner et al. [Kettner et al. 2008] presented a study of how rounding in floating point arithmetic affects the planar orientation predicate and as consequence the planar convex hull problems. The planar orientation predicate is the problem of finding whether three points p, q, r are collinear, make a left turn, or make a right turn. This predicate is computed by verifying the sign of a determinant involving the points.

This determinant will be positive, negative or zero which means that points (p, q, r) form a left turn, right turn or are collinear, respectively. Due to round-off errors in floating point arithmetic the results can be classified incorrectly due to *rounding to zero*, *perturbed zero*, or *sign inversion*. Respectively, it means a non-zero result may be rounded to zero, a zero result may be mis-classified as positive or negative, and a positive result may be mis-classified as negative or vice-versa.

To observe the occurrence of issues caused by floating-point arithmetic, Kettner et al. [Kettner et al. 2008] developed a program to apply planar orientation predicate ($orientation(p, q, r)$) on a point $p = (p_x + xu, p_y + yu)$ where u is the step between adjacent floating point numbers in the range of p and $0 \leq x, y \leq 255$. This results in a 256×256 matrix containing either 1, -1 or 0 if the point corresponding to the matrix position is to the right, to the left or on the line that passes through q and r. Figure 3 shows the geometry of this experiment for $p = (0.5, 0.5)$, $u = 2^{-53}$ and $q = (12, 12)$ and $r = (24, 24)$. White cells represent correct output. The black diagonal line is an approximation of line (q, r). Black cells represent incorrect output, that is, black points above the diagonal were considered to form a right turn with the line (q, r), which is not true, it also applies to the points below the diagonal which were said to form a left turn with line (q, r). Gray cells contain points considered collinear to (q, r). According to Kettner et al., even using extended double arithmetic was not enough to overcome this issue.

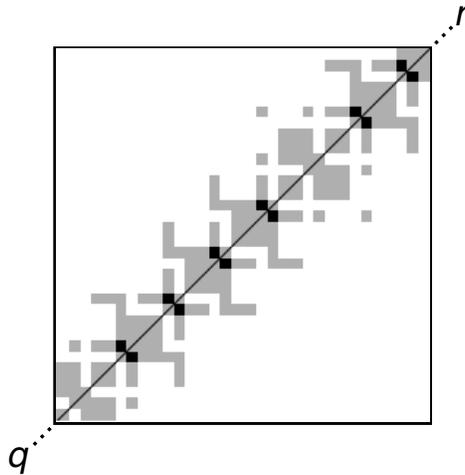As shown by [Kettner et al. 2008], these inconsistent results in

Figure 3. Geometry of the planar orientation predicate for double precision floating point arithmetic. White points represent correct outputs, gray were considered collinear and black cells points considered to be in the wrong side of the line. The diagonal line is an approximation to line (q, r). This Figure was created based on the experiments performed by Kettner et al.[Kettner et al. 2008].

$orientation(p, q, r)$ predicate could make algorithms that use this predicate (such as the Incremental Convex Hull algorithm) to fail.

A well-known technique to get around round-off errors in floating point arithmetic is the Epsilon-tweaking, that consists in comparing numbers using a relatively small tolerance value epsilon (ϵ). In practice, epsilon-tweaking fails in several situations [Kettner et al. 2008]. Snap rounding is another method to approximate arbitrary precision segments into fixed-precision numbers [Hobby 1999]. However, Snap rounding can generate inconsistencies and deform the original topology if applied consecutively on a data set. Some variations of this technique attempt to get around these issues [de Berg et al. 2007, Hershberger 2013].

One of the most robust ways for eliminating rounding errors in geometry is by using Exact Geometric Computation (ECG). According to Li [Li et al. 2005], any problem handled by other approaches can also be solved by ECG. Additionally, ECG can do even more and the solutions may be of higher quality. This can be achieved by using arbitrary precision rational numbers [Li et al. 2005], which eliminates rounding errors but considerably decreases performances as most operations are more computationally intensive.

