Simplifying Massive Contour Maps

Lars Arge[1], Lasse Deleuran[1], Thomas Mølhave[2⋆],
Morten Revsbæk[1], and Jakob Truelsen[3]

[1] MADALGO[†], Department of Computer Science, Aarhus University
[2] Department of Computer Science, Duke University.
[3] SCALGO, Scalable Algorithmics, Denmark

Abstract. We present a simple, efficient and practical algorithm for constructing and subsequently simplifying contour maps from massive high-resolution DEMs, under some practically realistic assumptions on the DEM and contours.

1 Introduction

Motivated by a wide range of applications, there is extensive work in many research communities on modeling, analyzing, and visualizing terrain data. A (3D) digital elevation model (DEM) of a terrain is often represented as a planar triangulation $\mathbf{M}$ with heights associated with the vertices (also known as a triangulated irregular network or simply a TIN). The l-level set of $\mathbf{M}$ is the set of (2D) segments obtained by intersecting $\mathbf{M}$ with a horizontal plane at height l. A *contour* is a connected component of a level set, and a *contour map* $\mathcal{M}$ the union of multiple level sets; refer to Figure 1. Contour maps are widely used to visualize a terrain primarily because they provide an easy way to understand the topography of the terrain from a simple two-dimensional representation.

Early contour maps were created manually, severely limiting the size and resolution of the created maps. However, with the recent advances in mapping technologies, such as laser based LIDAR technology, billions of (x, y, z) points on a terrain, at sub-meter resolution with very high accuracy ($\sim$10-20 cm), can be acquired in a short period of time and with a relatively low cost. The massive size of the data (DEM) and the contour maps created from them creates problems, since tools for processing and visualising terrain data are often not designed to handle data that is larger than main memory. Another problem is that contours generated from high-resolution LIDAR data are very detailed, resulting in a large amount of excessively jagged and spurious contours; refer to Figure 1. This in turn hinders their primary applications, since it becomes difficult to interpret the maps and gain understanding of the topography of the terrain. Therefore we are interested in simplifying contour maps.

⋆ This work is supported by NSF under grants CCF-06 -35000, CCF-09-40671, and CCF-1012254, by ARO grants W911NF-07-1-0376 and W911NF-08-1-0452, and by U.S. Army ERDC-TEC contract W9132V-11-C-0003.

† Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation

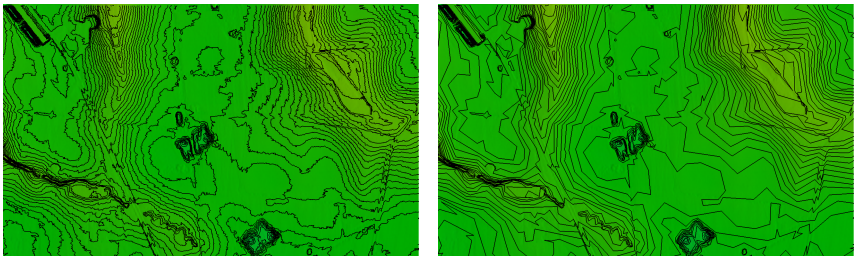

Fig. 1. A snapshot of contours generated from the Danish island of Als. The contour granularity is $\Delta = 0.5$m, the xy-constraints was $\varepsilon_{xy} = 4$m and the vertical constraint was $\varepsilon_z = 0.2$m. Left: Original map $\mathcal{M}$, right: simplified map.

Previous work: The inefficiency of most tools when it comes to processing massive terrain data stems from the fact that the data is too large to fit in main memory and must reside on slow disks. Thus the transfer of data between disk and main memory is often a bottleneck (see e.g. [10]). To alleviate this bottleneck one needs algorithms designed in the I/O-model of computation [5]. In this model, the machine consists of a main memory of size M and an infinite-size disk. A block of B consecutive elements can be transferred between main memory and disk in one *I/O operation* (or simply *I/O*). Computation can only take place on elements in main memory, and the complexity of an algorithm is measured in terms of the number of I/Os it performs. Over the last two decades, I/O-efficient algorithms and data structures have been developed for several fundamental problems. See recent surveys [6, 19] for a comprehensive review of I/O-efficient algorithms. Here we mention that scanning and sorting N elements takes $O(\text{Scan}(N)) = O(N/B)$ and $O(\text{Sort}(N)) = O(N/B \log_{M/B}(N/B))$ I/Os, respectively. The problem of computing contours and contour maps has previously been studied in the I/O-model [3, 2]. Most relevant for this paper, Agarwal *et al.* [2] presents an optimal I/O-efficient algorithm that computes a contour map in $O(\text{Sort}(N) + \text{Scan}(|\mathcal{M}|))$ I/Os, where $|\mathcal{M}|$ is the number of segments in the contour map and N is the number of triangles in the DEM. It also computes the nesting of the contours and the segments around each contour are produced in sorted order. However, this algorithm is not practical.

