I/O-Efficient Computation of Water Flow Across a Terrain

Lars Arge*
MADALGO
University of Aarhus
Aarhus, Denmark
large@madalgo.au.dk

Morten Revsbæk*
MADALGO
University of Aarhus
Aarhus, Denmark
mrevs@madalgo.au.dk

Norbert Zeh†
Faculty of Computer Science
Dalhousie University
Halifax, Canada
nzeh@cs.dal.ca

ABSTRACT

Consider rain falling at a uniform rate onto a terrain $\mathcal{T}$ represented as a triangular irregular network. Over time, water collects in the basins of $\mathcal{T}$, forming lakes that spill into adjacent basins. Our goal is to compute, for each terrain vertex, the time this vertex is flooded (covered by water). We present an I/O-efficient algorithm that solves this problem using $O(\text{sort}(X)\log(X/M)+\text{sort}(N))$ I/Os, where N is the number of terrain vertices, X is the number of pits of the terrain, $\text{sort}(N)$ is the cost of sorting N data items, and M is the size of the computer's main memory. Our algorithm assumes that the volumes and watersheds of the basins of $\mathcal{T}$ have been precomputed using existing methods.

Categories and Subject Descriptors

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

General Terms

Algorithms, Theory

Keywords

Terrains, geographical information systems, I/O-efficient algorithms

*Supported in part by the Danish National Research Foundation, the Danish Strategic Research Council, and by the US Army Research office. MADALGO is the Center for Massive Data Algorithmics, a center of the Danish National Research Foundation.

†Supported in part by the Natural Sciences and Engineering Research Council of Canada and the Canada Research Chairs programme. Part of this work was done while on sabbatical at MADALGO.

SCG'10, June 13–16, 2010, Snowbird, Utah, USA.

1. INTRODUCTION

An important problem in terrain analysis is the prediction of water flow across a terrain. Traditional approaches focus on computing the river network of the terrain under the assumption that water does not collect in the basins of the terrain. In reality, water *does* collect in the terrain's basins, particularly during heavy rainfall. This may cause basins to spill into adjacent basins, changing the river network of the terrain as a result. Thus, to model the flow of water across a terrain over time, it is necessary to compute the times at which the basins of the terrain spill. In this paper, we solve the more general problem of computing, for every vertex v of a terrain $\mathcal{T}$, the time t_v at which it is flooded. We assume the terrain $\mathcal{T}$ is represented as a triangular irregular network (TIN) and the amount of rain is uniform across the terrain.

The accuracy of predictions of natural phenomena, such as flooding, depends on the precision of the data used in these predictions. High-resolution elevation models even of fairly small geographic regions often exceed the size of a computer's main memory. Current GIS tools cannot handle data sets of this size efficiently and therefore need to work with models of lower resolutions. This sacrifices accuracy because lower-resolution models do not capture all terrain features. For example, in experiments we conducted to predict the flooding of coastal regions of Denmark due to rising sea levels, an elevation model with one data point every 10m failed to capture a dike around an island, and the island was predicted to be flooded if the sea rises by 2m, while the dike can in fact withstand 2m higher sea levels. Using a higher-resolution model with one data point every 2m, we obtained the correct prediction, but storing elevation data for Denmark alone requires 500GB of space at this resolution, an input size well beyond the size of main memory and beyond the reach of current GIS tools.

To process data sets beyond the size of main memory efficiently, it is necessary to develop I/O-efficient algorithms, that is, algorithms that focus on minimizing the number of disk accesses they perform to swap data between disk and internal memory, as a disk access is orders of magnitude slower than an internal-memory computation step. In this paper, we propose an I/O-efficient algorithm for computing the flooding times of all vertices of a terrain $\mathcal{T}$.

The core of the problem is computing the spill times of all basins of $\mathcal{T}$. A simple method to compute these spill times is to simulate the entire sequence of spill events of the basins of $\mathcal{T}$ and maintain the watersheds of all basins that yet have to spill, as well as predicted spill times for these basins based on their current watersheds. An internal-memory solution

based on this idea has been presented in [14]. In Section 3.2, we discuss an $O(N \log N)$ time implementation of this approach using a priority queue and a union-find structure. This simulation approach does not translate into an I/O-efficient algorithm, as the only known I/O-efficient union-find structure [1] requires the sequence of UNION operations to be known in advance. In simulating the sequence of spill events, the *set* of UNION operations is known, but their order depends on the spill times of the basins. Using an internal-memory union-find structure and an I/O-efficient priority queue (e.g., [3, 12]), a cost of $O(X\alpha(X) + \text{sort}(N))$ disk accesses can be achieved, where $\alpha(\cdot)$ is the inverse of Ackermann's function, $\text{sort}(N) \ll N$ is the I/O complexity of sorting N data items (see next section), and X is the number of pits of the terrain. In contrast, the algorithm presented in this paper achieves an I/O complexity of $O(\text{sort}(X) \log(X/M) + \text{sort}(N))$ disk accesses.

In the remainder of this section, we formally define the computational model we use, discuss previous work, and give an overview of our algorithm. Section 2 introduces the terminology and notation we use throughout the paper. Section 3 discusses our algorithm for computing the spill times of the terrain's basins and the flooding times of all terrain vertices. This algorithm makes use of an I/O-efficient meldable priority queue, which we discuss in Section 4.

1.1 I/O Model

We use the *I/O model* with one (logical) disk [2] to design and analyze our algorithm. In this model, the computer is equipped with a two-level memory hierarchy consisting of an *internal memory* and a (disk-based) *external memory*. The internal memory is capable of holding M data items (vertices, edges, etc.), while the external memory is of conceptually unlimited size. All computation has to happen on data in internal memory. Data is transferred between internal and external memory in blocks of B consecutive data items. Such a transfer is referred to as an *I/O operation* or *I/O*. The cost of an algorithm is the number of I/Os it performs. The number of I/Os required to read N contiguous items from disk is $\lceil N/B \rceil$. The number of I/Os required to sort N items is $\text{sort}(N) = \Theta((N/B) \log_{M/B}(N/B))$ [2]. For all realistic values of N, M, and B, we have $N/B < \text{sort}(N) \ll N$.

1.2 Related Work

Due to its importance, the problem of computing water flow across a terrain (usually in the form of a river network) has been studied extensively. Most existing methods for computing river networks assume that once water flows into a small basin of the terrain, it never flows out—in other words, basins do not spill. Therefore, to avoid water getting caught in small local basins, most flow modelling approaches first remove all basins by *flooding* the terrain, that is, conceptually pouring water onto the terrain until all basins are filled [4, 13, 15]. However, this often leads to unrealistic flow patterns, since many important geographical features are removed. Recent papers [1, 5] suggested *partial flooding* algorithms that flood only "small" basins, where the size of a basin can be defined using different measures, such as height, volume or area. Agarwal, Arge and Yi [1] described an $O(\text{sort}(N))$ I/O partial flooding algorithm that removes all basins of small height. This is done by computing the *topological persistence* [10, 11] of each basin and removing the ones with persistence below a small threshold. The key to obtaining an $O(\text{sort}(N))$ I/O solution to this problem is an algorithm that can process a sequence of N UNION and FIND operations using $O(\text{sort}(N))$ I/Os, assuming the entire sequence of operations is provided in advance. Arge and Revsbæk [5] extended the result of [1] to removing basins based on different geometric measures, including volume and area.

Partial flooding methods provide a basis only for approximate solutions to flow modelling, as the underlying assumption is that all "small" basins are flooded at a certain time, while all "big" basins are not. This assumption may not be true, as the spill time of a basin depends on its volume and its watershed, and the watershed of a basin may grow over time as a result of other basins spilling into it; in particular, the watershed of a big basin may grow to a size far exceeding the size of the watershed of a small basin and, thus, the big basin may spill *before* the small basin. To model the flow network at time t accurately, it is necessary to compute the times the basins of $\mathcal{T}$ fill and remove the basins that are full at time t. This is much harder than the partial flooding approaches discussed above, as the above methods are based on local measures associated with each basin, while the computation of the actual spill time of each basin β depends on the spill times of other basins that spill into β. As mentioned in the introduction, Liu and Snoeyink [14] presented an internal-memory algorithm for computing actual spill times and used it for flow prediction. While the paper does not present the algorithm in detail, we discuss an efficient implementation using a union-find structure and a priority queue in Section 3.2; this implementation is needed as part of our I/O-efficient algorithm.

We also require a tree structure that represents the nesting of the basins of $\mathcal{T}$. This tree has been termed the *merge tree* of $\mathcal{T}$ in [7,8] (also see Section 2) and can be computed using $O(\text{sort}(N))$ I/Os using the topological persistence algorithm of [1]. This algorithm is easily augmented to compute the lowest saddle on the boundary of each basin and, as shown in [5], the volume of each basin.

Computing the watershed sizes of all basins of $\mathcal{T}$ is harder. The watershed sizes of all basins of $\mathcal{T}$ are easily computed from the watershed sizes of all pits by summing watershed sizes bottom up in the merge tree. However, the only I/O-efficient algorithm for computing the watersheds of all pits of a terrain exactly is the one of [9]. This algorithm performs $O(\text{sort}(N + S))$ I/Os for fat terrains, where S is the size of the terrain's strip map. The size of the strip map of a fat terrain is $\Theta(N^2)$ in the worst case [9], in which case the exact computation of watersheds becomes infeasible. An approach often taken in practice [16–18] is to assume that water flows only along terrain edges, allowing the computation of watersheds using $O(\text{sort}(N))$ I/Os. While this may lead to poor approximations of the watersheds for irregular terrains [9], TIN's derived from LIDAR data, for example, are rather regular, and the computed watersheds match the real watersheds rather closely.

