Exact and Approximate Computations of Watersheds on Triangulated Terrains

Mark de Berg
Department of Computer Science
TU Eindhoven, the Netherlands
mdberg@win.tue.nl

Constantinos Tsirogiannis*
Department of Computer Science
TU Eindhoven, the Netherlands
ctsirogi@win.tue.nl

ABSTRACT

The natural way of modeling water flow on a triangulated terrain is to make the fundamental assumption that water follows the direction of steepest descent (DSD). However, computing watersheds and other flow-related structures according to the DSD model in an exact manner is difficult: the DSD model implies that water does not necessarily follow terrain edges, which makes designing exact algorithms difficult and causes robustness problems when implementing them. As a result, existing software implementations for computing watersheds are inexact: they either assume a simplified flow model or they perform computations using inexact arithmetic, which leads to inexact and sometimes inconsistent results. We perform a detailed study of various issues concerning the exact or approximate computation of watersheds according to the DSD model. Our main contributions are the following.

- We provide the first implementation that computes watersheds on triangulated terrains following strictly the DSD model and using exact arithmetic, and we experimentally investigate its computational cost. Our experiments show that the algorithm cannot handle large data sets effectively, due to the bit-sizes needed in the exact computations and the computation of an intermediate structure called the strip map.
- Using our exact algorithm as a point of reference, we evaluate the quality of several existing heuristics for computing watersheds. We also investigate hybrid methods, which use heuristics in a first phase of the algorithm and exact computation in a second phase. The hybrid methods are almost as fast as the heuristics, but give significantly more accurate results.
- We describe and theoretically analyze a new exact algorithm for computing watersheds, which avoids the computation of the strip map.

*Current affiliation: MADALGO, University of Aarhus, Denmark

ACM SIGSPATIAL GIS '11, November 1-4, 2011. Chicago, IL, USA

Categories and Subject Descriptors

F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Geometrical problems and computations*

General Terms

Algorithms, Performance, Experimentation

1. INTRODUCTION

Analyzing the flow of water on mountainous terrains is important for flood prediction, erosion simulation, the study of changes in the shape of glaciers, and other problems. Hence, the use of digital terrain representations to support flow analysis has received ample attention in hydrology and GIS. The most popular representations are the so-called *digital elevation model* (DEM) and the *triangulated irregular network* (TIN). A DEM represents a terrain by a regular grid on the xy-plane in which each grid cell is assigned an elevation; a TIN represents a terrain by a triangulation in the xy-plane in which each vertex is assigned an elevation. Both representations have advantages and disadvantages concerning for example accuracy, memory usage, and ease of implementation of algorithms. In this paper we restrict our attention to TIN representations.

When it comes to flow modeling TINs have an important property: the surface they define is continuous (unlike DEMs, which define a non-continuous surface). As a result, the flow of water on a TIN can be modeled in a very natural way using the fundamental assumption that water follows the direction of steepest descent (DSD). The DSD model is purely geometrical and as such does not completely capture the behavior on real terrains, where aspects like soil type and land cover also play a role. Nevertheless, purely geometric flow models are considered to be a useful abstraction for performing flow analysis and the DSD model is the most natural geometric flow model. Thus efficient and accurate algorithms for computing flow-related structures on a given TIN $\mathcal{T}$ according to the DSD model are highly desirable.

The most basic algorithmic question relating to flow is to compute, for a given point p on $\mathcal{T}$, where the water from p drains to. In other words, we want to compute the *trickle path* of p: the path of steepest descent starting at p and ending at the local minimum to which the water from p drains. Even this basic problem is not as easy as it seems. First of all, the DSD model does not specify how water flows across flat (horizontal) areas or when the DSD is not unique. How to deal with this is an important research topic in itself, which

is complementary to the topic that we address in this paper. Second, even when the DSD is unique everywhere it is not easy to compute trickle paths in an exact and robust manner. The difficulty lies in the fact that the trickle path does not necessarily follow edges of the TIN—it sometimes crosses through triangle interiors. This can cause robustness problems during the computations: the use of standard, fixed-precision arithmetic may lead to incorrect results. Note that a small error upstream in a flow path may cause a very large deviation downstream.

Another important algorithmic question—the one that is the main focus of our paper—is the computation of watersheds. The *watershed* of a point p on $\mathcal{T}$ is the region consisting of all points whose trickle paths reach p, and the *watershed map* of $\mathcal{T}$ is the subdivision of $\mathcal{T}$ induced by the watersheds of all local minima. Again, the fact that water can flow through triangle interiors makes the exact computation of watersheds difficult, both conceptually and from an implementation point of view. The use of finite-precision computations can even lead to inconsistent results such as a watershed containing no, or more than one, local minimum [12] and, if one is not very careful, to program crashes.

Previous work.

There exist many algorithms that compute the watershed map of a given TIN; an overview can be found in the paper by Čomić *et al.* [5]. Due to problems mentioned above, most of these algorithms do not follow the DSD model exactly. Instead, they often only consider flow along edges of the TIN. Thus, they restrict the flow to a discrete network rather than considering the whole TIN surface. One example of this approach is the popular algorithm of Mangan and Whitaker [13]; other examples are the methods of Takahashi *et al.* [20] and of Vincent and Soile [22]. Restricting flow to the TIN edges makes the computations easier, but it does not lead to exact results. For example, since these methods restrict their attention to the TIN edges, they assign each triangle to a single watershed even when the DSD model implies that the triangle interior is crossed by a watershed boundary. (Instead of considering the TIN edges, some methods [17] consider a network whose nodes are the triangle centers leading to similar problems.) Because local errors in the flow can have global effects, they can even assign triangles to the wrong watershed when no watershed boundary crosses them. From now on we refer to flow models (or algorithms) that do not strictly follow the DSD assumption as *inexact flow models* (or algorithms). The DSD model will sometimes be referred to as the *exact flow model*.

De Berg *et al.* [1] were the first to perform a theoretical study of the DSD model. In particular, they studied the complexity of various structures in the DSD model on a terrain with n triangles. For example, they showed that in the worst case a single trickle path may cross the same TIN edge several times. In fact, in a worst-case (and very unrealistic) scenario, a single trickle path can cross many edges many times, leading to a worst-case complexity of $\Theta(n^2)$ for a single trickle path. They also studied the worst-case complexity of watersheds and *river networks* [1]. Recently it was formally proved that this high complexity does not occur in realistic TINs: when the triangles do not have very small angles and some other mild conditions are satisfied, then the worst-case complexity of a trickle path is only $\Theta(n)$ [2].