4. Evaluation of Round-off Errors on Map Simplification

Similarly to other geometric problems, map simplification is also affected by round-off errors. As mentioned in section 3, *TopoVW* processes points in an order defined by their effective areas and only removes a point if its removal does not cause topological inconsistencies on the map. Given a polyline point v from a map, the removal of v causes a topological inconsistency if and only if there is another point (that may be a polyline or a control point) inside the triangle formed by v and its two adjacent vertices in its polyline.

If the *point-in-triangle* test fails returning a false positive a point that could have

been removed from the polyline will not be removed. If this test returns a false negative, on the other hand, topological inconsistencies may be created on the map.

In *TopoVW*, the test to determine if a point p lies inside the triangle T formed by points r, s and t is performed by computing the barycentric coordinates of p in T, i.e., p is expressed in terms of three scalars a, b and c such that $p_x = ar_x + bs_x + ct_x$, $p_y = ar_y + bs_y + ct_y$, and $a + b + c = 1$. Point p lies in T if and only if $0 \leq a \leq 1$ and $0 \leq b \leq 1$ and $0 \leq c \leq 1$. A function *is_inside*(r, s, t, p) to perform the *point in triangle* test using the barycentric coordinates was implemented in C++. This approach is similar to the one used by Kettner et al. shown in Section 3.2.

In a similar manner to the orientation test presented in the previous section, the function *is_inside*(r, s, t, p) may also return incorrect results in two situations:

- *false inside*: erroneously determine an outer point as inside;
- *false outside*: erroneously determine an inner point as outside;

Since *is_inside* is *TopoVW*'s key operation, the method may avoid simplifying lines due to *false inside* appearance. Even more alarming, it may remove points on the presence of *false outsides*, what would change the topological relationships. Figure 4 shows an example of *false outside* simplification. In this example there are two non-intersecting lines (solid and dashed) as shown in Figure 4(a), the zoomed area shows explicitly that both lines do not intersect. Point p is inside the triangle formed by points (r, q, w) with w not shown in the figure to preserve simplicity. However, due to a *false outside* failure point q is removed resulting in an intersection as seen in Figure 4(b).

Another instance of this problem is shown by Figure 5, where a single line is simplified. Similarly to the previous example, vertex p is inside the triangle formed by (r, q, w) but it is seen as a *false outside*. Vertex q is removed by the simplification process causing the line to self-intersect as seen in Figure 5(b).

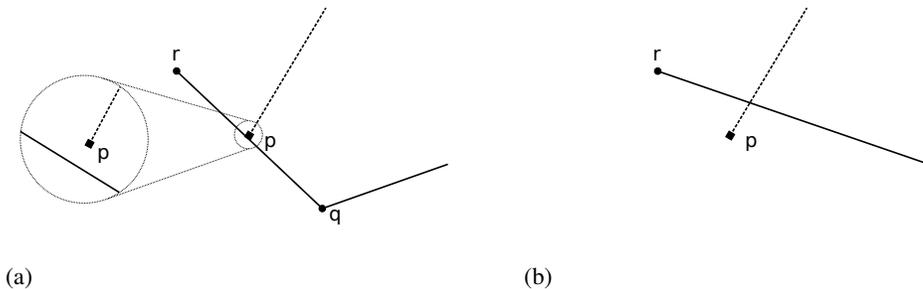(a)   (b)

Figure 4. (a) Example input on which false outside failure occur, two lines (solid and dashed) do not intersect. (b) Result of simplification with false outside, the removal of point q causes the lines intersect after simplification.