1.3 New Result

We present an algorithm that computes the flooding times of all terrain vertices using $O(\text{sort}(X) \log(X/M) + \text{sort}(N))$ I/Os, where N is the size of the terrain and X is the number of pits. This assumes that the watershed sizes and volumes of all basins are given and that every vertex is labelled with the pit whose watershed contains it. The cost of computing

this information has been discussed in the previous section. Our algorithm operates on the merge tree $\mathcal{M}$ of the given terrain $\mathcal{T}$. Given a node β of $\mathcal{M}$, we employ a recursive strategy to compute the spill times of all basins represented by descendants of β. If there are at most M such descendants, we use an augmented version of the internal-memory algorithm mentioned in the introduction to compute the spill times of their corresponding basins using $O(\text{sort}(Y))$ I/Os, where Y is the number of descendants of β plus the number of basins that spill into β. Otherwise we consider an appropriate path P from β to a descendant leaf and prove that we can compute the spill times of all basins whose parents belong to P using $O(\text{sort}(Y))$ I/Os, where Y is defined as above. The path P is chosen so that every subtree attached to P contains at most half of β's descendants, and we invoke our algorithm recursively on the root of each such subtree. This ensures that $\log(X/M)$ levels of recursion suffice to break $\mathcal{M}$ into subtrees of size at most M, to which the above internal-memory algorithm can be applied. The cost per level of recursion is $O(\text{sort}(X))$ I/Os. Hence, the total cost of the algorithm is $O(\text{sort}(X)\log(X/M))$ I/Os. This gives only the spill times of the basins. To compute the flooding times of *all* terrain vertices, we use a post-processing step that can be seen as a simpler version of the recursive step of our algorithm and performs $O(\text{sort}(N))$ I/Os.

The intuition behind the computation in each recursive step is the following. In general, there are two directions water can flow across any saddle of the terrain $\mathcal{T}$. Given the flow direction for each saddle, spill times could be computed rather easily, but the direction for each saddle is determined by the spill times of the two basins that merge at this saddle. For the saddles between the basins corresponding to the path P in each recursive steps, we can characterize each flow direction as either "confluent" (toward the basin represented by the leaf of P) or "diffluent" (away from the leaf), and we can show that confluent spill events that influence other spill events—we call these *watershed events*—can themselves depend only on confluent watershed events. This allows us to perform a sweep in the confluent direction first to compute the times of all confluent watershed events. In a second phase, we sweep in the diffluent direction and use the times computed in the first phase to compute the correct times for *all* spill events.

2. PRELIMINARIES

In this section, we introduce the basic terminology used throughout this paper. In the following, we make some assumptions about the structure of the terrain that simplify the exposition in the rest of the paper. All these assumptions can be removed using standard perturbation techniques.

Terrains, pits, and basins. We consider the terrain $\mathcal{T}$ to be represented as a triangular irregular network, which is a planar triangulation each of whose vertices v has an associated elevation $\mathcal{T}(v)$. The elevation of a point interior to a triangle is a linear interpolation of the elevations of the triangle vertices. In this manner, the terrain is represented as a continuous piecewise linear surface, and we use $\mathcal{T}(p)$ to refer to the elevation of the point $p \in \mathbb{R}^2$. We assume that no two adjacent vertices have the same elevation. A *pit* of $\mathcal{T}$ is a local minimum, that is, a terrain vertex all of whose neighbours have higher elevations. A *saddle point* of $\mathcal{T}$ is a vertex v with four vertices w_1, w_2, w_3, w_4 among its neighbours that satisfy $\max(\mathcal{T}(w_1), \mathcal{T}(w_3)) < \mathcal{T}(v) < \min(\mathcal{T}(w_2), \mathcal{T}(w_4))$ and appear in the order w_1, w_2, w_3, w_4 clockwise around v.

A *basin* is a maximal connected set of points $\beta \subseteq \mathbb{R}^3$ such that $\mathcal{T}((x,y)) \leq z \leq h_\beta$, for all $(x,y,z) \in \beta$, where h_β is a fixed elevation chosen as the upper boundary of β. A basin β is *maximal* if there exists a saddle point p_β on the boundary of β such that $\mathcal{T}(p_\beta) = h_\beta$. The point p_β is called the *spill point* of basin β, since water poured into β spills over p_β into an adjacent basin once β becomes full. We assume p_β is unique, that is, there are no two saddles with elevation h_β on the boundary of β. We also assume exactly two basins meet in each spill point. Throughout this paper, we are interested almost exclusively in maximal basins and refer to them simply as basins. Every basin contains at least one pit, and for every pit there exists a unique (maximal) basin that contains only this pit. We call such a basin *elementary*.

Trickle paths, watersheds, and tributaries. The *trickle path* of a point $q \in \mathcal{T}$ is the path that starts at q, continues in the direction of steepest descent for every point it visits, and ends either in a pit p or at the boundary of $\mathcal{T}$. In the former case, water falling onto q collects in the elementary basin corresponding to p; in the latter, it flows off the edge of the terrain. The *watershed* of a pit p is the set of points whose trickle paths end in p. The watershed W^0_β of a basin β is the union of the watersheds of all pits contained in β. More generally, we use W^t_β to denote the watershed of basin β at time t, which is the area such that water falling onto W^t_β at time t collects in basin β. A *tributary* of β is a basin τ such that $\tau \cap \beta = \emptyset$, τ spills before β, and water falling onto $W^{t_\tau}_\tau$ at the time t_τ when τ spills collects in β; that is, τ spills into β and the watershed of τ at this time becomes part of the watershed of β. Note that τ is usually a tributary of more than one basin. In particular, if β is the smallest basin that has τ as a tributary, then τ is a tributary of every basin that contains β but not τ.

Merge tree and flow tree. The basins of $\mathcal{T}$ form a hierarchy that is easily represented using a rooted tree, the *merge tree* $\mathcal{M}$ of $\mathcal{T}$. The leaves of $\mathcal{M}$ are the elementary basins of $\mathcal{T}$. A basin β_1 is the parent of a basin β_2 if and only if $\beta_2 \subset \beta_1$ and there exists no basin β_3 such that $\beta_2 \subset \beta_3 \subset \beta_1$. Under the assumptions we made about $\mathcal{T}$, $\mathcal{M}$ is a binary tree. For a subset S of nodes of $\mathcal{M}$, let $\mathcal{M}(S)$ be the subgraph of $\mathcal{M}$ induced by S. For a node α of $\mathcal{M}$, $\mathcal{M}_\alpha$ denotes the subtree of $\mathcal{M}$ induced by α and its descendants. We do not distinguish between merge tree nodes and their corresponding basins. For a basin β, we use V_β to denote β's volume, W^t_β to denote β's watershed at time t or its size (which will be clear from the context), E_β to denote the event that β spills into an adjacent basin, and t_β to denote the time of event E_β. We call E_β the *spill event* associated with β. See Figure 1 for an illustration of these definitions.

Now consider a subset S of nodes of $\mathcal{M}$ such that $\mathcal{M}(S)$ is a tree. The flow paths between the basins in S form a *flow tree* $\mathcal{F}(S)$ defined as follows. Let S' be the set of leaves of $\mathcal{M}(S)$. The elements of S' are the vertices of $\mathcal{F}(S)$. There is an edge between two such vertices α_1 and α_2 if their watersheds touch in the common spill point of two basins $\beta_1, \beta_2 \in S$. See Figure 2.

3. COMPUTING FLOODING TIMES

Our algorithm for computing the flooding times of all terrain vertices has two phases. The first phase (Sections 3.1–3.3) computes the spill times t_β of all basins of $\mathcal{T}$ using

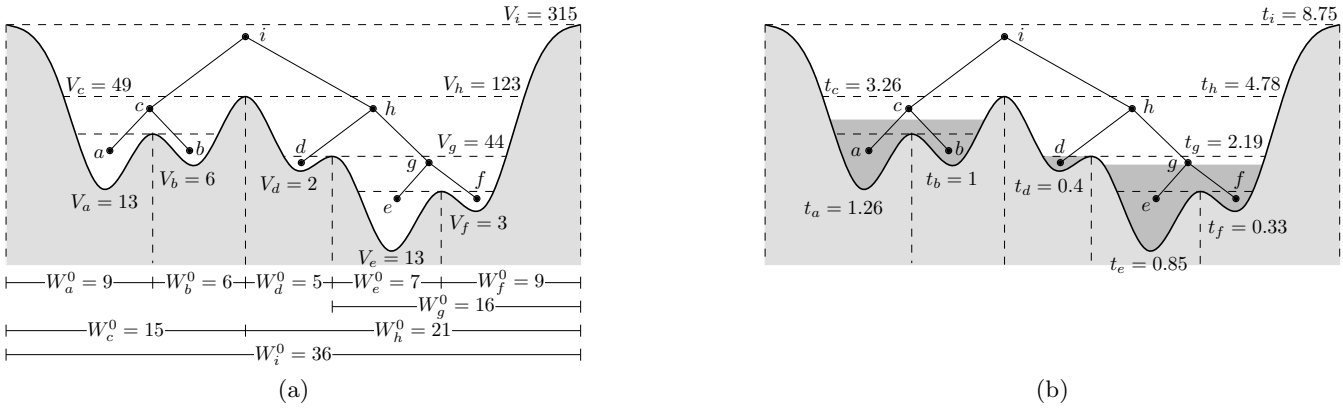

Figure 1: (a) A 1-d terrain with its corresponding merge tree and volumes and initial watersheds of its basins. (b) The spill times of the basins and the water levels in the basins at time $t = 2$.