The DSD model was further investigated by Yu *et al.* [23]. They introduced a key structure for computing watersheds, the so-called *strip map*, which is defined as follows. From each TIN vertex, expand all paths of locally steepest ascent and descent. These paths together partition the TIN surface into *strips*, which together form the strip map of the TIN. The "bottom" of each strip is a part of a valley edge. (A *valley edge*, or *channel*, is an edge such that the DSD in the interior of both of its incident triangles points towards this edge.) The crucial property of the strips is that the trickle path of any point inside a strip leads to the valley edge at its bottom and, hence, drains to the same local minimum. Thus any watershed in the watershed map is the union of one or more strips, and the watershed map is easily extracted from the strip map. Unfortunately, the complexity of the strip map is quite high. McAllister [14] presented an algorithm for computing the watershed map on a TIN that does not compute the complete strip map; instead he only considers paths of steepest descent/ascent from saddle vertices. Unfortunately this may lead to incomplete results [2, 10].

When it comes to actual implementations of watershed algorithms according to the DSD model, we are only aware of implementations of McAllister's algorithm [12, 14, 15]. The most detailed discussion of their implementation is given by Liu and Snoeyink [12]. They implement the algorithm using finite-precision arithmetic, and they observe that this may give inconsistent results (like watersheds that do not contain exactly one local minimum). They also discuss from a theoretical point of view some issues related to the use of *exact arithmetic* when implementing flow computations according to the DSD model. In particular, they consider the number of bits needed to exactly compute the intersection points of a trickle path with the TIN edges that it crosses. They show that this number grows linearly with the number of edges crossed, potentially leading to large bit-sizes in the computations. This leads to the following questions: Is it feasible in practice to compute trickle paths and watersheds on a TIN *exactly* according to the DSD model, using an algorithm based on the complete strip map and using exact arithmetic? What are the bit-sizes needed in the computations? And if the exact algorithm turns out to be impractical, then which of the inexact methods approximates the exact results best?

Our contribution.

We provide the first complete and exact implementation that computes watersheds on TINs according to the DSD model. Our implementation is based on the computational-geometry algorithms library CGAL [4], which provides an easy way to perform the computations using exact arithmetic. We experimentally investigate the performance of our implementation on several real-world data sets. In particular we measure the bit-sizes needed for the exact computations and the resulting memory usage of the algorithm. As it turns out, the large bit-sizes are not only a problem in theory, but also in practice. Moreover, the computation of the complete strip map is a significant bottleneck of the algorithm: the strip map has a much higher complexity then the final watershed map computed from it (and this problem is aggravated by the large bit-size problem). Hence, our implementation is impractical for large data sets.

Our exact algorithm also provides us with the opportunity to study the quality of the output of inexact (but hopefully more efficient) algorithms, since it can serve as a point of reference. Thus we implement three different inexact algo-

rithms from the literature—the algorithm by Mangan and Whitaker [13], the one by Takahashi *et al.* [20], and the one by Vincent and Soile [22]—and we compare their output to the output of our exact algorithm. All algorithms are quite efficient, but the algorithm by Mangan and Whitaker [13] turns out to produce the highest-quality results. We also propose and investigate hybrid methods, which are based on the above-mentioned heuristics, but perform part of the computation in an exact manner. These hybrid approaches turn out to be almost as fast as the heuristics, while giving significantly more accurate results.

As mentioned above, the explicit computation of the complete strip map is a major bottleneck. In a recent paper [3] we showed how to compute various flow-related structures in an implicit manner, thus greatly speeding up the worst-case theoretical running times. Based on ideas from that paper we describe an algorithm to compute the exact watershed map without computing the complete strip map as an intermediate structure. We perform a theoretical analysis of the running time of our algorithm as a function of the input size n and the output size k (that is, the size of the watershed map). The analysis shows that if the watershed map has small complexity—which is the case in practice, as our experiments have shown—then its theoretical running time is superior to the running time of algorithms that compute the complete strip map.

2. DESCRIPTION OF IMPLEMENTATION AND EXPERIMENT SETTINGS

In this section we describe our exact implementation for computing watersheds according to the DSD model, and the set-up for our experiments. Our software provides algorithms for computing flow paths, watersheds, strip maps, river networks and surface networks on TINs.

2.1 Implementation using CGAL

As mentioned in the introduction, Liu and Snoeyink [12] showed that fixed-precision arithmetic is insufficient for representing exactly the coordinates of the intersection points between flow paths and terrain edges. Indeed, as we follow a trickle path, it may cross a sequence of distinct transfluent edges $\{e_1, e_2, \ldots, e_k\}$—transfluent edges are edges that are neither valley edges nor ridge edges—and each crossing can increase the bit-size of the intersection point. More precisely, if the coordinates of the starting point of the trickle path and the coordinates of each vertex of $\mathcal{T}$ are numerical values of constant bit-size, then a single coordinate of the intersection point between this path and edge e_i can be a rational with bit-size $\Theta(i)$. Hence, using doubles or any other fixed-precision representation is insufficient to represent these coordinates exactly. The resulting inconsistencies may cause severe problems. Indeed, running our implementation using fixed-precision arithmetic (e.g. doubles) often results in program crashes. We thus need *exact arithmetic*.

We therefore base our algorithm on the *Computational Geometry Algorithms Library* (CGAL). CGAL is an open source software library in C++ that provides a wide range of geometric algorithms and data structures. This includes both basic subroutines (computing intersection points of geometric objects, distances, etcetera) as well as more involved algorithms and data structures (for convex hulls, Voronoi diagrams, point location, and so on). Two major advantages of CGAL are the flexibility that comes from the use of generic programming, and computational robustness through the use of exact arithmetic.

CGAL follows the generic programming paradigm, making heavy use of templates that provide the opportunity to define a class using different parameter-types for its representation. For example, an important notion in CGAL are the *geometric kernels*: classes that provide the definitions of essential geometric objects and functions applied to these objects. A kernel is a template class that takes as a parameter the number type that represents the encapsulated geometric objects and functions. Hence, by instantiating the Cartesian kernel of CGAL using the `double` C++ built-in type as a template parameter, the encapsulated object-types of the kernel such as points or segments are represented with cartesian coordinates of double floating point precision. It is the user then who chooses which number type to use and make the trade-off between exact (but sometimes slow) and fast (but not exact) computations.