5. The *EPLSimp* Method

To avoid adding topological errors to the map in the situations described in section 4, we developed *EPLSimp*, a simplification algorithm based on *TopoVW* that uses exact arithmetic to completely avoid the round-off errors that may happen during the point in triangle tests. In *EPLSimp*, all non-integers variables are represented using arbitrary-precision

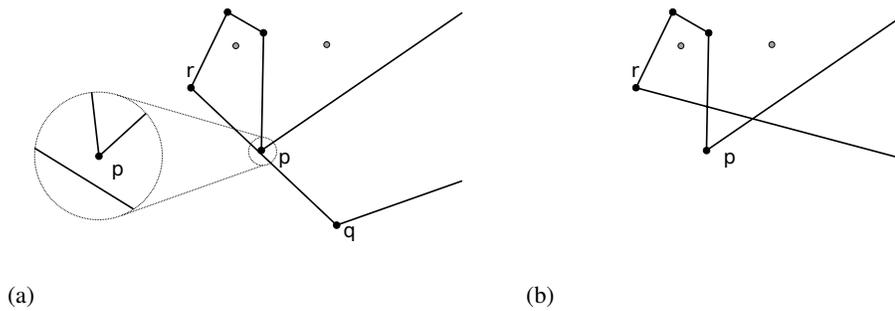(a)   (b)

Figure 5. (a) Example input of a single line and the occurrence of a false outside. (b) Simplification with a false outside point. The removal of point q produces a self-intersecting line.

rational numbers. Since exact arithmetic is usually much slower than arithmetic with floating point numbers (that usually can be performed natively on the CPU), some optimizations were implemented in order to reduce the performance penalty that it introduces.

First, similarly to *TopoVW*, we used a uniform grid to index the polyline and control points from the map. The idea is to create a regular grid, superimpose it with the map and insert in each cell c the control points and polyline points that are inside c. Then, given a triangle T, only points in the uniform grid cells intersecting T need to be tested in order to verify if there is a point inside T.

One advantage of the uniform grid over more complex data structures such as quadtrees is that it is easier to be constructed and maintained. Given a set S of points, we compute the uniform grid by performing only one pass through the dataset: for each point p in S, the cell c from the grid where p should be is computed (by dividing p's coordinates by the dimensions of the grid cells) and p is inserted in c.

Since the slowest step during the construction of the grid is the computation of the cell in which each point p is (due to the division operations with arbitrary-precision rationals), we used parallel programming to accelerate this step. The idea is to pre-compute in parallel the cell in which each point is and, after that, insert the points in the cells (this insertion step is not done in parallel in order to avoid the cost of synchronizations).

After indexing the points, the next step consists in simplifying polylines. As mentioned in section 3, *TopoVW* sorts the points based on their effective areas and processes them by removing the ones whose removal would not create topological problems in the map. To accelerate the simplification process used in *TopoVW*, we subdivided the polylines into sets such that polylines from different sets may be simplified independently in parallel not requiring the synchronization of data structures accesses.

Algorithm 1 presents the simplification algorithm and the strategy used for subdividing the polylines into sets that can be simplified in parallel. This subdivision is also performed using a uniform grid (this grid may have a resolution different from the uniform grid used for indexing the points). We create this new uniform grid and, then, insert in each grid cell the polylines that are completely inside this cell. The polylines in different grid cells could be processed independently since the triangle formed by any polyline point never contains a point from another cell. On the other hand, polylines intersecting

more than one cell cannot be processed in parallel without synchronization. For example, even though the polyline containing the vertex v in Figure 6 (a) does not intersect the cell containing the polygon P, before deleting v it is necessary to access the cell containing polygon P in order to verify if the deletion of v causes a topological inconsistency. Therefore, if the two polylines in this figure are simplified in parallel the algorithm would need to perform synchronizations.
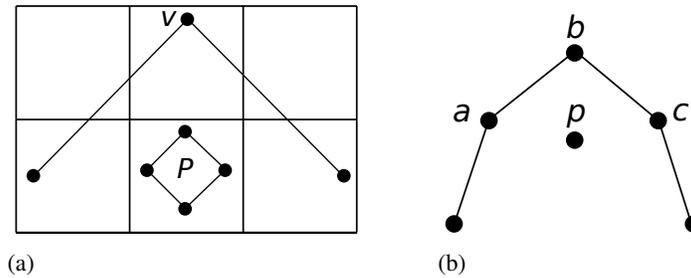

(a)   (b)

Figure 6. (a) Example where a polyline intersecting multiple cells needs to access data in a cell it does not intersect. (b) Example where the deletion of a point makes the deletion of other points infeasible .