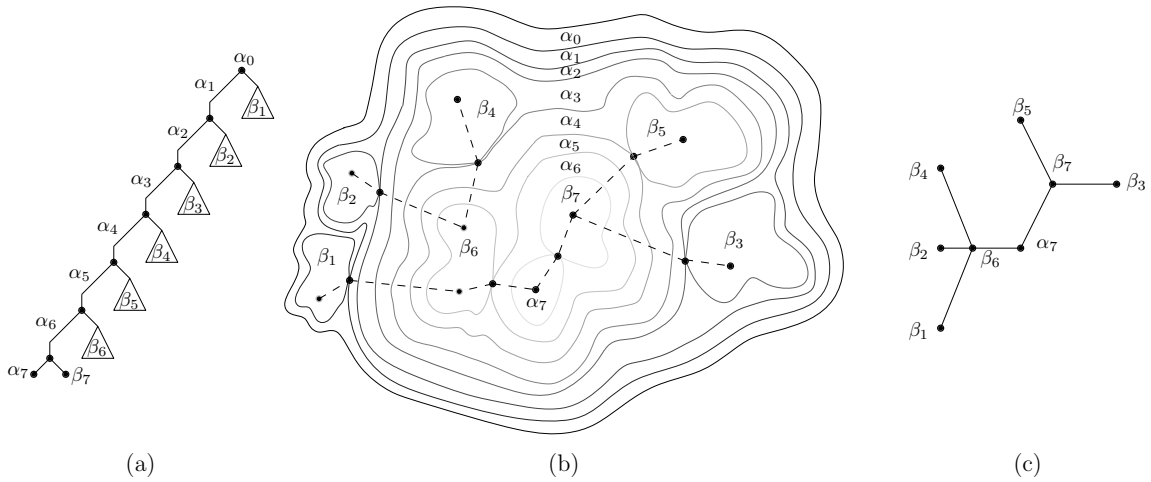

Figure 2: Figure (a) shows the merge tree $\mathcal{M}(S)$ of a set $S = \{\alpha_0, \alpha_1, \beta_1, \ldots, \alpha_7, \beta_7\}$ of basins. Figure (b) illustrates the nesting of these basins using contour lines through their spill points. Figure (c) shows the flow tree of basins $\alpha_7, \beta_1, \beta_2, \ldots, \beta_7$. Every edge corresponds to a spill point and joins the two basins containing the endpoints of the trickle paths starting at this spill point (shown as dashed lines in Figure (b)). Note that the trickle paths corresponding to different edges incident to a flow tree node β_i may end in different pits inside the basin β_i. This is the case for basin β_6 in Figure (b).

$O(\text{sort}(X)\log(X/M))$ I/Os. The second phase (Section 3.4) then computes the flooding times of all vertices of $\mathcal{T}$ using $O(\text{sort}(N))$ I/Os.

3.1 Computing Spill Times

We assume the merge tree $\mathcal{M}$ is given and every node $\beta \in \mathcal{M}$ stores V_β, W_β^0, as well as identifiers of the two elementary basins $\lambda_f(\beta)$ and $\lambda_r(\beta)$ such that $\lambda_f(\beta) \in \mathcal{M}_\beta$ and $W^0_{\lambda_f(\beta)}$ and $W^0_{\lambda_r(\beta)}$ touch in p_β; $\lambda_f(\beta)$ is needed for some data structure queries in our algorithm, while $\lambda_r(\beta)$ is the basin where water spilling from β would collect if we poured water only into β. We further assume the elementary basins of $\mathcal{T}$ have been numbered using a preorder traversal of the flow tree $\mathcal{F}$ of $\mathcal{T}$ starting at an arbitrary root, and every elementary basin stores the preorder interval of its descendants. This information can be computed from the input of our algorithm using $O(\text{sort}(X))$ I/Os using standard techniques. Details appear in the full paper.

Our algorithm for computing the spill times of all basins is recursive. Every recursive call takes a node $\beta \in \mathcal{M}$ and a list $\mathcal{R}_\beta$ of all tributaries of β as input. Each tributary $\tau \in \mathcal{R}_\beta$ stores its spill time t_τ, its watershed $W_\tau := W_\tau^{t_\tau}$ at time t_τ, and the preorder number of an elementary basin it contains. The task of the recursive call on a node β is to compute the spill times of all proper descendants of β in $\mathcal{M}$. The top-level invocation takes the root ρ of $\mathcal{M}$ and an empty list of tributaries as input (since ρ has no tributaries). We distinguish two cases for each recursive call.

If $|\mathcal{M}_\beta| \le M$, we use the algorithm in Section 3.2 below to solve the problem using $O(\text{sort}(X))$ I/Os, where X is the total input size of the invocation, that is, $X := |\mathcal{M}_\beta| + |\mathcal{R}_\beta|$.

If $|\mathcal{M}_\beta| > M$, we compute a *heaviest path* P in $\mathcal{M}_\beta$. This path consists of a sequence of nodes $\alpha_0, \alpha_1, \ldots, \alpha_k$, where $\alpha_0 = \beta$, α_k is a leaf, and, for $1 \le i \le k$, α_i is the child of α_{i-1} with the bigger subtree $\mathcal{M}_{\alpha_i}$ among the two children of α_{i-1}. We use β_i to denote the other child of α_{i-1}. See

Figure 2(a). The first step of the algorithm is to compute the spill times t_{α_i} and t_{β_i} and the set $\mathcal{R}_{\beta_i}$ of tributaries of β_i, for all $1 \leq i \leq k$. To finish the computation of spill times, we recursively invoke the algorithm on each node β_i, $1 \leq i \leq k$. As we show in Section 3.3, the computation of the spill times t_{α_i} and t_{β_i}, for $1 \leq i \leq k$, and of the lists of tributaries of the basins $\beta_1, \beta_2, \ldots, \beta_k$ takes $\mathrm{O}(\mathrm{sort}(k + |\mathcal{R}_\beta|)) = \mathrm{O}(\mathrm{sort}(X))$ I/Os, where $X := |\mathcal{M}_\beta| + |\mathcal{R}_\beta|$. The path P can be computed using $\mathrm{O}(\mathrm{sort}(X))$ I/Os using the Euler tour technique and list ranking [6]. This gives the following result.

THEOREM 1. *The spill times of all basins of a terrain $\mathcal{T}$ can be computed using $\mathrm{O}(\mathrm{sort}(X)\log(X/M))$ I/Os, where X is the number of elementary basins of $\mathcal{T}$.*

PROOF. We prove in Section 3.2 that, given the tributaries of β, we compute the correct spill times for all basins in $\mathcal{M}_\beta$ in the case $|\mathcal{M}_\beta| \leq M$. In Section 3.3, we prove that, in the case $|\mathcal{M}_\beta| > M$, we compute the spill times of basins α_i and β_i and the tributary lists $\mathcal{R}_{\beta_i}$, for $1 \leq i \leq k$, correctly. The correctness of the algorithm then follows by induction. The I/O complexity of an invocation of the algorithm with total input size $X := |\mathcal{M}_\beta| + |\mathcal{R}_\beta|$ and merge tree size $Y := |\mathcal{M}_\beta|$ is given by the recurrence

$$T(X,Y) = \begin{cases} \mathrm{O}(\mathrm{sort}(X)) & Y \leq M \\ \mathrm{O}(\mathrm{sort}(X)) + \sum_{i=1}^{k} T(X_i, Y_i) & Y > M \end{cases},$$

where $Y_i := |\mathcal{M}_{\beta_i}|$ and $X_i := Y_i + |\mathcal{R}_{\beta_i}|$. By the definition of a tributary, every basin τ with $\tau \in \mathcal{M}_\beta$ or $\tau \in \mathcal{R}_\beta$ can be the tributary of at most one basin β_i. Hence, we have $\sum_{i=1}^{k} X_i \leq X$. Furthermore, the choice of the path P as a heaviest path ensures that $Y_i \leq Y/2$, for all $1 \leq i \leq k$. Together, these two facts imply that the recurrence solves to $T(X,Y) = \mathrm{O}(\mathrm{sort}(X)\log(Y/M))$. For the root of $\mathcal{M}$, we have $X = Y$, that is, the overall complexity of the algorithm is $\mathrm{O}(\mathrm{sort}(X)\log(X/M))$, as claimed. □

3.2 Small Basins

To solve the case $|\mathcal{M}_\beta| \leq M$, we provide an implementation of the algorithm of [14] using a union-find structure and a priority queue and extend it to take the tributaries of the basin β into account when computing the spill times of all sub-basins of β. Before running the actual algorithm, we compute, for each tributary τ of β, the elementary sub-basin $\lambda_r(\tau)$ of β that τ spills into; that is, τ spills into β across a saddle on the boundary of $W^0_{\lambda_r(\tau)}$. These basins can be computed from the preorder numbers stored with the tributaries of β and the preorder intervals of the elementary sub-basins of β using $\mathrm{O}(\mathrm{sort}(X))$ I/Os. Details appear in the full paper.