In our implementation we use the GNU number type `Gmpq` [9] which is appropriate for computations between rationals of arbitrary precision. In fact, we use the more refined CGAL number type `CGAL::Lazy_exact_nt<Gmpq>` [11]. The latter number type uses an *algebraic filtering technique*: it uses fixed-precision arithmetic whenever it is possible to avoid costly exact computations, but it switches to exact arithmetic when fixed precision is not enough to determine correctly the output of some predicate. This filtering technique makes it possible to construct topologically correct flow paths (and derived drainage structures) using smaller bit-size than the bit-size of their exact representation. At the end of the computation, it is still possible to extract the exact coordinates. The watersheds in the output are represented as graph type objects that follow the standards of the Boost Graph Library [8].

The algorithm we used for computing the watershed map of an input TIN $\mathcal{T}$ is inspired by the work of Yu *et al.* [23]. First we compute the strip map of $\mathcal{T}$ by expanding from each vertex $v \in \mathcal{T}$ the path of locally steepest descent and the path of locally steepest ascent. All points in the interior of the same strip have their trickle paths overlapping at the lower strip boundary and thus drain to the same local minimum. Hence, we label the subfaces of each strip with the local minimum to which it drains. Then we delete from the strip map the parts of the constructed paths that do not overlap with terrain edges and that are also adjacent to strips labeled with the same local minimum. The resulting subdivision is then the watershed map of $\mathcal{T}$.

2.2 Experimental Set-Up

The experiments were conducted using an Intel i5 four-core CPU where each core is a 3.20 GHz processor. However, as there is no parallelization in the algorithms that we implemented, the computations were handled each time by only one of the processors. The main memory of this system is 3.4 Gigabytes. Our code runs on a Linux Ubuntu operating system version 10.10 using the GNU g++ version compiler 4.4.4. Our implementation is compatible with version 3.8 of CGAL.

The TINs we have used in our experiments were constructed using data obtained from the U.S. Geological Survey (USGS) online server U.S. National Elevation Dataset[21]. Each data set is a DEM in the `ADF` file format and consists of a regular

3,612 × 3,612 grid at 30m resolution, where for each grid cell the elevation is given as a 4-byte floating point value. Each data set represents a certain region in the United States. The names that we use to refer to each of these data sets, as well as their elevation ranges and the geographical areas that they model, are summarized in Table 1.

Table 1: The data sets used in the experiments.

name	modeled region	elevation range
`duchesne`	Duchesne (UT)	[1412 m, 3737 m]
`nazareth`	Nazareth (TX)	[1051 m, 1349 m]
`parnassus`	Mount Parnassus (CO)	[1551 m, 4351 m]

To construct each input TIN, we selected 1,201 × 1,201 grid cells from the central region of the DEM. We chose to model this restricted region of the DEM to reduce undersampling effects (as compared to modeling the whole region), while still modeling a reasonably "interesting" region. From this region we then sampled n points (centers of grid cells) that constitute the vertex set of the TIN, using a greedy method that attempts to minimize the elevation difference between the TIN surface and the remaining points [16]. The value of n depends on the experiment. In case neighbouring vertices have the same elevation, we apply a small perturbation, so that the flow is always well defined and the issue of how to deal with flat areas does not influence our results.

3. THE COMPLEXITY OF FLOW STRUCTURES

In this section we present our experimental investigation of the complexity of several flow structures on TINs. Our goal is to provide an evaluation of both the combinatorial complexity of TIN drainage structures and their total bit-size complexity when the computations are done exactly.

3.1 The Complexity Of Flow Paths

In the first set of experiments we measure the combinatorial complexity and the bit-size of individual flow paths that are expanded from the vertices of a TIN. The results of such experiments can provide insight on how the complexity of more complicated drainage structures grows as the input size increases. Indeed, the strip map of a TIN, and therefore also its watershed map, are computed by expanding paths of steepest gradient from every vertex of the TIN. If the average bit-size of individual flow paths rises considerably as the size of the input increases then this poses a significant restriction on the size of the TINs for which we can compute the strip map and the watershed map using exact arithmetic. Next we provide a detailed description of the first set of our experiments.

Consider a vertex v of a TIN $\mathcal{T}$. A path of steepest descent or steepest ascent on $\mathcal{T}$ consists of line segments, which either lie in the interior of a TIN triangle or are (a part of) a TIN edge. The number of line segments the path consists of is called its *combinatorial complexity*. The total number of bytes needed to represent the coordinates of the vertices of the path—note that these are not necessarily TIN vertices—is called the *bit-size* of the path. For each vertex v of $\mathcal{T}$ we construct two paths, namely the paths of steepest descent and the path of steepest ascent from v. We measure the following quantities as a function of n, the number of TIN vertices.

$cc(n)$: the average combinatorial complexity of the paths;
$bs(n)$: the average bit-size of the paths.

Note that the paths that we consider can overlap significantly. For example, when the path of steepest descent from a vertex v passes through a vertex w (usually after first following a valley edge) then from that point on it will coincide with the steepest-decent path from w. We therefore also measure the average *exclusive combinatorial complexity* and the average *exclusive bit-size* of the paths, denoted by $cc^*(n)$ and $bs^*(n)$, respectively. These are defined in the same way as $cc(n)$ and $bs(n)$, except that we only consider the part of each path up to the first encountered vertex (after the starting point). Studying the complexity of this part of the path alone is important for estimating the size of the strip map; trickle paths and up-paths of vertices may overlap to a large extent and if we would sum the total complexity of each path individually we would make an overestimation on the size of the entire structure.

We have computed the described complexity values for flow paths on TINs of different values of n but constructed from the same DEM data set. More specifically, from the `nazareth` DEM data set we have constructed 50 TIN instances, with $n = 1000k$ and k ranging from 1 to 50. The results of these experiments are presented in Figs. 1 and 2.

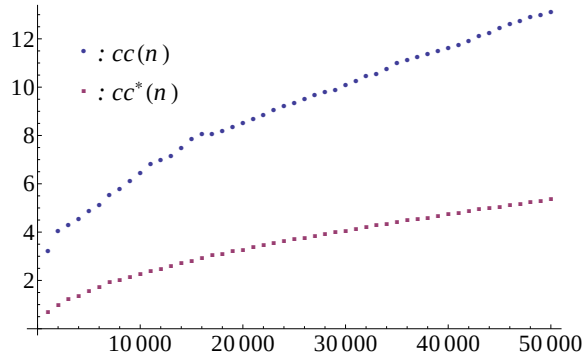

Figure 1: The average combinatorial complexity $cc(n)$ and average exclusive combinatorial complexity $cc^*(n)$ of the paths of steepest ascent and descent on the `nazareth` data set, as a function of the number of TIN vertices.