After processing all the polylines lying completely in single cells, we repeat the simplification process for the polylines intersecting more than one cell. In order to be able to do that in parallel, we reduce the uniform grid resolution, reclassify the remaining polylines and, then, simplify the ones that lie in single cells in this new uniform grid. This process is repeated until there is no more polyline to be simplified (eventually all the polylnes will be processed since when the uniform grid is reduced to one cell all polylines that were not processed yet will lie in this unique cell).

To avoid the necessity of synchronizations between threads processing different sets of polylines, the simplification stopping criteria used in *EPLSimp* is the effective area of the points. That is, the thread simplifying a set of polylines stops the process whenever the point with smallest effective area in the set has an area greater than a given threshold. If the stopping criteria was the number of points removed, synchronizations would be necessary to ensure that all threads stop simplifying lines when the global number of removed points reaches the target number.

It is important to mention that we have considered other two parallelization strategies. First, we could pre-process the map verifying for each point if there is another point inside the triangle defined by it and its two neighbors. This pre-processing could be performed in parallel. After labeling the points that can safely be removed (that is, the ones without other points in their triangles), we could just remove the ones with smaller effective areas. This strategy would not work very well because when a point is removed the triangle of its two neighbors change. For example, in Figure 6 (b), any of the points a or b or c may be removed without changing the topological relationship between the polyline and the control point p. However, if a or c is removed the triangle associated with b will contain p and, therefore, b will not be a candidate to be removed anymore.

Another parallel strategy would be to perform the point inside triangle test in parallel. That is, given a triangle T, after using the uniform grid to select the points that are candidate to be in T we could perform the test to verify if each point is really inside T

169

Algorithm 1 Parallel map simplification algorithm.

1: M: input map
2: $MaxArea$: maximum effective area of a point to be removed
3: $GridSize$: initial resolution of the uniform grid used to separate the polylines.
4: **while** $GridSize > 0$ **do**
5:     $ug \leftarrow GridSize \times GridSize$ uniform grid
6:     **for** each polyline p in M not simplified yet **do**
7:         **if** p is completely inside a cell c from ug **then**
8:             Insert p into c
9:         **end if**
10:     **end for**
11:     **for** each cell c in ug **do** //Parallel for loop
12:         //Iterate in an order based on the points' effective areas
13:         **for** each point v_i in polylines from $c \mid effectiveArea(v_i) < MaxArea$ **do**
14:             **if** $\nexists$ point $p \mid is\_inside(v_{i-1}, v_i, v_{i+1}, p)$ **then**
15:                 Remove the point v_i from its polyline
16:             **end if**
17:         **end for**
18:     **end for**
19:     $GridSize \leftarrow GridSize/2$
20: **end while**

---

in parallel. However, preliminary experiments showed that, because of the uniform grid, the average number of points that need to be effectively tested in this step is usually small and, therefore, the performance gain obtained by processing them in parallel would not be good if compared with the overheads associated with the parallelism.

6. Experimental Evaluation

We evaluated *EPLSimp* by implementing it in C++ (the library GM-PXX [Granlund and the GMP development team 2014] was used to provide arbitrary precision arithmetic) and performing experiments in some small datasets artificially generated to contain polylines and control points that would introduce topological errors in the simplification performed by *TopoVW*. Furthermore, experiments were performed in 3 real-world maps in order to evaluate the performance of *EPLSimp*. The computer used has a dual E5-2687 8-core/16-thread Intel Xeon CPU and 128 GB of RAM.