The algorithm maintains a set of *active* basins in $\mathcal{M}_\beta$, which are the basins that have not spilled yet but whose children have already spilled. A basin that has spilled is *finished*, and a basin with at least one unfinished child is *inactive*. The set of active basins always contains the next basin in $\mathcal{M}_\beta$ to spill. We maintain the set of active basins using two data structures: a priority queue Q and a union-find structure U. The priority queue stores the active basins with priorities equal to their predicted spill times—these times decrease over time as we discover more tributaries of active basins. The union-find structure stores the elementary basins (leaves) in $\mathcal{M}_\beta$ and allows us to find, for each such basin α, the active basin α' such that W^0_α is currently part of $W_{\alpha'}$. More precisely, a $\textsc{Find}(\alpha)$ operation returns a representative elementary sub-basin of α', and it is easy to ensure that at all times, the representative of α' stores the ID of α'. Initially, all elementary basins are active and all other basins are inactive; the predicted spill time of an elementary basin α is $t'_\alpha := V_\alpha / W^0_\alpha$. To allow the updating of predicted spill times, each active basin α also stores the time u_α when its watershed changed last—that is, the spill time of its most recent tributary—as well as its residual volume V^r_α at time u_α, which is the portion of V_α left to be filled at this time. Initially, $V^r_\alpha = V_\alpha$ and $u_\alpha = 0$, for every elementary basin α of $\mathcal{T}$.

After initializing the algorithm's data structures as just described, we process the events in Q and $\mathcal{R}_\beta$ by increasing time until Q contains only the basin β. The details of each iteration are as follows. Let τ be the next tributary in $\mathcal{R}_\beta$ to be processed, and let α be the active basin with minimum priority in Q. If $t_\tau < t'_\alpha$, we remove τ from $\mathcal{R}_\beta$ and process E_τ as a *watershed event* with time t_τ, as described below. If $t'_\alpha < t_\tau$, we remove α from Q and consider its sibling σ. If σ is not finished, we process E_α as a *watershed event* with time t'_α; otherwise we process it as a *basin event* with time t'_α.

Watershed event: A watershed event occurs when a basin α spills into a non-full basin α', thereby increasing the watershed of α'. To process a watershed event E_α with time t, we find the active basin α' that α spills into using a $\textsc{Find}(\lambda_r(\alpha))$ operation on U. We update the information for α' as $V^r_{\alpha'} := V^r_{\alpha'} - W_{\alpha'}(t - u_{\alpha'})$, $W_{\alpha'} := W_{\alpha'} + W_\alpha$, $u_{\alpha'} := t$, and $t'_{\alpha'} := t + V^r_{\alpha'}/W_{\alpha'}$, and then update the priority of α' in Q accordingly. If $\alpha \in \mathcal{R}_\beta$, this finishes the processing of E_α. If $\alpha \in \mathcal{M}_\beta$, we need to update U to reflect that water falling or flowing onto W_α after time t flows into α'. We do this by performing a $\textsc{Union}(\lambda_f(\alpha), \lambda_r(\alpha))$ operation on U and ensuring that the representative of the resulting set of elementary basins points to α'. We also label the node α in $\mathcal{M}_\beta$ as finished.

Basin event: A basin event occurs when the second of two sibling basins becomes full, leaving its parent basin in $\mathcal{M}_\beta$ to fill. When processing a basin event E_α with time t, for some $\alpha \in \mathcal{M}_\beta$, both α and its sibling σ in $\mathcal{M}_\beta$ are full at time t. Thus, we label α as finished in $\mathcal{M}_\beta$ and mark its parent γ as active. Water that flowed into α before time t now collects in γ. Thus, we set $W_\gamma := W_\alpha$ and ensure that the representative of α now points to γ. Since both α and σ are full at time t, we set $V^r_\gamma := V_\gamma - V_\alpha - V_\sigma$, $u_\gamma := t$, and $t'_\gamma := t + V^r_\gamma / W_\gamma$. Then we insert γ into Q with priority t'_γ.

The following lemma states the correctness and I/O complexity of the above procedure.

LEMMA 1. *Let β be a basin with at most M sub-basins, and let $X := |\mathcal{M}_\beta| + |\mathcal{R}_\beta|$. The spill times of all basins $\alpha \in \mathcal{M}_\beta$ can be computed using $\mathrm{O}(\mathrm{sort}(X))$ I/Os.*

PROOF. To bound the I/O complexity of the procedure, we observe that Q, U, and $\mathcal{M}_\beta$ fit in memory and that scanning the sorted list of tributaries $\mathcal{R}_\beta$ requires us to keep only one buffer block of size B in memory. Thus, apart from sorting the tributaries in $\mathcal{R}_\beta$ by their spill times using $\mathrm{O}(\mathrm{sort}(X))$ I/Os, the algorithm only loads $\mathcal{M}_\beta$ into memory and scans $\mathcal{R}_\beta$, which takes $\mathrm{O}(X/B)$ I/Os. All other processing happens in memory.

To prove the correctness of the algorithm, we consider the sequence of events $E_1, E_2, \ldots, E_X$ affecting basin β and its sub-basins, sorted by their times $t_1, t_2, \ldots, t_X$. That is, every event E_i is an event E_α with $\alpha \in \mathcal{M}_\beta$ or such that $\alpha \notin \mathcal{M}_\beta$ and α is a tributary of a basin $\alpha' \in \mathcal{M}_\beta$, in which case α is also a tributary of β and belongs to $\mathcal{R}_\beta$.

We use t'_i to denote the "current" predicted time of the event E_i. Consider the basin α such that $E_i = E_\alpha$. Then we define $t'_i := t_i$ if α is a tributary of β and $t'_i := \infty$ if $\alpha \in \mathcal{M}_\beta$ and α is inactive; if $\alpha \in \mathcal{M}_\beta$ and α is active, we define t'_i to be its priority in Q. We use induction on i to prove that $t'_i = t_i$ when event E_i is processed. To do so, we prove that the following holds after processing event E_i:

(i) Events $E_1, E_2, \ldots, E_i$ have been processed. No other events have been processed.

(ii) $W_\alpha = W_\alpha^{t_i}$, for all active basins $\alpha \in \mathcal{M}_\beta$.

(iii) $t'_{i+1} = t_{i+1}$, while $t'_j \geq t_j$, for all $j > i+1$.

The base case is $i = 0$, setting $t_0 = 0$. Property (i) holds because no events have been processed yet. Property (ii) holds because initially $W_\alpha = W_\alpha^0$, for every elementary basin in $\mathcal{M}_\beta$, which is exactly the set of active basins at the beginning of the procedure. To see that (iii) holds, observe that $t'_j = t_j$ if $E_j = E_\tau$, for some tributary τ of β. If $E_j = E_\alpha$, for $\alpha \in \mathcal{M}_\beta$, and α is inactive, then $t'_j = \infty$; if α is active, then $t'_j = V_\alpha / W_\alpha^0 \geq t_j$. Moreover, if $E_1 = E_\alpha$, for some $\alpha \in \mathcal{M}_\beta$, then α has no tributaries and $t_1 = V_\alpha / W_\alpha^0 = t'_1$.

For the inductive step, consider $i > 0$ and assume the claim holds for all $j < i$. By part (i) of the inductive hypothesis, E_{i-1} is the $(i-1)$st event to be processed. By part (iii) of the inductive hypothesis, we have $t'_i = t_i$ and $t'_j \geq t_j > t_i$, for all $j > i$, after processing E_{i-1}. Hence, E_i is the ith event to be processed. This establishes (i).

To prove (ii), we consider the two possible types of event E_i separately. Let $E_i = E_\alpha$. If E_i is a watershed event, part (ii) of the inductive hypothesis implies that we identify the correct watershed $W_{\alpha'}$ into which α spills at time t_i and, hence, that we update $W_{\alpha'}$ to $W_{\alpha'}^{t_i}$; all other watersheds remain unchanged. If E_i is a basin event, α and its sibling are full at time t_i, while α's parent γ has to fill. Furthermore, all water flowing into γ immediately before time t_i collects in α because α's sibling is already full. Therefore, $W_\gamma^{t_i} = W_\alpha^{t_i}$, and we correctly set $W_\gamma := W_\alpha$.