Although the problem that contour maps generated from high-resolution LIDAR data contain excessively jagged contours can be alleviated by contour map simplification (while also alleviating some of the scalability problems encountered when processing contour maps), the main issue is of course to guarantee the accuracy of the simplified contour map. There are two fundamental approaches to simplifying a contour map: The DEM $\mathbf{M}$ can be simplified before computing the contour map $\mathcal{M}$, or $\mathcal{M}$ can be simplified directly. There has been a lot of work on simplifying DEMs; refer e.g. to [9, 14] and the references therein. However, most often the simple approaches do not provide a guarantee on the simplification accuracy, while the more advanced approaches are not I/O-efficient and therefore do not scale to large data sets. Developing I/O-efficient DEM simplification algorithms with simplification guarantees has shown to be

a considerable challenge, although an $O(\text{Sort}(N))$ I/O (topological persistence based) algorithm for removing "insignificant" features from a DEM (resulting in small contours) has recently been developed [4]. Simplifying the contour map $\mathcal{M}$ directly is very similar to simplifying a set of polygons (or polygonal lines) in the plane. Polygonal line simplification is a well studied problem; refer to [16] for a comprehensive survey. However, there are at least three important differences between contour maps and polygonal line simplification. Most noticeably, simplifying a contour line in the plane using a polygonal line simplification algorithms will, even if it guarantees simplification accuracy in the plane, not provide a z-accuracy guarantee. Furthermore, simplifying the contours individually may lead to intersections between the simplified contours. Finally, when simplifying contour maps its very important to preserve the relationships between the contours (the *homotopic* relationship), that is, maintain the nesting of the contours in the map. Note that intersections are automatically avoided and homotopy preserved when simplifying the DEM before computing the contour map.

One polygonal line simplification algorithm that is often favored for its simplicity and high subjective and objective quality on real life data is the Douglas-Peucker line simplification algorithm [12]. The algorithm simplifies a contour by removing points from the contour while ensuring that the distance between the original and the simplified contour is within a distance parameter ε_{xy} (but it does not guarantee that it removes the optimal number of endpoints under this constraint). Modifications of this algorithm have been developed, that remove self-intersections in the output [18], as well as ensure homotopy relative to a set of obstacles (polygons) [11, 8]. However, these modified algorithms are complicated and/or not I/O-efficient (and do also not consider z-accuracy).

Our results: In this paper we present a simple, efficient and practical algorithm for constructing and subsequently simplifying contour maps from massive high-resolution DEMs, under some practically realistic assumptions on the DEM and contours. The algorithm guarantees that the contours in the simplified contour map are homotopic to the unsimplified contours, non-intersecting, and within a distance of ε_{xy} of the unsimplified contours in the xy plane. Furthermore, it guarantees that for any point p on a contour in the l-level-set of the simplified contour map, the difference between l and the elevation of p in $\mathbf{M}$ (the z-error) is less than ε_z. We also present experimental results that show a significant improvement in the quality of the simplified contours along with a major (about 90%) reduction in size.

Overall, our algorithm has three main components. Given the levels $\ell_1, \ldots, \ell_d$, the *first component*, described in Section 3, computes the segments in the contour map $\mathcal{M}$. The component also computes level-sets for each of the levels $\ell_i \pm \varepsilon_z$, $1 \leqslant i \leqslant d$. The contours generated from these extra levels will be used to ensure that the z-error is bounded by ε_z. We call these contours *constraint* contours and mark the contours in $\mathcal{M}$ that are not constraint contours. The component also orders the segments around each contour and computes how the contours are nested. It uses $O(\text{Sort}(|\mathcal{M}|))$ I/Os under the practically realistic assumptions that each contour, as well as the contour segments intersected by any horizontal

line, fit in memory. This is asymptotically slightly worse than the theoretically optimal but complicated algorithm by Agarwal *et al.* [2]. The *second component*, described in Section 4, computes, for each of the marked contours P, the set $\mathcal{P}$ of contours that need to be considered when simplifying P. Intuitively, $\mathcal{P}$ contains all (marked as well as constraint) contours that can be reached from P without crossing any other contour of $\mathcal{M}$. Although each contour can be involved in many sets $\mathcal{P}$, the entire algorithm uses only $O(\text{Sort}(|\mathcal{M}|))$ I/Os. The *third component*, described in Section 5, simplifies each marked contour P within $\mathcal{P}$. This is done using a modified version of the Douglas-Peucker line simplification algorithm [12]. As with the Douglas-Peucker algorithm it guarantees that the simplified contour P' is within distance ε_{xy} of the original contour P, but it also guarantees that P' is homotopic to P (with respect to $\mathcal{P}$) and that P' does not have self intersections. The existence of the constraint contours in $\mathcal{P}$ together with the homotopy guarantee that the z-error is within ε_z. Under the practically realistic assumptions that each contour P along with $\mathcal{P}$ fits in internal memory, the algorithm does not use any extra I/Os.