Discussion.

As can be seen in Fig. 1, the total (exclusive) combinatorial complexity of the paths increases with n (not surprisingly), though not linearly. Moreover, the average exclusive complexity is rather small: most paths quickly merge with other paths. The maximum combinatorial complexity of a single path (not shown in the figure) for the terrain with 50,000 vertices was 66. The average bit-sizes grow faster than the combinatorial complexity of the paths, indicating that the bit-sizes of the individual vertices on the paths are increasing. For a TIN of 50,000 vertices, the average exclusive bit-size of a path is close to 1.3 kilobytes while the average exclusive combinatorial complexity is roughly 5. Since a path with five segments has six vertices, each having an x-, a y-, and a z-coordinate, this means we need about 72

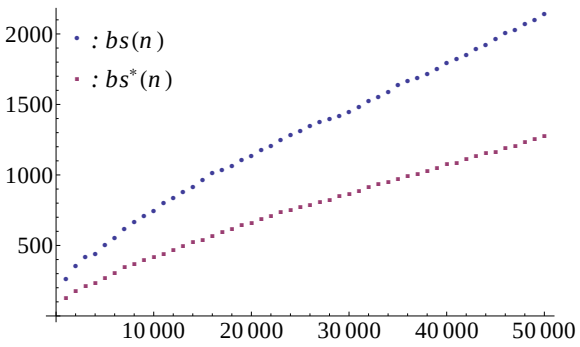

Figure 2: The average bit-size $bs(n)$ and average exclusive bit-size $bs^*(n)$ of the paths of steepest ascent and descent on the `nazareth` data set, as a function of the number of TIN vertices.

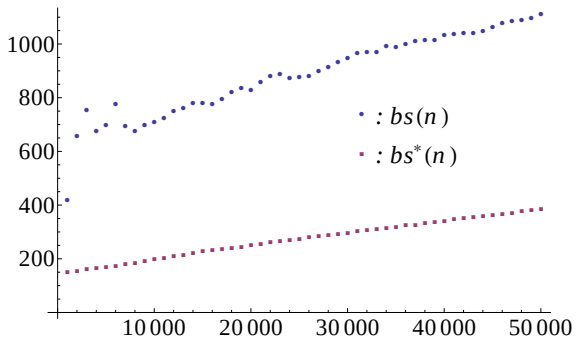

Figure 3: The average bit-size $bs(n)$ and average exclusive bit-size $bs^*(n)$ of the paths of steepest ascent and descent on the `parnassus` data set, as a function of the number of TIN vertices.

bytes per coordinate. The maximum number of bytes for a single coordinate (not shown in the figure) was even 708 bytes, showing that very large bit sizes really arise in practice when doing exact computations. To represent all the paths together, we need approximately 130 MB, which is more than 100 times the 1.2 MB that would be needed for the TIN vertices if we used double precision floating point arithmetic. Since we have $2n$ paths, and the average exclusive complexity is 5, a factor 10 of the blow-up is caused by increasing combinatorial complexity. Another factor 10 is caused by the increase in the bit-size of the coordinates, which is thus a serious problem in practice. We will refer to these results later on, when we will experimentally evaluate the bit-size and the combinatorial complexity of larger drainage structures.

Note that the goal of our experiments is not to provide a detailed analysis of the precise dependency of the complexity of flow structures on n, or to give an extensive evaluation of the complexity for landscapes of all kinds of different morphologies. The main goal is to get some insight into the (in)feasibility of computing these structures exactly. Therefore we have chosen a data set which illustrates the potential blow-up well. We have repeated the same experiments for the other data sets mentioned in Section 2.2. For the majority of these data sets, the numbers observed in the measured complexities differ only up to a small constant factor from the values for the `nazareth` data set (with the `nazareth` giving slightly higher complexities). Thus, even though for other data sets the blow-up is smaller, it will still be too costly to use exact computations when the data sets become large. For completeness we give the results for the average bit-size for one more data set, namely the `parnassus` data set–see Figure 3.

3.2 The Complexity of Watersheds, Strip Maps and River Networks

Our next step in the experimental evaluation of the exact DSD flow model is to measure the complexity of more involved drainage structures. In the following set of experiments we measure the combinatorial complexity and the bit-size of watershed maps and river networks on TINs. Recall that to compute the watershed map we have first to construct a more refined intermediate structure, the *strip map*. We already mentioned that the boundaries between incident watersheds in the watershed map is a subset of the paths constituting the strip map. Thus, although the strip map is not by itself a structure that is used in hydrological applications, its complexity provides further insight as to the computational effort that is needed for computing exact watersheds on a TIN. For this reason, in this set of experiments we measure also the combinatorial complexity and the bit-size of the strip map. With the results of these experiments we intend to provide a clear picture of the computational demands of the studied flow model along with the restrictions on the size of input data that can be processed.

For the current set of experiments we have used the same 50 TIN instances derived from the `nazareth` data set that were used for the experiments in Section 3.1. Recall that these TINs consist of from 1000 up to 50,000 vertices and are constructed using a greedy method that tries to minimize the elevation difference between the TIN surface and grid points of the complete data set. For each of these TIN instances we computed the combinatorial complexity and the bit-size of the watershed map, the strip map, and the river network of the TIN. (Here we only count the number of edges, and the bit-size needed to represent their coordinates; the pointers needed for the graph structure (edges, etc) are not taken into account to avoid dependency on implementation details.) The results of this experiments are depicted in Figs. 4 and 5. We also provide the results of the same experiments applied to TINs constructed from the `parnassus` data set–see Fig. 6.

Discussion.