Finally, consider (iii). We consider only events $E_j = E_\alpha$ with α an active basin in $\mathcal{M}_\beta$ because, by definition, $t'_j = t_j$, for every spill event E_j of a tributary of β, and $t'_j = \infty$, for every spill event E_j of an inactive basin in $\mathcal{M}_\beta$. It is easily verified that we update t'_α correctly whenever we increase W_α. By (ii), each event E_h with $h \leq i$ updates W_α if and only if $E_h = E_\tau$, for some tributary τ of α. Thus, $t'_j \geq t_j$, for all $j \geq i$. For $j = i+1$, the spill events of the tributaries of α are a subset of $E_1, E_2, \ldots, E_i$, as they have to happen before $E_\alpha = E_{i+1}$. Thus, we have $t'_{i+1} = t_{i+1}$ after E_i is processed. □

3.3 Basin Sets With Linear Merge Trees

Now consider a path $P = \langle \beta = \alpha_0, \alpha_1, \ldots, \alpha_k \rangle$ from a merge tree node β to a descendant leaf α_k, and let β_i be the sibling of α_i, for all $1 \leq i \leq k$. The basic step of our solution for the case $|\mathcal{M}_\beta| > M$ computes the spill times of $\alpha_1, \beta_1, \alpha_2, \beta_2, \ldots, \alpha_k, \beta_k$, as well as the lists of tributaries $\mathcal{R}_{\beta_1}, \mathcal{R}_{\beta_2}, \ldots, \mathcal{R}_{\beta_k}$ of the basins $\beta_1, \beta_2, \ldots, \beta_k$. Our algorithm for this problem operates on the flow tree $\mathcal{F} := \mathcal{F}(S)$ of the set S of basins in P. Note that every edge in $\mathcal{F}$ corresponds to the spill point connecting two basins α_i and β_i, for $1 \leq i \leq k$, and that $\lambda_f(\alpha_i) = \lambda_r(\beta_i)$ and $\lambda_f(\beta_i) = \lambda_r(\alpha_i)$ in this case. So the tree is easy to construct using a constant number of sorting and scanning steps of S. Details appear in the full paper.

We call a spill event E_{β_i} *confluent*, as the water spilling from β_i at time t_{β_i} spills towards α_k. Similarly, we call a spill event E_{α_i} *diffluent*, as the water spilling from α_i at time t_{α_i} spills away from α_k (α_k is a sub-basin of α_i). We define $\mathcal{F}_{\beta_i}$ to be the subtree of $\mathcal{F}$ rooted in β_i, assuming α_k is chosen as the root of $\mathcal{F}$. Our algorithm proceeds in two phases. The *confluent phase* computes times $t'_{\beta_1}, t'_{\beta_2}, \ldots, t'_{\beta_k}$. These times satisfy $t'_{\beta_i} = t_{\beta_i}$ if E_{β_i} is a watershed event and $t'_{\beta_i} \geq t_{\beta_i}$ if E_{β_i} is a basin event. In addition, this phase constructs a list L_{β_i}, for each $1 \leq i \leq k$, which is a superset of those tributaries of β_i that belong to $\mathcal{F}_{\beta_i}$ or spill into β across a saddle on the boundary of a basin $\beta_j \in \mathcal{F}_{\beta_i}$. The second, *diffluent phase* computes times $t''_{\alpha_1}, t''_{\alpha_2}, \ldots, t''_{\alpha_k}$ and $t''_{\beta_1}, t''_{\beta_2}, \ldots, t''_{\beta_k}$ from the times and potential tributary lists computed in the confluent phase, and we prove that $t''_{\alpha_i} = t_{\alpha_i}$ and $t''_{\beta_i} = t_{\beta_i}$, for all $1 \leq i \leq k$. In the process, the diffluent phase computes the list $\mathcal{R}_{\beta_i}$ of tributaries of β_i, for every $1 \leq i \leq k$.

Intuitively, this two-phase approach works because the confluent phase ensures that the spill times of all basins β_i that are tributaries of other basins are computed correctly (since β_i can be a tributary only if its spill event is a watershed event). A basin α_i can have only confluent tributaries, since basins $\alpha_1, \alpha_2, \ldots, \alpha_k$ are nested. A basin β_i can have only one diffluent tributary, namely α_i, because β_i is a sub-basin of α_j, for all $j < i$, and α_i is a super-basin of α_h, for all $h > i$. Thus, by processing the basins "outwards" from α_k—that is, by decreasing index i—in the diffluent phase, we can ensure that the spill times of all tributaries of a basin are known by the time the basin is processed.

3.3.1 From Tributaries to Spill Times

Both phases of our algorithm use the same elementary procedure to compute a predicted spill time for a basin. To avoid duplication, we describe this procedure here and refer to it as procedure FindSpillTime later. The input of this procedure is a basin α, a time u_α, the residual volume V_α^r of α at time u_α, and the watershed $W_\alpha = W_\alpha^{u_\alpha}$ of α at time u_α. In addition, we are given a priority queue Q containing a set of potential tributaries of α; each entry $\tau \in Q$ satisfies $t'_\tau > u_\alpha$, where t'_τ denotes the priority of τ. The output of the procedure is a predicted spill time t'_α of α and a list L of entries removed from Q while computing t'_α.

Initially, we set $t'_\alpha := u_\alpha + V_\alpha^r / W_\alpha$. Then we repeat the following steps to update t'_α until either Q is empty or the minimum entry τ in Q satisfies $t'_\tau \geq t'_\alpha$. We remove the minimum entry τ (which satisfies $t'_\tau < t'_\alpha$) from Q and append it to L. Then we set $V_\alpha^r := V_\alpha^r - W_\alpha(t'_\tau - u_\alpha)$, $W_\alpha := W_\alpha + W_\tau$, $u_\alpha := t'_\tau$, and $t'_\alpha := u_\alpha + V_\alpha^r / W_\alpha$.

Lemma 2. *Assume at the beginning of procedure* FindSpillTime, *W_α and V_α^r reflect the actual watershed and residual volume of α at time u_α, Q contains only entries τ with $t'_\tau \geq u_\alpha$, and $t'_\tau < t_\alpha$ if and only if τ is a tributary of α. Assume further that every tributary $\tau \in Q$ of α satisfies $t'_\tau = t_\tau$. Then $t'_\alpha \geq t_\alpha$ when the procedure terminates. If Q*

contains every *tributary* τ *of* α *with* $t_\tau \geq u_\alpha$*, then* $t'_\alpha = t_\alpha$*, and* $L = \{\tau \in \mathcal{R}_\alpha \mid t_\tau \geq u_\alpha\}$.

PROOF. First assume for the sake of contradiction that $t'_\alpha < t_\alpha$ when the algorithm terminates. Then the last element $\tau \in Q$ processed by procedure FINDSPILLTIME satisfies $t'_\tau < t'_\alpha$, as the update of t'_α when processing an entry τ does not decrease t'_α below t'_τ. Hence, every processed entry τ satisfies $t'_\tau < t_\alpha$. This implies that all processed elements are tributaries of α that satisfy $u_\alpha < t'_\tau = t_\tau < t_\alpha$, and we process them in order. Thus, after processing every tributary τ, we have $W_\alpha \leq W_\alpha^{t_\tau}$, which implies that $t'_\alpha \geq t_\alpha$, a contradiction.

Now assume Q contains all tributaries of α and $t'_\alpha > t_\alpha$ when procedure FINDSPILLTIME finishes. This is possible only if we do not process all tributaries of α. An unprocessed tributary τ would have to remain in Q by the end of the procedure and, thus, satisfies $t'_\tau > t'_\alpha$. However, $t'_\tau = t_\tau < t_\alpha < t'_\alpha$, again a contradiction.

Finally, observe that, if $t'_\alpha = t_\alpha$, we process exactly the elements $\tau \in Q$ with $t'_\tau < t_\alpha$, and place them into L. These are exactly the tributaries of α that satisfy $t_\tau \geq u_\alpha$. □

3.3.2 Confluent Phase

To implement the confluent phase of the algorithm, we root the flow tree $\mathcal{F}$ in α_k and process its nodes in postorder. With every node $\alpha \in \mathcal{F}$ we associate a list $\mathcal{S}_\alpha \subseteq \mathcal{R}_\beta$ containing all tributaries τ of β that spill into β across a saddle on the boundary of W_α^0. These lists are easy to compute using $O(\text{sort}(X))$ I/Os from the preorder numbers of the elementary basins associated with the tributaries, assuming that every basin in $\mathcal{M}$ stores the largest preorder interval of all its elementary sub-basins, which is easily computed in a preprocessing step using a bottom-up traversal in $\mathcal{M}$. Details appear in the full paper.

During the traversal of $\mathcal{F}$, we ensure that visiting a node β_i produces a priority queue Q_{β_i} that contains all basins τ that satisfy (i) $\tau \in \{\beta_j\} \cup \mathcal{S}_{\beta_j}$, for some $\beta_j \in \mathcal{F}_{\beta_i}$, and (ii) $t'_\tau \geq t'_{\beta_h}$, for all β_h on the path from β_j to β_i in $\mathcal{F}$, inclusive. At any point during the postorder traversal, there is a set of *active* nodes, which are nodes that have been visited already but whose parents in $\mathcal{F}$ have not. We maintain the priority queues of all active nodes in a sequence sorted in the order these nodes are visited and represent this sequence of priority queues using a data structure that supports INSERT and DELETEMIN operations on the last priority queue in the sequence, the creation of a new priority queue at the end of the sequence, as well as a MELD operation, which replaces the last two priority queues in the sequence with their union. In Section 4, we show that the external heap of Fadel et al. [12] can be extended to process a sequence of N INSERT, DELETEMIN, CREATE, and MELD operations using $O(\text{sort}(N))$ I/Os.