Finally, the details on the implementation of our algorithm are given in Section 6 along with experimental results on a terrain data set of Denmark with over 12 billion points.

2 Preliminaries

Terrains: We consider the terrain $\mathbf{M}$ to be represented as a triangular irregular network, which is a planar triangulation whose vertices v are associated with heights $h(v)$. The height of a point interior to a triangle is determined by linearly interpolating from the triangle vertex heights; we use $h(p)$ to refer to the height of any point p in $\mathbb{R}^2$.

Paths and polygons: Let $p_1, \ldots, p_n$, be a sequence of $n > 1$ points in $\mathbb{R}^2$. The *path* Q defined by these points is the set of line segments defined by pairs of consecutive points in the sequence. The points $p_1, \ldots, p_n$ are called the *vertices* of Q. A *simple path* is a path where only consecutive segments intersect, and only at the endpoints. Given integers $1 \leqslant i < j \leqslant n$, the *sub-path* Q_{ij} is the path defined by the vertices $p_i, p_{i+1}, \ldots, p_j$. We abuse notation slightly by using Q to denote both the sequence of vertices, and the path itself. We define the size of Q as its number of segments, i.e. $|Q| = n - 1$. A path Q' is a *simplification* of a path Q if $Q' \subseteq Q$ and the vertices of Q' appear in the same order as in Q.

A *polygon* (*simple polygon*) is a path (simple path) P where $p_1 = p_n$. A simple polygon P partitions $\mathbb{R}^2 \setminus P$ into two open sets — a bounded one called *inside* of P and denoted by P^{i}, and an unbounded one called *outside* of P and denoted by P^{o}. We define a *family of polygons* to be a set of non-intersecting and vertex-disjoint simple polygons. Consider two simple polygons P_1 and P_2 in a family of polygons A. P_1 and P_2 are called *neighbors* if no polygon $P \in A$ separates them, i.e., there is no P where one of P_1 and P_2 is contained in P^{i} and the other in P^{o}. If P_1 is a neighbor of P_2 and $P_1 \subset P_2^{\text{i}}$, then P_1 is called a *child* of P_2, and P_2 is called the *parent* of P_1; we will refer to the parent of P

(a) (b)

Fig. 2. (a) Polygonal domain $\mathcal{P}$ (solid lines) of P (dashed lines). (b) Polygon P' is homotopic to P in $\mathcal{P}$, Q is not.

as $\hat{P}$. The *topology* of a family of polygons $\mathcal{M}$ describes how the polygons are nested i.e. the parent/child relationship between polygons. Given a polygon P in A, the *polygonal domain of* P, denoted $\mathcal{P}$, consists of the neighbors $P_1 \dots P_k$ of P in A; refer to Figure 2(a). We define the size of $\mathcal{P}$ to be $|\mathcal{P}| = |P| + \sum_i |P_i|$.

Intuitively, two paths Q and Q' are *homotopic* with regards to a polygonal domain $\mathcal{P}$ if one can be continuously transformed into the other without intersecting any of the polygons of $\mathcal{P}$; refer to Figure 2. Path Q' is *strongly homotopic* to Q if Q' is a simplification of Q and if every segment $q'_i q'_{i+1}$ in Q' is homotopic to the corresponding sub-path $Q_{k,l}$ where $q'_i = q_k$ and $q'_j = q_l$. It follows that Q and Q' are also homotopic, but the reverse implication does not necessarily hold.

Given two indices $1 \leqslant i, j \leqslant n$ we define the distance $d(p, i, j)$ between any point $p \in \mathbb{R}^2$ and line segment $p_i p_j$ as the distance from p perpendicular to the line defined by $p_i p_j$. We define the error $\varepsilon(i, j)$ of replacing $P_{i,j}$ with the line segment $p_i p_j$ to be the maximum distance between the vertices of P_{ij} and $p_i p_j$, i.e. $\varepsilon(i, j) = \max\{d(p_i, i, j), d(p_{i+1}, i, j), \dots, d(p_j, i, j)\}$. Let P' be a simplification of P. Given a *simplification threshold* ε, we say that P' is a *valid simplification* of P if it is a simple polygon homotopic to P in $\mathcal{P}$ and $\varepsilon(i, j) < \varepsilon$ for any segment $p_i p_j$ of P'.