The most striking result from the experiments is the large difference between the complexity (and the bit-size) of the strip map, which is just an intermediate structure, and the complexity (and the bit-size) of the watershed map and river network. Indeed, while the combinatorial complexity of the strip map is much higher than that of the TIN—given the results of the previous subsection, this was to be expected—the complexity of the watershed map is comparable to that of the TIN, and the complexity of the river network is even smaller. When one considers the bit-size instead of the combinatorial complexity, there is a small increase for the watershed map and river network, as compared to the TIN. Still, the bit-size for the watershed map is no more the twice

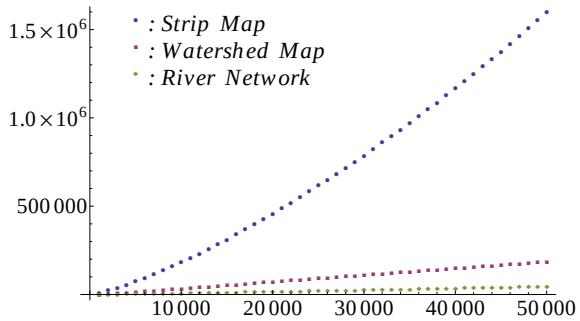

Figure 4: The combinatorial complexity of the strip map, watershed map, and river network for the `nazareth` data set, as a function of the number of TIN vertices.

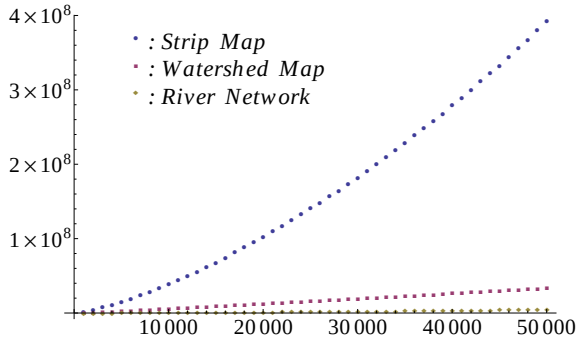

Figure 5: The bit-size of the strip map, watershed map, and river network for the `nazareth` data set, as a function of the number of TIN vertices.

the bit-size needed for the representation of the input (and the river network is even smaller). The bit-size of the strip map, on the other hand, grows to approximately 400 MB for the TIN with 50,000 vertices.(Note that size of the strip map is even higher than the total size of the exclusive paths from the vertices, because the strip map includes all *locally* steepest paths, so that more paths may emanate from each vertex.)

We conclude that the construction of the strip map is a major computational bottleneck when computing watersheds or river networks on a TIN using the DSD model. In fact, explicit computation of the strip map is prohibitive for large TINs. Indeed, one of the reasons that we chose to conduct experiments for the presented range of input sizes is because we experienced problems with the main memory usage when processing larger TINs: the 3.4 GB of RAM of our workstation proved to be barely sufficient for computing the strip map for TINs of 200,000 vertices, let alone the extended time needed for these computations. On the other hand, the watersheds and river networks themselves have reasonable complexity, even when computed exactly and the bit-size of their exact coordinates is taken into account. This raises two questions. First, given the fact that computing watersheds exactly by computing the strip map is infeasible for large data sets, what is the quality of known heuristics for computing these structures? Second, is it possible to

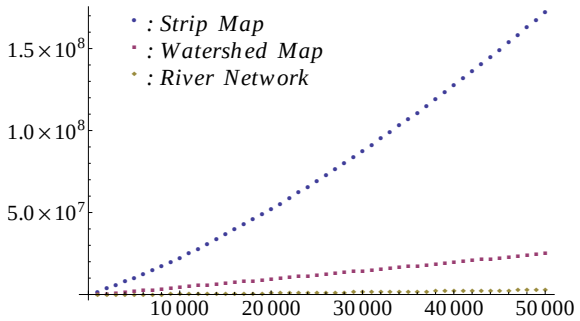

Figure 6: The bit-size of the strip map, watershed map, and river network for the `parnassus` data set, as a function of the number of TIN vertices.

compute watersheds exactly without explicitly computing the strip map as an intermediate structure? We study these two questions in the next sections.

4. QUALITY OF INEXACT FLOW MODELS

The DSD model treats the TIN as a continuous surface, where water can flow along edges as well as across triangle interiors. We saw in the previous section that performing exact computations based on the DSD model is expensive. Discrete flow models, where water is propagated only through a fixed network (such as the TIN edge set), avoid the combinatorial blow-up and bit-size problems and are thus potentially much more efficient. However, the output induced by such a model may contain inconsistencies with the DSD assumption. Thus we would like to assess the quality of the output of these approximate methods.

So far, the only attempts to measure the output quality of a flow model were based either on visual criteria (which is rather subjective), or on a comparison with output induced by a higher resolution data set (which probably tells us more about the quality of the low-resolution data set than about the flow model). The availability of an exact implementation of the DSD model provides us with a point of reference to evaluate the quality of the output of the inexact but more efficient discrete flow models: for small to medium size TINs, we can compare the output of the exact DSD model with the output of the discrete flow models and see then which discrete model gives the best approximation to the DSD model.

We note here that using the exact DSD model as a point of reference does not imply that the DSD model always produces results that represent accurately the flow of water on the real terrain. However, the discrete flow models are all based on the assumption that water follows the direction of steepest descent; they just approximate the computation of flow for efficiency reasons. Hence, it makes sense to compare their output to the exact DSD model.

We have selected three popular methods for computing watersheds on TINs using a discrete flow model. The first method is similar to the algorithm proposed by Mangan and Whitaker [13], and involves expanding approximate paths of steepest descent from TIN vertices following only the edges of the TIN. We refer to this method as the *steepest-neighbour method*. The second method considers a space-sweep mechanism for the computation of watersheds [22]. We call this

method the *simulated immersion* method, following the original term from the literature. The third method considers computing the boundaries of watersheds by expanding descending and ascending paths of TIN edges from the saddle vertices of the terrain [20]. We refer to this method as the *boundary-expansion* method. These three methods are very efficient in terms of memory usage and computational time. For all these methods, watersheds consist of only full triangles, unlike the exact DSD model where watershed boundaries could extend also through triangle interiors. Hence, the combinatorial complexity of the watersheds computed by these methods is the same as the size of the input. Also, the bit-size of the computed watersheds is basically the same as the bit-size of the input TIN; this is because we only need a few extra bits for each vertex, edge and triangle of the TIN to indicate the watershed that it belongs to. Next we provide a more detailed description for each of these heuristics.