The processing of a node β_i distinguishes two cases. If β_i is a leaf, we create a new priority queue Q_{β_i} at the end of the sequence of priority queues of active nodes. If β_i is an internal node with children $\beta_{j_1}, \beta_{j_2}, \ldots, \beta_{j_h}$, the corresponding priority queues $Q_{\beta_{j_1}}, Q_{\beta_{j_2}}, \ldots, Q_{\beta_{j_h}}$ are at the end of the current sequence of priority queues, and we construct Q_{β_i} by melding these priority queues. In both cases, we continue by inserting all elements $\tau \in \mathcal{S}_{\beta_i}$ into Q_{β_i}, with priority $t'_\tau = t_\tau$. Then we use procedure FINDSPILLTIME to compute t'_{β_i} and L_{β_i}; the input to the procedure is β_i, Q_{β_i}, $u_{\beta_i} := 0$, $W_{\beta_i} := W^0_{\beta_i}$, and $V^r_{\beta_i} := V_{\beta_i}$. Once this procedure terminates, we insert β_i into Q_{β_i}, with priority t'_{β_i}.

The confluent phase terminates when the traversal reaches the root α_k of $\mathcal{F}$. Since E_{α_k} is a diffluent spill event, we do not compute its time in this phase of the algorithm. We only construct a list L_{α_k} by collecting all elements in $Q_{\beta_{j_1}}, Q_{\beta_{j_2}}, \ldots, Q_{\beta_{j_h}}$ and $\mathcal{S}_{\alpha_k}$, where $\beta_{j_1}, \beta_{j_2}, \ldots, \beta_{j_h}$ are the children of α_k in $\mathcal{F}$.

To establish the correctness of the confluent phase, we require a number of technical lemmas. The first one characterizes the spill events of basins on the flow path between a basin and its tributary. For three basins $\beta_j, \beta_h \in \mathcal{F}$ and a basin $\alpha \in \{\alpha_i, \beta_i\}$, we say β_h is on the path from β_j to α in $\mathcal{F}$ if β_h is not a sub-basin of α but belongs to the path from β_j to every sub-basin $\alpha' \in \mathcal{F}$ of α.

LEMMA 3. *Consider a tributary* τ *of a basin* $\alpha \in \{\alpha_i, \beta_i\}$*, and assume that* $\tau \in \{\beta_j\} \cup \mathcal{S}_{\beta_j}$*, for some* j*. Then* E_{β_h} *is a watershed event, for every basin* β_h *on the path from* β_j *to* α *in* $\mathcal{F}$ *that is not a sub-basin of* α_i*.*

PROOF. For $j > i$, the lemma holds vacuously because β_j is a sub-basin of α_i in this case and $\alpha \in \{\alpha_i, \beta_i\}$; this implies that every basin β_h on the path from β_j to α is a sub-basin of α_i. For $j \leq i$, assume there exists a basin β_h on the path from β_j to α that is not a sub-basin of α_i and such that E_{β_h} is a basin event. Since β_h is not a sub-basin of α_i, we have $h \leq i$. If $\alpha = \alpha_i$, this implies that α is a sub-basin of α_h and $t_\alpha \leq t_{\alpha_h} \leq t_{\beta_h}$. If $\alpha = \beta_i$, then $h < i$ because $\beta_h \neq \alpha$. Thus, α is again a sub-basin of α_h; in particular, $t_\alpha \leq t_{\beta_h}$. However, since β_j is a tributary of α, we have $t_{\beta_j} < t_\alpha$ and, therefore, $t_{\beta_j} < t_{\beta_h}$ in both cases. For β_j to be a tributary of α, there has to exist a down-hill path from β_j to α at time t_{β_j}. By the choice of β_h, any path from β_j to α has to pass through a point interior to β_h first and then through p_{β_h}. Since β_h is not full at time t_{β_j}, no such path is down-hill at time t_{β_j}, a contradiction. □

Our next lemma shows that, if a basin $\tau \in \{\beta_j\} \cup \mathcal{R}_{\beta_j}$ is a tributary of a basin β_i with $\beta_j \in \mathcal{F}_{\beta_i}$, then $\tau \in L_{\beta_i}$, that is, τ is processed when computing t'_{β_i}; otherwise, if it is a tributary of $\alpha \in \{\alpha_i, \beta_i\}$, then it is in L_γ, for some sub-basin γ of α_i. The first part is used to prove in Lemma 5 below that the confluent phase computes the correct spill times for all confluent watershed events. The second part is used to establish the correctness of the diffluent phase, discussed in Section 3.3.3.

LEMMA 4. *Consider a basin* $\tau \in \{\beta_j\} \cup \mathcal{S}_{\beta_j}$ *with* $\tau \in L_\gamma$*. Assume further that every basin* β_h *in* $\mathcal{F}_\gamma$ *satisfies* $t'_{\beta_h} \geq t_{\beta_h}$*, with equality if* E_{β_h} *is a watershed event. If* τ *is a tributary of* β_i *and* $\beta_j \in \mathcal{F}_{\beta_i}$*, then* $\gamma = \beta_i$*. Otherwise, if* τ *is a tributary of a basin* $\alpha \in \{\alpha_i, \beta_i\}$*, then* γ *is a sub-basin of* α_i*.*

PROOF. First assume τ is a tributary of β_i and $\beta_j \in \mathcal{F}_{\beta_i}$. Then E_τ is a watershed event and either $\tau \in \mathcal{S}_{\beta_j}$ or $\tau = \beta_j$. In both cases, we have $t'_\tau = t_\tau$, in the latter case by the assumption that $t'_{\beta_h} = t_{\beta_h}$ for every watershed event on the path from β_j to γ. If γ belongs to the path from β_j to β_i and $\gamma \neq \beta_i$, then we have $t'_\gamma = t_\gamma < t_\tau$ because τ is a tributary of β_i and, hence, E_γ is a watershed event. However, $\gamma \neq \alpha_k$ in this case, and every element τ in L_γ is removed from Q_γ while computing t'_γ. Thus, $t'_\tau < t'_\gamma$, a contradiction. This shows that γ is an ancestor of β_i in $\mathcal{F}$. This, however, implies that $t'_{\beta_i} \geq t_{\beta_i} > t_\tau = t'_\tau$. Thus, the computation of t'_{β_i} removes τ from Q_{β_i}, and $\tau \in L_{\beta_i}$.

Now assume that τ is a tributary of a basin $\alpha \in \{\alpha_i, \beta_i\}$ and, if $\alpha = \beta_i$, that $\beta_j \notin \mathcal{F}_{\beta_i}$. Then, if γ belongs to the path from β_j to α_i, we obtain $t'_\gamma = t_\gamma < t_\tau = t'_\tau$ on the one hand, because τ is a tributary of α and, by Lemma 3, E_γ is a watershed event; on the other hand, $t'_\gamma > t'_\tau$ because otherwise $\tau \notin L_\gamma$. Thus, we obtain a contradiction and γ is a sub-basin of α_i. □

LEMMA 5. *For all $1 \le i \le k$, we have $t'_{\beta_i} \ge t_{\beta_i}$. If E_{β_i} is a watershed event, equality holds.*

PROOF. By induction on $|\mathcal{F}_{\beta_i}|$. The base case is $|\mathcal{F}_{\beta_i}| = 0$, in which case there is nothing to prove. So consider a node β_i and assume the claim holds for all its descendants. Then, by Lemma 4, every tributary $\tau \in \{\beta_j\} \cup \mathcal{S}_{\beta_j}$, for some descendant β_j of β_i, is inspected when computing t'_{β_i}, and each such tributary satisfies $t'_\tau = t_\tau$ by the inductive hypothesis. In particular, each such tributary belongs to Q_{β_i} when invoking procedure FINDSPILLTIME to compute t'_{β_i}. We prove that any other element $\alpha \in Q_{\beta_i}$ satisfies $t'_\alpha \ge t_{\beta_i}$. By Lemma 2, this implies that $t'_{\beta_i} \ge t_{\beta_i}$, with $t'_{\beta_i} = t_{\beta_i}$ if Q_{β_i} contains *all* tributaries of β_i, which is the case if E_{β_i} is a watershed event. Indeed if there was a tributary τ such that $\tau \in \{\beta_j\} \cup \mathcal{R}_{\beta_j}$, for some $\beta_j \notin \mathcal{F}_{\beta_i}$, the water spilling from τ into β_i would have to flow through a point interior to α_i and then through $p_{\alpha_i} = p_{\beta_i}$. Since $t_\tau < t_{\beta_i}$ and E_{β_i} is a watershed event, α_i is not full at time t_τ, that is, the path from τ to β_i cannot be down-hill at time t_τ, a contradiction.