Contours and contour maps: For a given terrain $\mathbf{M}$ and a *level* $\ell \in \mathbb{R}$, the *ℓ-level set* of $\mathbf{M}$, denoted by $\mathbf{M}_\ell$, is defined as $h^{-1}(\ell) = \{x \in \mathbb{R}^2 \mid h(x) = \ell\}$. A *contour* of a terrain $\mathbf{M}$ is a connected component of a level set of $\mathbf{M}$. Given a list of levels $\ell_1 < \ldots < \ell_d \in \mathbb{R}$, the *contour map* $\mathcal{M}$ of $\mathbf{M}$ is defined as the union of the level-sets $\mathbf{M}_{\ell_1}, \dots, \mathbf{M}_{\ell_d}$. For simplicity, we assume that no vertex of $\mathbf{M}$ has height $\ell_1, \dots, \ell_d$ and we assume that $\mathbf{M}$ is given such that M_{ℓ_1} consists of only a single boundary contour $\mathcal{U}$. This implies that the collection of contours in the contour map form a family of polygons and that each polygon in the family, except $\mathcal{U}$, has a parent. It allows us to represent the topology of $\mathcal{M}$ as a tree $\mathcal{T} = (V, E)$ where the vertices V is the family of polygons and where E contains an edge from each polygon $P \neq \mathcal{U}$ to its parent polygon $\hat{P}$. The root of $\mathcal{T}$ is $\mathcal{U}$. We will refer to $\mathcal{T}$ as the *topology tree* of $\mathcal{M}$.

3 Building the Contour Map

In this section we describe our practical and I/O-efficient algorithm for constructing the contour map $\mathcal{M}$ of the terrain $\mathbf{M}$ and the topology tree $\mathcal{T}$ of $\mathcal{M}$,

given a list of regular levels $\ell_1 < \ldots < \ell_d \in \mathbb{R}$. We will represent $\mathcal{M}$ as a sequence of line segments such that the clockwise ordered segments in each polygon P of $\mathcal{M}$ appear consecutively in the sequence, and $\mathcal{T}$ by a sequence of edges (P_2,P_1) indicating that P_2 is the parent of P_1; all segments in $\mathcal{M}$ of polygon P will be augmented with (a label for) P and the BFS number of P in $\mathcal{T}$. We will use that the segments in any polygon P in $\mathcal{M}$, as well as the segments in $\mathcal{M}$ intersecting any horizontal line, fit in memory.

Computing contour map $\mathcal{M}$: To construct the line segments in $\mathcal{M}$, we first scan over all the triangles of $\mathbf{M}$. For each triangle f we consider each level ℓ_i within the elevation range of the three vertices of f and construct a line segment corresponding to the intersection of z_{ℓ_i} and f. To augment each segment with a polygon label, we then view the edges as defining a planar graph such that each polygon is a connected component in this graph. We find these connected components practically I/O-efficiently using an algorithm by Arge et al. [7], and then we use the connected component labels assigned to the segments by this algorithm as the polygon label. Next we sort the segments by their label. Then, since the segments of any one polygon fit in memory, we can in a simple scan load the segments of each polygon P into memory in turn and sort them in clock-wise order around the boundary of P.

Computing the topology tree $\mathcal{T}$ of $\mathcal{M}$: We use a plane-sweep algorithm to construct the edges of $\mathcal{T}$ from the sorted line segments in $\mathcal{M}$. During the algorithm we will also compute the BFS number of each polygon P in $\mathcal{T}$. After the algorithm it is easy to augment every segment in $\mathcal{M}$ with the BFS number of the polygon P it belongs to in a simple sorting and scanning step.

For a given $\mu \in \mathbb{R}$, let y_μ be the horizontal line through μ. Starting at $\mu = y_\infty$, our algorithm sweeps the line y_μ though the (pre-sorted) edges of $\mathcal{M}$ in the negative y-direction. We maintain a search tree $\mathcal{S}$ on the set of segments of $\mathcal{M}$ that intersect y_μ. For each edge in $\mathcal{S}$ we maintain the invariant that we have already computed its parent and that each edge also knows the identity of its own polygon. The set of edges in $\mathcal{S}$ changes as the sweep-line encounters endpoints of segments in $\mathcal{M}$, and each endpoint v from some polygon P has two adjacent edges s_1 and s_2. If the other two endpoints of s_1 and s_2 are both above y_μ we simply delete s_1 and s_2 from $\mathcal{S}$. If both endpoints are below y_μ and there is not already a segment from P in $\mathcal{S}$, then this is the first time P is encountered in the sweep. We can then use $\mathcal{S}$ to find the closest polygon segment s_3 from some other polygon P_1 to the left of v. Depending on the orientation of s_3 we know that the parent of P, $\hat{P}$ is either P_1, or the parent of P_1 which is stored with P_1 in $\mathcal{S}$, and we can output the corresponding edge $(P, \hat{P})$ of $\mathcal{T}$. It is easy to augment this algorithm so that it also computes the BFS number of each P in $\mathcal{T}$ by also storing, with each edge of $\mathcal{S}$, the BFS rank r of its parent, the rank of a child polygon is then $r + 1$. Details will appear in the full version of this paper.