- *Steepest-Neighbour Method:* This method first assigns each TIN vertex to a local minimum by expanding an approximate path of steepest descent from this vertex, as follows. Consider a vertex v of the input TIN $\mathcal{T}$ and let $E(v)$ be the set of edges in $\mathcal{T}$ incident to v. The assumption is that water from a vertex v flows through the edge $e \in E(v)$ with the steepest descending slope. Hence, starting from v, we expand a path by picking recursively the steepest descending edge incident to the current path vertex until we encounter a local minimum v_m. Then we assign v, and all the other vertices on this path, to the watershed of v_m. After assigning each vertex of $\mathcal{T}$ to a local minimum according to this procedure, we then determine which triangles are included in the watershed of each minimum. A triangle t is included in the watershed of local minimum v_m if at least two of its incident vertices drain to v_m. If all vertices of t drain to different minima then we assign t to the local minimum that receives water from the vertex of t with the smallest elevation.

- *Simulated Immersion:* This method assigns the TIN vertices to the watersheds of the terrain's local minima using a space-sweep technique. Intuitively, we start from the lowest vertex of the TIN and as we move upwards we construct the terrain watersheds by labeling the vertices that appear on the contours of the TIN. More formally, we first sort the TIN vertices in order of increasing elevation. We then scan the ordered sequence of vertices starting from the lowest vertex. Each time we encounter a local minimum v_m, we assign v_m to its own watershed. Each time we encounter a vertex v that is not a local minimum we look at the neighbouring vertices of v with smaller elevation than v. If all neighbours are assigned to the watershed of a local minimum v_m then we assign also v to v_m. If not all neighbours are assigned to the same minimum, in our implementation of the method, we search for the local minimum to which the largest number of neighbours of v are assigned and then we assign v to the watershed of this minimum.

- *Boundary Expansion:* In this method, the watershed boundaries are outlined by expanding ascending and descending paths from the saddle vertices of the TIN. These paths are sequences of TIN edges that connect saddle vertices with local minima and local maxima on the terrain. Thus, the method is somewhat similar to the exact method of Liu and Snoeyink [12], except that only paths are expanded from saddles and that paths are restricted to TIN edges. For more details on how these paths are expanded in the vicinity of each saddle vertex, the reader may refer to the work of Edelsbrunner *et al.* [7]. The TIN is subdivided by the computed paths into regions where, hopefully, each region contains only one local minimum. The triangles that are contained in such a region form then the watershed of this minimum. However, it is not always guaranteed that each of the delineated regions contains exactly one local minimum. In our implementation, for the case that more than one minima appear within the same region R, we assign a triangle t to the watershed of a minimum $v_m \in R$ if (most of) the vertices of t are closer to v_m than to any other minimum in R. For the case that no minimum exists within R, the triangles within R are not assigned to any watershed.

To compare the quality of these three heuristics, we compute the watershed map of a given TIN four times: once using each heuristic, and once using our exact implementation of the DSD model. Then, for each heuristic, we compute the percentage of the TIN area that is assigned by this heuristic to the same watersheds as in the DSD model. We conducted this experiment using three different TIN data sets, `nazareth`, `duchesne` and `parnassus`, each consisting of 50,000 vertices. These TINs represent different types of landscapes, so that we can get a first impression of whether the performance of the heuristics is affected by the terrain morphology. (To draw firm conclusions about this, a more extensive investigation would be needed.)

Most available digital terrain data sets contain many spurious minima. These minima induce very small watersheds in the watershed map, which do not substantially affect the flow of water on the TIN surface. For this reason, it is common practice in hydrological applications to merge these small watersheds into larger ones. This process is called *hydrological conditioning* [6] and there exist many different approaches to do this. In most cases, after computing the watersheds of all minima on the initial terrain model, watersheds are classified as either significant or as insignificant depending on some geometric characteristic like the *topological persistence* of the watershed [19] or the watershed *area measure*, and then insignificant watersheds are removed by merging.

We have measured the performance of the flow heuristics both before and after the conditioning. Although the watershed subdivision after the removal of the spurious minima is what is sought in practice, we also measured the performance of the examined heuristics considering all the minima. This is because the performance of a method that computes watersheds at this stage influences the decisions taken during the conditioning: if some method assigns large regions of the terrain to watersheds of the wrong minima then a wrong set of minima will be considered as insignificant and thus will be removed during the unification process. The criterion that we used for deciding which minima are spurious is the topological persistence of each watershed. The topological persistence of a watershed is the elevation difference of its minimum and its lowest saddle point, where water would spill over into a neighbouring watershed. Watersheds

with the smallest topological persistence get merged with the neighbouring watershed into which water would leak from the lowest saddle point, until 30 watersheds were left.

For the heuristics, we consider two variants of the conditioning algorithm based on topological persistence, which differ only in the way in which the lowest saddle vertex is determined for each watershed:

- In the *standard method* we simply use for each watershed the lowest saddle point on its boundary.
- In the *hybrid method* we proceed as follows. The watersheds which are computed by the heuristics (usually) have different boundaries than the watersheds computed with the exact method and therefore they are not adjacent to the same saddle vertices. Thus, the persistence value of a watershed computed by the heuristic can be different from the persistence value of the corresponding watershed in the exact method. The idea is now to use the persistence value of the exact method when we do the conditioning on the heuristically computed watersheds. We do not want to compute the exact watersheds to determine these values, but, fortunately, this is not necessary: we can just expand paths of locally steepest descent from each saddle, using the exact flow model, determine which local minimum is reached from the saddle, and then associate the saddle to the watershed of that local minimum. Thus, a saddle may be associated to a watershed even if it does not lie on its boundary, as computed by the heuristic.

The results of the experiments are shown in Table 2.

Table 2: Performance of the heuristics (SN = Steepest neighbour, SI = Simulated Immersion, BE = Boundary Expansion), measured as the percentage of TIN surface area that is assigned to the same minima as in the DSD model, before and after conditioning.

	SN	SI	BE
before conditioning			
`nazareth`	74.6	57.1	78.1
`duchesne`	77.9	52.6	73.0
`parnassus`	83.6	61.7	82.3
after conditioning: standard / hybrid			
`nazareth`	54.6 / 95.1	48.4 / 90.5	44.0 / 92.5
`duchesne`	93.7 / 95.9	84.9 / 94.5	86.0 / 90.8
`parnassus`	93.3 / 96.8	78.9 / 93.8	80.1 / 91.4

Discussion.