So consider an element $\alpha \in Q_{\beta_i}$, and assume that $\alpha \in \{\beta_j\} \cup \mathcal{S}_{\beta_j}$, for some $\beta_j \in \mathcal{F}_{\beta_i}$, and $t'_\alpha < t_{\beta_i}$. We prove that α is a tributary of β_i. We have $t'_\alpha \ge t'_{\beta_h}$, for all β_h on the path from β_j to β_i, excluding β_i, because otherwise β_h would have removed α from Q_{β_h} when computing t'_{β_h}. Since, by the inductive hypothesis, $t_{\beta_h} \le t'_{\beta_h}$, this implies that $t_{\beta_h} < t_{\beta_i}$. Also by the inductive hypothesis, we have $t'_\alpha \ge t_\alpha$. If $t_\alpha \ge t_{\beta_h}$, for all β_h on the path from β_j to β_i, then α is a tributary of β_i. If $t_\alpha < t_{\beta_h} \le t'_\alpha$, for some β_h, then, by the inductive hypothesis, E_α is not a watershed event. In particular, $\alpha = \beta_j$, for some $j < i$, and $t_{\beta_i} < t_{\alpha_j} \le t_{\beta_j} = t_\alpha$ because β_i is a sub-basin of α_j. This contradicts our assumption that $t_\alpha \le t'_\alpha < t_{\beta_i}$. □

3.3.3 Diffluent Phase

In the diffluent phase, we process the basins in $\mathcal{M}_\beta$ in the order $\alpha_k, \beta_k, \alpha_{k-1}, \beta_{k-1}, \ldots, \alpha_1, \beta_1$. We initialize a priority queue Q to contain all events in L_{α_k}. Then we repeat the following steps for $i = k, k-1, \ldots, 1$:

First we compute t''_{α_i}. If $i = k$, we do this by invoking procedure FINDSPILLTIME with arguments α_k, Q, $u_{\alpha_k} := 0$, $W_{\alpha_k} := W^0_{\alpha_k}$, and $V^r_{\alpha_k} := V_{\alpha_k}$. If $i < k$, we invoke the procedure with arguments α_i, Q, $u_{\alpha_i} := \max(t''_{\alpha_{i+1}}, t''_{\beta_{i+1}})$, $W_{\alpha_i} := \max(W_{\alpha_{i+1}}, W_{\beta_{i+1}})$, and $V^r_{\alpha_i} := V_{\alpha_i} - V_{\alpha_{i+1}} - V_{\beta_{i+1}}$. We place the elements of Q processed by procedure FINDSPILLTIME into a list $\mathcal{R}''_{\alpha_i}$.

Then we compute t''_{β_i}. To do so, we insert α_i (with priority t''_{α_i}) and the events in L_{β_i} into Q and invoke procedure FINDSPILLTIME with arguments β_i, Q, $u_{\beta_i} := 0$, $W_{\beta_i} := W^0_{\beta_i}$, and $V^r_{\beta_i} := V_{\beta_i}$. We place the elements of Q processed by procedure FINDSPILLTIME into a list $\mathcal{R}''_{\beta_i}$.

The computation of t''_{α_i} and t''_{β_i} may not process all elements in Q, and some of these elements may be sub-basins of α_i. These elements should be ignored in subsequent computations, as they cannot be tributaries of any α_j or β_j with $j < i$. To ensure this, we augment the above procedure to ignore in iteration i all elements α_j or β_j in Q with $j > i$. (Note that we ignore only these basins, not the elements of $\mathcal{S}_{\alpha_k}$ or $\mathcal{S}_{\beta_j}$ with $j > i$.)

LEMMA 6. *For all $1 \le i \le k$, we have $t_{\alpha_i} = t''_{\alpha_i}$, $t_{\beta_i} = t''_{\beta_i}$, $\mathcal{R}_{\beta_i} = \mathcal{R}''_{\beta_i}$, and $\mathcal{R}_{\alpha_i} = \mathcal{R}''_{\alpha_k} \cup \bigcup_{j=i}^{k-1} \mathcal{R}''_{\beta_j}$.*

PROOF. By induction on i. For $i = k$, Lemmas 4 and 5 imply that every tributary τ of α_k belongs to L_{α_k} and satisfies $t'_\tau = t_\tau$. Any other basin $\alpha \in L_{\alpha_k}$ satisfies $t'_\alpha \ge t_\alpha$, and the same arguments as in the proof of Lemma 5 show that $t'_\alpha \ge t_{\alpha_k}$ in this case. Hence, by Lemma 2, the first iteration computes $t''_{\alpha_k} = t_{\alpha_k}$ and processes only tributaries of α_k, that is, $\mathcal{R}''_{\alpha_k} = \mathcal{R}_{\alpha_k}$.

Now consider β_k. By Lemmas 4 and 5, every tributary of β_k belongs to $\{\alpha_k\} \cup L_{\alpha_k} \cup L_{\beta_k}$, and we have just argued that the computation of α_k processes only tributaries of α_k. Hence, Q contains all tributaries of β_k. We have just argued that $t''_{\alpha_k} = t_{\alpha_k}$ and, by Lemma 5, any other tributary τ of β_k satisfies $t'_\tau = t_\tau$. For an element $\alpha \in Q$ that is not a tributary of β_k, the same arguments as in the proof of Lemma 5 show that $t'_\alpha \ge t_{\beta_k}$. (However, we have to distinguish the cases $\beta_j \in \mathcal{F}_{\beta_k}$ and $\beta_j \notin \mathcal{F}_{\beta_k}$, where $\alpha \in \{\beta_j\} \cup \mathcal{R}_{\beta_j}$.) Thus, by Lemma 2, we compute $t''_{\beta_k} = t_{\beta_k}$ and $\mathcal{R}''_{\beta_k} = \mathcal{R}_{\beta_k}$.

Now assume that $i < k$ and that the inductive hypothesis holds for all $j > i$. By Lemmas 4 and 5, every tributary τ of α_i is contained in $L := L_{\alpha_k} \cup \bigcup_{j=i+1}^{k} L_{\beta_j} = Q \cup \bigcup_{j=i+1}^{k} (\mathcal{R}''_{\alpha_j} \cup \mathcal{R}''_{\beta_j})$ and satisfies $t'_\tau = t_\tau$. Using the arguments from the proof of Lemma 5 again, every $\alpha \in L$ that is not a tributary of α_i satisfies $t'_\alpha \ge t_{\alpha_i}$. Moreover, the elements of $L \setminus Q$ are exactly the tributaries of α_{i+1} and β_{i+1}, that is, the tributaries τ of α_i that satisfy $t_\tau < u_{\alpha_i}$. Hence, by Lemma 2, we compute $t''_{\alpha_i} = t_{\alpha_i}$ and $\mathcal{R}''_{\alpha_i}$ to contain exactly the tributaries τ of α_i that satisfy $t_\tau \ge u_{\alpha_i}$. Thus, $\mathcal{R}_\alpha = \mathcal{R}''_{\alpha_i} \cup \bigcup_{j=i+1}^{k} (\mathcal{R}''_{\alpha_j} \cup \mathcal{R}''_{\beta_j})$. The correctness proof for the computation of t''_{β_i} and $\mathcal{R}''_{\beta_i}$ is identical to that for β_k. □

LEMMA 7. *The computation of the spill times t_{α_i} and t_{β_i} and of the tributary lists $\mathcal{R}_{\beta_i}$, for all $1 \le i \le k$, takes $O(\mathrm{sort}(X))$ I/Os.*

PROOF. The correctness of the algorithm is established by Lemma 6. Its I/O-complexity is established as follows: Both phases of the algorithm perform $O(X)$ priority queue operations, as every event is inserted and removed from a priority queue only once in each of the two phases and the confluent phase performs at most X MELD operations. By Theorem 3 in Section 4, these priority queue operations cost $O(\mathrm{sort}(X))$ I/Os. Apart from this, the algorithm traverses trees $\mathcal{F}$ and $\mathcal{M}(S)$ once each and scans lists associated with the tree nodes of total length $O(X)$. After arranging the tree nodes and list elements in the right order, using a sorting step, the scanning of these lists takes $O(X/B)$ I/Os. □

3.4 Computing the Flooding Times of All Terrain Vertices

The terrain vertices that are not spill points of basins can be seen as partitioning the basins of $\mathcal{T}$ into sub-basins as follows. For each terrain vertex v, let α_v be the smallest basin that contains v. For a basin β, let $V(\beta)$ be the set of terrain vertices with $\alpha_v = \beta$. Each vertex $v \in V(\beta)$ defines a sub-basin $\beta_{\mathcal{T}(v)}$ of β containing the portion of β below elevation $\mathcal{T}(v)$. Let $v_1, v_2, \ldots, v_k$ be the vertices in $V(\beta)$ sorted by

increasing elevation. Then the water from every tributary in $\mathcal{R}''_{\beta_i}$ first collects in $\beta_{\mathcal{T}(v_1)}$; once $\beta_{\mathcal{T}(v_1)}$ is full, the water collects in $\beta_{\mathcal{T}(v_2)}$, and so on. This is just a simplified version of the diffluent flow computation in Section 3.3.3, and computing the spill times of basins $\beta_{\mathcal{T}(v_1)}, \beta_{\mathcal{T}(v_2)}, \ldots, \beta_{\mathcal{T}(v_k)}$—that is, the flooding times of vertices $v_1, v_2, \ldots, v_k$—from the list $\mathcal{R}''_\beta$ takes $\mathrm{O}(\mathrm{sort}(N))$ I/Os in total for all basins of $\mathcal{T}$. The computation of the sets $V(\beta)$ can be incorporated in the computation of the basins of $\mathcal{T}$ without increasing the I/O complexity of that phase. Details appear in the full paper. Thus, we have the following result.