Analysis: The algorithm for computing the contour map $\mathcal{M}$ uses $O(\mathrm{Sort}(|\mathcal{M}|))$ I/Os: First it scans the input triangles to produce the segments of $\mathcal{M}$ and invokes the practically efficient connected component algorithm of Arge et al. [7]

that uses $O(\text{Sort}(|\mathcal{M}|))$ I/Os under the assumption that the segments in $\mathcal{M}$ intersecting any horizontal line fit in memory. Then it sorts the labeled segments using another $O(\text{Sort}(|\mathcal{M}|))$ I/Os. Finally, it scans the segments to sort each polygon in internal memory, utilizing that the segments in any polygon P in $\mathcal{M}$ fits in memory.

Also the algorithm for computing the topology tree $\mathcal{T}$ uses $O(\text{Sort}(|\mathcal{M}|))$: After scanning the segments of $\mathcal{M}$ to produce L, it performs one sort and one scan of L, utilizing the assumption that $\mathcal{S}$ fits in memory. In the full paper we show that augmenting each segment in $\mathcal{M}$ with the BFS number of the polygon P that it belongs to, can also be performed in $O(\text{Sort}(|\mathcal{M}|))$ in a simple way.

4 Simplifying Families of Polygons

In this section we describe our practical and I/O-efficient algorithm for simplifying a set of marked polygons in a family of polygons given an algorithm for simplifying a single polygon P within its polygonal domain $\mathcal{P}$. In essence the problem consist of computing $\mathcal{P}$ for each marked polygon P. We assume the family of polygons is given by a contour map $\mathcal{M}$ represented by a sequence of line segments such that the clockwise ordered segments in each polygon P of $\mathcal{M}$ appear consecutively in the sequence, and a topology tree $\mathcal{T}$ given as a sequence of edges $(P,\hat{P})$ indicating that $\hat{P}$ is the parent of P; all segments in $\mathcal{M}$ of polygon P are augmented with (a label for) P and the BFS number of P in $\mathcal{T}$.

To compute $\mathcal{P}$ for every marked P we need to retrieve the neighbors of P in $\mathcal{M}$. These are exactly the parent, siblings and children of P in $\mathcal{T}$. Once $\mathcal{P}$ and the simplification P' of P has been computed we need to update $\mathcal{M}$ with P'. We describe an I/O-efficient simplification algorithm that allows for retrieving the polygonal domains and updating polygons without spending a constant number of I/Os for each marked polygon. The algorithm simplifies the polygons across different BFS levels of $\mathcal{T}$ in order of increasing level, starting from the root. Within a given level the polygons are simplified in the same order as their parents were simplified. Polygons with the same parent can be simplified in arbitrary (label) order. Below we first describe how to reorder the polygons in $\mathcal{M}$ such that they appear in the order they will be simplified. Then we describe how to simplify $\mathcal{M}$.

Reordering: To reorder the polygons we first compute the *simplification rank* of every polygon P i.e. the rank of P in the simplification order described above. The simplification rank for the root of $\mathcal{T}$ is 0. To compute ranks for the remaining polygons of $\mathcal{T}$, we sort the edges $(P, \hat{P})$ of $\mathcal{T}$ in order of increasing BFS level of P. By scanning through the sorted list of polygons, we then assign simplification ranks to vertices one layer at a time. When processing a given layer we have already determined the ranks of the previous layer and can therefore order the vertices according to the ranks of their parents. After computing the simplification ranks we can easily reorder the polygons in a few sort and scan steps. Details will appear in the full paper.

Simplifying: Consider the sibling polygons $P_1, P_2 \dots P_k$ in $\mathcal{M}$ all sharing the same parent P in $\mathcal{T}$. The polygonal domains of these sibling polygons all share the polygons $P, P_1, P_2 \dots P_k$. We will refer to these shared polygons as the *open polygonal domain* of P and denote them $\mathcal{P}_{open}(P)$. It is easily seen that $\mathcal{P}$ for P_i where $i = 1 \dots k$ is equal to $\mathcal{P}_{open}(P)$ together with the children of P_i.

We now traverse the polygons of $\mathcal{M}$ in the order specified by their simplification ranks, and refer to P as an *unfinished polygon* if we have visited P but not yet visited all the children of P. During the traversal we will maintain a queue Q containing an open polygonal domain for every unfinished polygon. The algorithm handles each marked polygon P as follows; if P is the root of $\mathcal{T}$ then $\mathcal{P}$ simply corresponds to the children of P which are at the front of $\mathcal{M}$. Given $\mathcal{P}$ we invoke the polygon simplification algorithm to get P'. Finally, we put $\mathcal{P}_{open}(P')$ at the back of Q. If P is not the root of $\mathcal{T}$, it will be contained in the open polygonal domain $\mathcal{P}_{open}(\hat{P})$. Since $\hat{P}$ is the unfinished polygon with lowest simplification rank, $\mathcal{P}_{open}(\hat{P})$ will be the front element of Q. If P is the first among its siblings to be visited, we retrieve $\mathcal{P}_{open}(\hat{P})$ from Q, otherwise it has already been retrieved and is available in memory. To get $\mathcal{P}$, we then retrieve the children of P from $\mathcal{M}$ and combine them with $\mathcal{P}_{open}(\hat{P})$ (if P is a leaf then $\mathcal{P} = \mathcal{P}_{open}(\hat{P})$). Finally, we invoke the polygon simplification algorithm on P and $\mathcal{P}$ to get P' and put the open polygonal domain of P' at the back of Q. It is easy to see that this algorithm simplifies $\mathcal{M}$, details will appear in the full version of this paper.