Before conditioning, the immersion method gives the worst results. The other two methods are comparable, both having overlaps with the exact method which are roughly 70% – 80%. After conditioning using the standard method, the steepest-neighbour heuristic gives consistently the best results. For the `duchesne` and `parnassus` data sets the performance of this method is around 93% which is a good approximation of the exact result. However, all heuristics present a very poor performance for the `nazareth` data set; the performance is close to 50% for each heuristic. This is even worse than the performance of the heuristics before conditioning. This is possibly due to the fact that the `nazareth` data set represents a terrain with a small variation in the elevations of the vertices and many spurious watersheds. Small changes on the boundaries of the outlined watersheds may lead to the computation of different persistence values for each minimum, which in turns results in a different sequence of watershed unification operations.

Conditioning according to the hybrid method leads to a very good performance for all examined heuristics; all heuristics lead to watershed maps that have roughly an 90% – 96% overlap with the one computed by the exact method. Still, the steepest-neighbour method yields slightly better results than the other two heuristics.

The question is, of course, which price do we pay for using exact computations to assign saddles to watersheds in terms of computation times. The next table shows that the price is small: the overall computation times (computing initial watersheds + conditioning) for the hybrid methods is only about 2% – 11% more than the computation times when we use the standard method for conditioning.

Table 3: Computation times in seconds of the heuristics for computing watersheds, and of the conditioning algorithms.

	SN	SI	BE
computing initial watersheds			
`nazareth`	15	15	46
`duchesne`	14	14	45
`parnassus`	14	14	45
conditioning: standard / hybrid			
`nazareth`	112 / 117	113 / 116	115 / 117
`duchesne`	117 / 134	121 / 135	128 / 138
`parnassus`	64 / 72	67 / 74	67 / 76

5. AN OUTPUT-SENSITIVE ALGORITHM FOR COMPUTING WATERSHED MAPS

In the previous section we studied various heuristics for computing watersheds. The resulting watersheds were clearly different from those computed by the exact algorithm, so the question arises whether one can compute the exact watersheds in a more efficient manner than first computing the strip map. Recall that the strip map is just an intermediate structure and it has much higher complexity than the final watershed map. In this section we therefore design an *output-sensitive* algorithm: an algorithm whose running time depends on the size of the watershed map itself and not on the size of the strip map.

In a previous paper [3] an efficient mechanism was developed that extracts important information related to drainage structures on TINs without computing these structures explicitly. More specifically, a technique was described that can expand $\Theta(n)$ flow paths on a TIN, without computing all the intersection points of the paths with the TIN edges. Using suitable data structures, the algorithm can be made to run in $O(n \log n)$ time. This is possible since we treat paths as piecewise linear curves on the surface of the terrain; these curves are evaluated only at selected points using linear functions defined by the triangles of $\mathcal{T}$. The authors show that

this mechanism can be used to compute, in $O(n \log n)$ time, for each local minimum the set of triangles that are *fully* included in its watershed, or the surface network [18] of the terrain. It should be emphasized here that this running-time analysis is done in the standard way, that is, it considers numerical operations as taking unit time—it does not consider the bit-sizes needed to do the computations exactly. We will come back to this issue in more detail later.

We will show that the same basic mechanism can be used to obtain an output-sensitive algorithm for computing the watershed map on a TIN. The key idea is to examine adjacent strips before computing their boundaries explicitly. We call two strips in the strip map *adjacent* if they share a common boundary. If two adjacent strips belong to different watersheds then their common boundary is a watershed boundary and therefore it also appears in the watershed map. Thus, we will use the implicit mechanism to find out first which strips boundaries/flow paths appear as watershed boundaries in the watershed map and then expand explicitly only those paths. We next provide a more detailed description of the algorithm.

The boundary of a strip consists of a segment of a valley edge, a segment of a ridge edge and paths of locally steepest ascent/descent that emanate from TIN vertices on the sides of the strip; the endpoints of the valley edge segment and of the ridge edge segment are the points where the side paths hit these edges for the first time. (Recall that a valley edge is an edge such that the DSD in the interior of both of its incident triangles points towards this edge. Similarly, a ridge edge is an edge such that the DSD in the interior of both of its incident triangles points away from the edge.) We call the endpoints of the valley edge segment of a strip the *foot points* of the strip and we call the endpoints of the ridge edge segment of the strip the *head points* of the strip.

Two strips are adjacent if and only if they share a common side path, or (part of) the same valley edge segment, or (part of) the same ridge edge segment. Two strips share a common side path only if they share a common terrain vertex. Thus, for a strip s we can find which other strips are adjacent to s if we examine the boundaries only around the TIN vertices incident to s and the foot points and head points of s.

To find out which strips constitute the strip map, we cannot afford to compute the strips explicitly. Instead, we seek for the TIN vertices incident to each strip and compute the head points and foot points of the strip. Thus, for each vertex $v \in \mathcal{T}$ consider the set of triangles in $\mathcal{T}$ incident to v. We call these triangles the *star* of v. For all the paths of locally steepest descent and all the paths of locally steepest ascent that emanate from v, we compute only the part of these paths that extends through the star of v. Computing these path segments for each vertex on $\mathcal{T}$ takes $O(n)$ operations in total. The path segments that we constructed around v subdivide the star of v into regions, where each region belongs to a different strip. To compute the foot points and the head points of each strip we expand the rest of the paths that we expanded around each vertex but this time implicitly, using the mechanism of de Berg [3]. This takes $O(n \log n)$ numerical operations in total. We label each expanded path with the two strips that share this path on their boundary. This way, after computing the foot points and head points, we can find out exactly which paths represent the boundary of the same strip by just traversing the foot and head segments. This takes $\Theta(n)$ operations for all strips in total. By traversing these segments and given the strip labels of each path, we have also computed for each strip all the strips that are adjacent to it.

Next we compute for each strip the local minimum whose watershed contains this strip. To do this we pick an arbitrary point in the interior of each strip; for instance we can pick a point in the interior of each strip foot segment. We then expand simultaneously the trickle path from each of the selected points using the implicit expansion mechanism. This takes $O(n \log n)$ operations in total. Thus we compute for each of these paths the local minimum where the path ends. We then label with this local minimum the strip from which the path came from. Then, having labeled all the strips,we look at their common boundaries. If two adjacent strips share a common path boundary and are labeled with different minima, we expand this path explicitly. If two adjacent strips share a common head segment we just mark this segment as part of the boundary of their watersheds [1]. This process takes $\Theta(k)$ numeric operations in total, where k is the total combinatorial complexity of all the explicitly expanded paths. From the above it follows that $O(n \log n + k)$ numerical operations are sufficient for the execution of the entire algorithm. From the analysis also of the implicit expansion mechanism, we conclude that during this process we have to compute at most $O(n + k)$ path points.