In the first set of experiments, we used the maps artificially generated to contain points in positions where the point-in-triangle tests would give a false negative answer (similar to the examples presented in section 4) and, therefore, methods such as *TopoVW* would create topological errors during the map simplification. As expected, because of the use of exact arithmetic, *EPLSimp* was able to simplify these maps without creating any topological inconsistency.

Next, we performed experiments in three datasets to verify the overhead added by the use of arbitrary precision rational numbers in *EPLSimp*. Dataset 1 was the largest dataset used in the ACM GISCUP competition 2014. It contains 30000 polyline points and 1607 control points. Dataset 2 represents the Brazilian county subdivision map avail-

able in the IBGE (the Brazilian geography agency) website and it contains 300000 polyline points and 10000 control points (the control points were positioned randomly in the map). Dataset 3 represents the United States county subdivision map available in the United States Census website and it has 4 million polyline points and 10 million control points (that were also positioned randomly in the map).

The choice of the dimensions of the uniform grid used by *TopoVW* and *EPLSimp* to index the points affects the performance of both methods and it can be performed using several strategies. For example, *TopoVW* automatically defines the grid size by computing the total number of polylines/control points in the map and chooses the grid dimension estimating the average number of points per cell close to a constant (this constant was determined experimentally). Since the best grid size for *TopoVW* may not be the best grid size for *EPLSimp* and since we want to compare the performance of these two methods, we chose experimentally, for each method and dataset, a configuration that presents the best performance (for example, in dataset 2, *TopoVW* and *EPLSimp* used grids with, respectively, 512^2 and 2048^2 cells).

The uniform grid that *EPLSimp* uses to classify the polylines that are processed in parallel was configured to have initially 256^2 cells and to iteratively reduce the resolution to half after completely processing each set of polylines that can be processed in parallel. As mentioned in section 5, this process is repeated until all polylines have been simplified, what happens, in the worst case, when the grid has only one cell.

Table 1 presents the wallclock-time (in milliseconds) of the two methods in two situations: in the first one they were configured to remove the maximum amount of points that they can remove without creating topological errors. In the second one, they were configured to remove 50% of the points. Row *initialize* contains the time for initializing the algorithm and includes the time for creating the data structures (such as the uniform grids). Row *simplify* contains the time spent in the simplification process. In all tests *EPLSimp* was tested using 16 *threads*.

EPLSimp was, on average, less than twice slower than *TopoVW*, even though we store and process all points coordinates using arbitrary precision rational numbers, that are much more computationally expensive to process than floating point numbers. This happens because *EPLSimp* was carefully implemented using techniques such as parallel computing and the uniform grid to accelerate the simplification process. It is worth mentioning that one of the advantages of the uniform grid over other indexing techniques (such as Quadtrees) is that it is easily parallelizable and can be created by performing a single pass over the data (this is particularly important for efficiency since the indexing is performed using coordinates represented by rational numbers).

Table 2 evaluates the scalability of *EPLSimp* considering 5 different number of threads. In these datasets, *EPLSimp* had a speedup of $2\times$ when two threads were used and this speedup increased slowly for larger amounts of threads. For example, the running-time using 16 threads was not much different from the time using 8 threads. Some reasons for this behavior are: first, due to Amdahl's law, sequential parts of the algorithm limits its scalability; furthermore, some polylines sets may take more time to be simplified than others, what causes load imbalance in the threads; finally, when several threads run in parallel the memory accesses may saturate the memory bus. Anyway, it is worth mentioning

Table 1. Times (in ms) for the main steps of the map simplification algorithms. Rows *Max.* represents the time for removing the maximum amount of points from the map while rows *Half* represents the time to remove half of the points.

	Dataset	1		2		3	
	Method	TopoVW	*EPLSimp*	TopoVW	*EPLSimp*	TopoVW	*EPLSimp*
Max.	Initialize	4	22	28	190	1828	5353
	Simplify	39	60	626	445	46069	57095
	Total	43	82	654	635	47897	62448
Half	Initialize	4	22	28	186	1847	5447
	Simplify	25	41	357	331	23021	48090
	Total	29	63	384	517	24868	53537

Table 2. Times (in ms) for initializing and simplifying maps from the 3 datasets considering different number of threads. The simplification was configured to remove the maximum amount of points from the maps.