Analysis: Both reordering and simplifying is done with a constant number of sorts and scans of $\mathcal{M}$ and therefore require $O(\text{Sort}(|\mathcal{M}|))$ I/Os.

5 Internal Simplification Algorithm

In this section we describe our simplification algorithm, which given a single polygon P along with its polygonal domain $\mathcal{P}$ outputs a valid simplification P' of P.

Simplifying P: We first show how to compute a simplification Q^* of a path or polygon Q such that Q^* is homotopic to Q in $\mathcal{P}$, based on the Douglas-Peucker algorithm [12]. The recursive algorithm is quite simple. Given Q and a sub-path Q_{ij} for $i < j + 1$ to simplify, we find the vertex p_k in Q_{ij} maximizing the error $\varepsilon(i, j)$. Then we insert vertex p_k in the output path Q^* and recurse on Q_{ik} and Q_{kj}. When the error is sufficiently small, i.e. $\varepsilon(i, j) < \varepsilon_{xy}$, we check if the segment of Q^* is homotopic to P_{ij}. If this is the case the recursion stops. By construction every segment $p_i p_j$ of Q^* is homotopic to Q_{ij}. This implies that Q^* is strongly homotopic to Q, which again implies that Q^* and Q are homotopic.

When the input to the algorithm above is a polygon P the output is also a polygon P^*. However, even though P^* is homotopic to P, it is not necessarily a valid simplification since P^* may not be simple. Thus after using the algorithm above we may need to turn P^* into a simple polygon P'. This is done by finding all segments s of P^* that intersect and add more vertices from P to those

segments using the same algorithm as above. Once this has been done we check for new intersections and keep doing this until no more intersection are found. Details will appear in the full paper.

Checking segment-sub-path homotopy: To check if a segment p_ip_j is homotopic to P_{ij} in the above algorithm we need to be able to navigate the space around $\mathcal{P}$. Since $\mathcal{P}$ is given as a set of ordered simple polygons, we can efficiently and easily compute its trapezoidal decomposition $\mathcal{D}$ using a simple sweep line algorithm on the segments of $\mathcal{P}$. To check homotopy we use the ideas of Cabello *et al.* [8] but arrive at a simpler algorithm by taking advantage of $\mathcal{D}$. We define the *trapezoidal sequence* $t(Q)$ of a path Q to be the sequence of trapezoids traversed by Q, sorted in the order of traversal. Using an argument similar to the ones used in [8, 17] it is easy to show that if two paths have the same trapezoidal sequence then they are homotopic. Furthermore, in the full paper we show that if $t(Q)$ contains the subsequence $tt't$ for trapezoids $t, t' \in \mathcal{D}$ then this subsequence can be replaced by t without affecting Q's homotopic relationship to any other path; we call this a *contraction* of $t(Q)$. By repeatedly performing contractions on the sequence $t(Q)$ until no more contractions are possible we get a new sequence $t_c(Q)$, called the *canonical trapezoidal sequence* of Q. Q and Q' are homotopic if and only if $t_c(Q) = t_c(Q')$ [8, 17].

Our algorithm for checking if two paths/polygons are homotopic simply computes and compares their canonical trapezoidal sequences. Note however that, for a path Q, the sizes of $t(Q')$ and $t_c(Q')$ are not necessarily linear in the size of the decomposition $\mathcal{D}$. In our case, we are interested in checking an instance of the strong homotopy condition, i.e. we want to check if a segment $s = p_ip_j$ is homotopic to $Q_{i,j}$. Since s is a line segment, we know that $t_c(s) = t(s)$, and we can thus simply traverse the trapezoids along in Q_i, tracing out $t_c(Q_{ij})$, and check that s intersects the same trapezoids as we go along, we do not need to precompute and store) $t(Q_{ij})$.

Analysis We assume that $\mathcal{P}$ and Q fit in memory. Since the size of $\mathcal{D}$ is linear in the size of $\mathcal{P}$, it also fits in memory and the entire algorithm thus uses no I/Os. The I/Os needed to bring $\mathcal{P}$ and Q into memory were accounted for in the previous section.

6 Experiments

In this section we describe the experiments performed to verify that our algorithm for computing and simplifying a contour map $\mathcal{M}$ performs well in practice.