As we mentioned already, we measure the computational complexity of the algorithm as the total number of numerical operations that are carried out by the algorithm. This does not include the bit-sizes of the numbers that are handled in these operations. It is the case that the presented mechanism may not avoid to compute points of large bit-size. However, the main goal of this mechanism is to reduce the combinatorial size of the computed data; as we expand a set of flow paths, the goal is to compute fewer path points than if we naively constructed the complete representation of these paths. As we observed from the experiments in Section 3.2, the combinatorial complexity of the strip map of a TIN seems to grow like a superlinear function of the input size on adversarial terrains. Thus, the average number of paths that intersect a terrain edge is not a constant but grows as the input size increases. We expect that, in practice, the presented algorithm will compute on average only a constant number of intersection points per edge. This can be only verified by implementing this mechanism and conducting experiments similar to the ones of Section 3.2.

6. CONCLUDING REMARKS

We presented the first implementation of an algorithm that computes watersheds on a TIN following the exact DSD model, that is, where water always follows the direction of steepest descent on the TIN surface. Since the algorithm needs exact arithmetic, it is a rather costly process because it first computes the strip map and because the exact computations need very large bit-sizes for the coordinates. Hence, the algorithm cannot be used on large data sets. However, the implementation allowed us to investigate to what extent the output of existing heuristics is consistent with the exact flow model. Of the heuristics we investigated, the one pro-

[1]Two adjacent strips that share a common foot segment are always part of the same watershed. Yet, we compute either way the two foot points of each strip to keep track of the topology of the strip.

posed by Mangan and Whitaker [13] performs best. In practice, one often applies conditioning to get rid of small watersheds. We showed that doing this using a hybrid method, which assigns saddles to watersheds using exact computations, produces very good results while being almost as fast as the standard method. Hence, we feel this is a good approach to use in practice. Finally, we presented an exact algorithm for computing watersheds that avoids computing the strip map as an intermediate structure. We leave the implementation of this new algorithm, and the investigation of its running time and memory usage in practice, for future research.

7. REFERENCES

[1] M. de Berg, P. Bose, K. Dobrint, M. van Kreveld, M. Overmars, M. de Groot, T. Roos, J. Snoeyink and S. Yu. The Complexity of Rivers in Triangulated Terrains. In *Proc. 8th Canadian Conference on Computational Geometry*, pages 325–330, 1996.

[2] M. de Berg, O. Cheong, H. Haverkort, J. Lim and L. Toma. The Complexity of Flow on Fat Terrains and its I/O-Efficient Computation. *Computational Geometry: Theory and Applications* 43(4): 331–356, 2010.

[3] M. de Berg, H. Haverkort and C. Tsirogiannis. Implicit Flow Routing on Terrains with Applications to Surface Networks and Drainage Structures. In *Proc. 22nd ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 285–296, 2011.

[4] CGAL, Computational Geometry Algorithms Library. `http://www.cgal.org`.

[5] L. Čomić, L. De Floriani and L. Papaleo. Morse-Smale Decompositions for Modeling Terrain Knowledge. In *Proc. 7th International Conference on Spatial Information Theory*, pages 426–444, 2005.

[6] A. Danner, T. Moelhave, K. Yi, P.K. Agarwal, L. Arge and H. Mitasova. TerraStream: From Elevation Data to Watershed Hierarchies. In *Proc. 15th Annual ACM International Symposium on Advances in Geographic Information Science (ACM GIS)*, pages 212–219, 2007.

[7] H. Edelsbrunner, J. Harer and A. Zomorodian. Hierarchical Morse-Smale Complexes for Piecewise Linear 2-Manifolds. *Discrete & Computational Geometry*, 30(1):87–107, 2003.

[8] A. Fabri, F. Cacciola and R. Wein. CGAL and the Boost Graph Library. In *CGAL User and Reference Manual*, CGAL Editorial Board, 3.7 edition, 2010.

[9] T. Granlund. GMP, the GNU multiple precision arithmetic library. `http://gmplib.org/`.

[10] H. Haverkort, personal communication.

[11] M. Hemmer, S. Hert, L. Kettner, S. Pion and S. chirra. Number Types. In *CGAL User and Reference Manual*. CGAL Editorial Board, 3.8 edition, 2011.

[12] Y. Liu and J. Snoeyink. Flooding Triangulated Terrain. In *Proc. 11th International Symposium on Spatial Data Handling*, pages 137–148, 2005.

[13] A. Mangan and R. Whitaker. Partitioning 3D Surface Meshes Using Watershed Segmentation. *IEEE Transaction on Visualization and Computer Graphics*, 5(4):308–321,1999.

[14] M. McAllister. A Watershed Algorithm for Triangulated Terrains. In *Proc. 11th Canadian Conference on Computational Geometry*, pages 103–106, 1999.

[15] M. McAllister and J. Snoeyink. Extracting Consistent Watersheds From Digital River And Elevation Data. *Annual Conference of the American Society for Photogrammetry and Remote Sensing*, 1999.

[16] E. Moet. Experimental Verification of a Realistic Input Model for Polyhedral Terrains. Technical report UU-CS-2007-052, Utrecht University, 2007.

[17] Palacios-Velez, O.L. and B. Cuevas-Renaud. Automated river-course, ridge and basin delineation from digital elevation data. *Journal of Hydrology* 86: 299–314 (1986).

[18] J. Pfaltz. Surface Networks. *Geographical Analysis Journal*, 8:77-93, 1976.

[19] M. Revsbæk. I/O Efficient Algorithms for Batched Union-Find with Dynamic Set Properties and its Application to Hydrological Conditioning. Master's Thesis, Department of Computer Science, Aarhus University, 2007.

[20] S. Takahashi, T. Ikeda, T.L. Kunii, and M. Ueda. Algorithms for Extracting Correct Critical Points and Constructing Topological Graphs from Discrete Geographic Elevation Data. *Computer Graphics Forum*, 14(3)181–192, 1995.

[21] United States Geological Survey, Seamless Data Warehouse Webpage. `http://seamless.usgs.gov/`.

[22] L. Vincent and P. Soile. Watershed In Digital Spaces: An Eficient Algorithm Based on Immersion Simulation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(6):583–598, 1991.

[23] S. Yu, M. van Kreveld and J. Snoeyink. Drainage Queries in TINs: From Local to Global and Back Again. In *Proc. 7th International Symposium on Spatial Data Handling*, pages 13–1, 1996.