		Initialization			Simplification		
	Dataset	1	2	3	1	2	3
Threads	1	71	655	26833	176	1574	250237
	2	91	568	15483	152	1150	131310
	4	54	422	9853	99	689	82641
	8	34	240	6552	61	483	62089
	16	22	190	5353	60	445	57095

that typical computers nowadays have 2 or 4 cores and, therefore, *EPLSimp* is able to present a good scalability in those computers.

7. Conclusion and Future Works

This paper presented *EPLSimp*, an algorithm for map simplification that does not produce topological inconsistencies. It uses arbitrary precision numbers to avoid round-off errors caused by floating-point arithmetic, which could lead to topological inconsistencies even in methods designed to avoid these problems, such as *TopoVW*.

EPLSimp was implemented to be efficient even though it uses arbitrary precision numbers, which are much slower to be processed than floating-point numbers. This efficiency improvement was achieved by using a uniform grid to index the geometric objects and, also, high performance computing. As a result, using 16 threads *EPLSimp* was, on average, less than twice slower than *TopoVW*, even though the latter performs all computation using inexact floating-point numbers (that are natively supported by the CPU) and then can generate "wrong" (or inconsistent) results.

For future work, we intend to develop other GIS algorithms using arbitrary precision arithmetic. Furthermore, adapting *EPLSimp* to simplify vector drawings and 3D objects is also an interesting future research topic.

8. Acknowledgement

References

de Berg, M., Halperin, D., and Overmars, M. (2007). An intersection-sensitive algorithm for snap rounding. *Computational Geometry*, 36(3):159–165.

Douglas, D. H. and Peucker, T. K. (1973). Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica: The International Journal for Geographic Information and Geovisualization*, 10(2):112–122.

Goldberg, D. (1991). What every computer scientist should know about floating-point aritmetic. *ACM Computing Surveys*, 23(1):5–48.

Granlund, T. and the GMP development team (2014). The gnu multiple precision arithmetic library.

Gruppi, M. G., Magalhães, S. V. G., Andrade, M. V. A., Franklin, W. R., and Li, W. (2015). An efficient and topologically correct map generalization heuristic. In *Proceedings of the 17th International Conference on Enterprise Information Systems (ICEIS)*.

Hershberger, J. (2013). Stable snap rounding. *Computational Geometry*, 46(4):403–416.

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

Jiang, B., Liu, X., and Jia, T. (2013). Scaling of geographic space as a universal rule for map generalization. *Annals of the Association of American Geographers*, 103(4):844–855.

João, E. (1998). *Causes and Consequences of map generalization*. CRC Press.

Kettner, L., Mehlhorn, K., Pion, S., Schirra, S., and Yap, C. (2008). Classroom examples of robustness problems in geometric computations. *Computational Geometry*, 40(1):61–78.

Li, C., Pion, S., and Yap, C.-K. (2005). Recent progress in exact geometric computation. *The Journal of Logic and Algebraic Programming*, 64(1):85–111.

Li, L., Wang, Q., Zhang, X., and Wang, H. (2013). An algorithm for fast topological consistent simplification of face features. *Journal of Computational Information Systems*, 9(2):791–803.

Ramer, U. (1972). An iterative procedure for the polygonal approximation of plane curves. *Computer Graphics and Image Processing*, 1(3):244–256.

Saalfeld, A. (1999). Topologically consistent line simplification with the douglas-peucker algorithm. *Cartography and Geographic Information Science*, 26(1):7–18.

Visvalingam, M. and Whyatt, J. (1993). Line generalisation by repeated elimination of points. *The Cartographic Journal*, 30(1):46–51.