Implementation: We implemented our algorithm using the TPIE[1]: environment for efficient implementation of I/O-efficient algorithms, while taking care to handle all degeneracies (e.g. contours with height equal to vertices of $\mathbf{M}$, contour points with the same $x-$ or y-coordinate, and the existence of a single boundary contour). The implementation takes an input TIN $\mathbf{M}$ along with parameters ε_{xy} and ε_z, and Δ, and produces a simplified contour map $\mathcal{M}$ with equi-spaced contours at distance Δ.

ε_{xy} in m.	0.2	0.5	1	2	3	5	10	15	20	25	50
Output points (%)	40.4	23.7	15.2	10.2	8.8	7.9	7.6	7.6	7.6	7.6	7.6
ε_z points	0.8	5.0	13.9	33.5	46.0	59.3	71.7	76.3	78.8	80.4	84.1
ε_{xy} points	99.2	95.0	86.1	66.5	54.0	40.7	28.3	23.7	21.2	19.6	15.9

Table 1. Results for Funen with different ε_{xy} thresholds ($\varepsilon_z = 0.2m$ and with $\Delta = 0.5m$).

We implemented one major internal improvement compared to algorithm described in Section 5, which results in a speed-up of an order of magnitude: As described in Section 5 we simplify a polygon P by constructing a trapezoidal decomposition of its entire polygonal domain $\mathcal{P}$. In practice, some polygons are very large and have many relatively small child polygons. In this case, even though a child polygon is small, its polygonal domain (and therefore also its trapezoidal decompositions) will include the large parent together with its siblings. However, it is easy to realize that for a polygon P it is only the subset of $\mathcal{P}$ within the bounding box of P that can constrain its simplification, and line segments outside the bounding box can be ignored when constructing the trapezoidal decomposition. We incorporate this observation into our implementation by building an internal memory R-tree [15] for each polygon P in the open polygonal domain $\mathcal{P}_{open}(\hat{P})$. These R-trees are constructed when loading large open polygonal domain into memory. To retrieve the bounding box of a given polygon P in $\mathcal{P}_{open}(\hat{P})$, we query the R-trees of its siblings and its parent, and retrieve the children of P as previously.

Data and Setup: All our experiments were performed on a machine with an 8-core Intel Xenon CPU running at 3.2GHz and 12GB of RAM. For our experiments we used a terrain model for the entire country of Denmark constructed from detailed LIDAR measurements (the data was generously provided to us by COWI A/S). The model is a $2m$ grid model giving the terrain height for every $2m \cdot 2m$ in the entire country, which amounts to roughly 12.4 billion grid cells. From this grid model we built a TIN by triangulating the grid cell center points. Before triangulating and performing our experiments, we used the concept of topological persistence [13] to compute the depth of depressions in the model. This can be done I/O-efficiently using an algorithm by Agarwal et. al [4]. For depressions that are not deeper than Δ it is coincidental whether the depression results in a contour or not. In case a contour is created it appears noisy and spurious in the contour map. For our experiments, we therefore raised depressions with a depth less than Δ. We removed small peaks similarly by simply inverting terrain heights. Our results show that this significantly reduces the size of the non-simplified contour map. Details will appear in the full paper.

Experimental Results: In all our experiments we generate contour maps with $\Delta = 0.5m$, and since the LIDAR measurements on which the terrain model of Denmark is based have a height accuracy of roughly $0.2m$, we used $\varepsilon_z = 0.2m$ in the experiments. In order to determine a good value of ε_{xy} we first performed experiments on a subset of the Denmark dataset consisting of $844,554,140$ grid cells and covering the island of Funen. Below we first describe the results of

Dataset	Funen	Denmark
Input points	365,641,479	4,046,176,743
Contours	636,973	16,581,989
Constraint factor	3	3
Running time (hours)	1.5	39
Output points (% of input points)	7.9	8.2
ε_z points (% of output points)	59.3	57.8
ε_{xy} points (% of output points)	40.7	42.2
Total number of intersections	38,992	585,813
Contours with intersections (% of input contours)	2.4	1.1

Table 2. Results for Funen and Denmark with $\Delta = 0.5m$, $\varepsilon_z = 0.2m$ and $\varepsilon_{xy} = 5m$.

these experiments and then we describe the result of the experiments on the entire Denmark dataset. When discussing our results, we will divide the number of contour segment points in the simplified contour map (output points) into ε_z *points* and ε_{xy} *points*. These are the points that were not removed due to the constraint contours and the constraints of our polygon simplification algorithm (e.g. ε_{xy}), respectively.

Funen dataset: The non-simplified contour map generated from the triangulation of Funen consists of $636,973$ contours with $365,641,479$ points (not counting constraint contours). The results of our test runs are given in Table 1. The number of output points is given as a percentage of the number of points in the non-simplified contour map (not counting constraint contours). From the table it can be seen that the number of output points drops significantly as ε_{xy} is increased from $0.2m$ up to $5m$. However, for values larger than $5m$ the effect on output size of increasing ε_{xy} diminishes. This is most likely linked with high percentage of ε_z points in the output e.g. for $\varepsilon_{xy} = 10m$ we have that 71.7% of the output points are ε_z points (and increasing ε_{xy} will not have an effect on these).

Denmark dataset: When simplifying the contour map of the entire Denmark dataset we chose $\varepsilon_{xy} = 5m$, since our test runs on Funen had shown that increasing ε_{xy} further would not lead to a significant reduction in output points. Table 2 gives the results of simplifying the contour map of Denmark. The non-simplified contour map consists of $4,046,176,743$ points on $16,581,989$ contours. Adding constraint contours increases the contour map size with a factor 3 (the *constraint factor*) both in terms of points and contours. In total it took 39 hours to generate and simplify the contour map and the resulting simplified contour map contained 8.2% of the points in the non-simplified contour map (not counting constraint contours). Since 57.8% of the output points were ε_z points, it is unlikely that increasing ε_{xy} would reduce the size of the simplified contour map significantly. This corresponds to our observations on the Funen dataset. Table 2 also contains statistics on the number of self-intersections removed after the simplification (as discussed in Section 5); both the actual number of intersections and the percentage of the contours with self-intersections are given. As it can be seen these numbers are relatively small and their removal does not contribute significantly to the running time. The largest contour consisted of $8,924,584$

vertices while the largest family consisted of $19,145,568$ vertices, which easily fit in memory.

References

1. TPIE - Templated Portable I/O-Environment. http://madalgo.au.dk/tpie.
2. P. Agarwal, L. Arge, T. Mølhave, and B. Sadri. I/O-efficient algorithms for computing contours on a terrain. In *Proc. Symposium on Computational Geometry*, pages 129–138, 2008.
3. P. K. Agarwal, L. Arge, T. M. Murali, K. Varadarajan, and J. S. Vitter. I/O-efficient algorithms for contour line extraction and planar graph blocking. In *Proc. ACM-SIAM Symposium on Discrete Algorithms*, pages 117–126, 1998.
4. P. K. Agarwal, L. Arge, and K. Yi. I/O-efficient batched union-find and its applications to terrain analysis. *ACM Trans. Algorithms*, 7(1):11:1–11:21, Dec. 2010.
5. A. Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988.
6. L. Arge. External memory data structures. In J. Abello, P. M. Pardalos, and M. G. C. Resende, editors, *Handbook of Massive Data Sets*, pages 313–358. 2002.
7. L. Arge, K. Larsen, T. Mølhave, and F. van Walderveen. Cleaning massive sonar point clouds. In *Proc ACM SIGSPATIAL International Symposium on Advances in Geographic Information Systems*, pages 152–161, 2010.
8. S. Cabello, Y. Liu, A. Mantler, and J. Snoeyink. Testing homotopy for paths in the plane. In *Proc. Symposium on Computational geometry*, pages 160–169, 2002.
9. H. Carr, J. Snoeyink, and M. van de Panne. Flexible isosurfaces: Simplifying and displaying scalar topology using the contour tree. *Computational Geometry*, pages 42–58, 2010. Special Issue on the 14th Annual Fall Workshop.
10. A. Danner, T. Mølhave, K. Yi, P. Agarwal, L. Arge, and H. Mitasova. TerraStream: From elevation data to watershed hierarchies. In *Proc. ACM International Symposium on Advances in Geographic Information Systems*, pages 28:1–28:8, 2007.
11. M. de Berg, M. van Kreveld, and S. Schirra. A new approach to subdivision simplification. In *Proc. 12th Internat. Sympos. Comput.-Assist. Cartog.*, pages 79–88, 1995.
12. D. Douglas and T. Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature, 1973.
13. H. Edelsbrunner, D. Letscher, and A. Zomorodian. Topological persistence and simplification. In *Proc. IEEE Symposium on Foundations of Computer Science*, pages 454–463, 2000.
14. M. Garland and P. Heckbert. Surface simplification using quadric error metrics. In *Proc. Computer graphics and interactive techniques*, pages 209–216, 1997.
15. A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. SIGMOD International Conference on Management of Data*, pages 47–57, 1984.
16. P. S. Heckbert and M. Garland. Survey of polygonal surface simplification algorithms. Technical report, CS Dept., Carnegie Mellon U. to appear.
17. J. Hershberger and J. Snoeyink. Computing minimum length paths of a given homotopy class. *Comput. Geom. Theory Appl.*, pages 63–97, 1994.
18. A. Saalfeld. Topologically consistent line simplification with the douglas peucker algorithm. In *Geographic Information Science*, 1999.
19. J. Vitter. External memory algorithms and data structures: Dealing with MASSIVE data. *ACM Computing Surveys*, pages 209–271, 2001.