THEOREM 2. *The flooding times of all vertices of a terrain $\mathcal{T}$ with N vertices and X pits can be computed using $\mathrm{O}(\mathrm{sort}(X)\log(X/M) + \mathrm{sort}(N))$ I/Os, given the watersheds and volumes of all basins of $\mathcal{T}$.*

4. A MELDABLE PRIORITY QUEUE

In this section, we discuss how to extend the external heap of Fadel et al. [12] to obtain a meldable priority queue that can be used in the procedure in Section 3.3.2. This structure maintains a sequence of priority queues $Q_1, Q_2, \ldots, Q_k$ under CREATE, INSERT, DELETEMIN, and MELD operations. Given the current sequence $Q_1, Q_2, \ldots, Q_k$, a CREATE operation creates a new, empty priority queue Q_{k+1} and appends it to the end of the sequence; an INSERT(x) operation inserts element x into Q_k; a DELETEMIN operation deletes and returns the minimum element in Q_k; a MELD operation replaces Q_{k-1} and Q_k with a new priority queue $Q'_{k-1} := Q_{k-1} \cup Q_k$ containing the elements from Q_{k-1} and Q_k. We prove the following result.

THEOREM 3. *There exists a linear-space data structure that uses $\mathrm{O}(\mathrm{sort}(N))$ I/Os to process a sequence of N* CREATE, INSERT, DELETEMIN, *and* MELD *operations.*

4.1 The Structure

Each priority queue Q_i is represented as an *external heap* similar to the one presented in [12]. The underlying structure is a rooted tree whose internal nodes have between $a = M/(4B)$ and $b = M/B$ children. There are two types of leaves: *regular* leaves are on the lowest level of the tree, which we call the *leaf level*; leaves above the leaf level are *special*.

Every node v has an associated buffer X_v. If v is an internal node, X_v has size M. If v is a leaf, there is no bound on the size of X_v. The elements in X_v are sorted, and the buffer contents of adjacent nodes satisfy the *heap property*: for every node v with parent u and every pair of elements $x \in X_u$ and $y \in X_v$, we have $x \leq y$.

To support INSERT and DELETEMIN operations efficiently, every priority queue Q_i is equipped with an *insert/delete buffer* or *I/D buffer* for short. This buffer is capable of holding up to M elements and stores the minimum elements in Q_i as well as newly inserted elements. To distinguish newly inserted elements in Q_i's I/D buffer from minimum elements, Q_i has an associated priority p_i, which is the minimum priority of the elements stored in the external part of Q_i.

The I/D buffers of priority queues $Q_1, Q_2, \ldots, Q_k$ are kept on a *buffer stack*, with the buffer of Q_1 at the bottom and the buffer of Q_k at the top. The I/D buffers of consecutive priority queues are separated by special *marker elements*. We always keep the topmost M elements of this stack in memory, which ensures in particular that the I/D buffer of Q_k is in memory.

4.2 Operations

Create. To create a new priority queue Q_{k+1}, a CREATE operation pushes a new marker element onto the buffer stack.

Insert. An INSERT operation on Q_k adds the inserted element x to Q_k's I/D buffer. If the buffer now contains M elements, we FLUSH the buffer as described below. Then we FILL the I/D buffer with the $M/2$ minimum elements in Q_k and update p_k accordingly.

DeleteMin. A DELETEMIN operation on Q_k returns the minimum element in Q_k's I/D buffer. Before doing so, however, it checks whether the I/D buffer of Q_k is empty or its minimum element is greater than p_k. If so, it FLUSHes the I/D buffer and then FILLs it with the $M/2$ minimum elements in Q_k.

Meld. A MELD operation behaves differently depending on the structure of the two involved priority queues, Q_{k-1} and Q_k. During different stages of its life time, a priority queue may have no external portion because all its elements fit in the I/D buffer. If at least one of the two priority queues, say Q_k, has no external portion, we destroy its I/D buffer, and INSERT its elements into Q_{k-1}. If both priority queues have external portions, we FLUSH and destroy their I/D buffers, MERGE their two trees as described below, and then FILL the I/D buffer of the merged priority queue $Q'_{k-1} := Q_{k-1} \cup Q_k$ with the $M/2$ minimum elements in Q'_{k-1}.

Flush. A FLUSH operation of an I/D buffer containing K elements creates a new leaf l at the leaf level and stores these K elements in l. Then it applies a HEAPIFY operation to the path from l to the root to restore the heap property. If the addition of l increased the degree of l's parent p to $M/B + 1$, we split p into two nodes p' and p'', each with half of p's children. We associate the buffer of p with p' and populate the buffer of p'' with the M smallest elements in its subtree using a FILL operation. If this split increases the degree of p's parent to $M/B + 1$, we apply this rebalancing procedure recursively until we reach the root. If the root has degree greater than M/B, we split it into two nodes with a new parent.

Fill. A FILL operation applied to a node v repeatedly takes the smallest element stored in the buffers of v's children, removes it from the corresponding child's buffer and stores it in v's buffer. This continues until M elements have been collected in v's buffer or there are no elements left in the buffers of v's children. Whenever a child buffer runs empty while filling v's buffer, its buffer is filled recursively before continuing to fill v's buffer. Once there are no more elements left in a node v's subtree, we mark v as *exhausted*, in order to avoid repeatedly trying to fill v with elements. More precisely, a node is marked as exhausted once its buffer becomes empty and all its children are exhausted. To fill the I/D buffer of a priority queue Q_k with the $M/2$ smallest elements in Q_k, we transfer the $M/2$ smallest elements from the root of Q_k into the I/D buffer, FILLing the root's buffer whenever it runs empty. If the root becomes exhausted in the process, we delete the entire external portion of the priority queue.

Merge. A MERGE operation between two trees T_1 and T_2 of heights $h_1 > h_2$ proceeds as follows. We locate a node v at height $h_2 + 1$ in T_1 and make the root r_2 of T_2 a child of v. We also create a new special leaf node l that is a child of v. Now let $v = v_0, v_1, \ldots, v_h$ be the ancestors of v, by increasing distance from v, let K_i be the number of elements in X_{v_i}, and let $K = \sum_{i=0}^{h} K_i$. First we collect the elements in $X_{v_0}, X_{v_1}, \ldots, X_{v_h}$ in sorted order and store them in l. Then we repeatedly remove the minimum element from X_{r_2} and X_l, refilling X_{r_2} as necessary, until we have collected the K smallest elements in the subtrees rooted at r_2 and l. We store these elements in buffers $X_{v_0}, X_{v_1}, \ldots, X_{v_h}$, sorted top-down and so that each node v_i receives K_i elements. We clear the "exhausted" labels of all nodes among $v_0, v_1, \ldots, v_h$. If v has degree greater than $M/2$ as a result of gaining two children, r_2 and l, we rebalance the tree using node splits starting at v similar to the rebalancing done after a FLUSH operation.

Heapify. The final operation to discuss is the HEAPIFY operation. Given a node v and its path $v = v_0, v_1, \ldots, v_h$ to the root, a HEAPIFY operation ensures that the elements stored on the path satisfy the heap property. It assumes that the elements in $v_1, v_2, \ldots, v_h$ already do so, but some elements in v_0 may be less than elements stored at higher nodes. To restore the heap property, we sort the elements in v_0 and collect the elements in $v_1, v_2, \ldots, v_h$ in sorted order. Then we merge the two sorted sequences to obtain a sorted sequence of the elements stored in $v_0, v_1, \ldots, v_h$ and distribute these elements over the buffers of nodes $v_0, v_1, \ldots, v_h$, assigning the same number of elements to each node as it had before the operation and storing the elements sorted top-down on the path. If some of the nodes on the path were marked as exhausted before this operation, we unmark them.

4.3 Analysis

To prove the correctness of all priority queue operations, we need to verify that the heap property is maintained at all times. This is little more than an exercise and is therefore omitted.

It is easy to see that every CREATE, INSERT, DELETEMIN, and MELD operation makes only $O(1)$ changes to the buffer stack and thus has an amortized cost of $O(1/B)$ I/Os, excluding the manipulations performed by the FLUSH, FILL, MERGE, and HEAPIFY operations they trigger. The following two lemmas bound the cost of all FLUSH, FILL, MERGE, and HEAPIFY operations performed during a sequence of N priority queue operations. Due to lack of space, their proofs are omitted.

LEMMA 8. *During a sequence of N priority queue operations, at most $O(N/M)$ FLUSH, FILL, MERGE, and HEAPIFY operations are performed.*

LEMMA 9. *The amortized cost per FLUSH, FILL, MERGE or HEAPIFY operation is $O((M/B)\log_{M/B}(N/M))$ I/Os.*

5. REFERENCES

[1] P. Agarwal, L. Arge, and K. Yi. I/O-efficient batched union-find and its applications to terrain analysis. In *Proc. 22nd SCG*, pp. 167–176, 2006.

[2] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Comm. of the ACM*, 31(9):1116–1127, 1988.

[3] L. Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003.

[4] L. Arge, J. Chase, P. Halpin, L. Toma, D. Urban, J. S. Vitter, and R. W. and. Flow computation on massive grid terrains. In *Proc. 9th ACM GIS*, pp. 82–87, 2001.

[5] L. Arge and M. Revsbæk. I/O-efficient contour tree simplification. In *Proc. 20th ISAAC*, pp. 1155–1165, 2009.

[6] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proc. 6th SODA*, pp. 139–149, 1995.

[7] A. Danner. *I/O Efficient Algorithms and Applications in Geographic Information Systems*. PhD thesis, Dept. of Comp. Sci., Duke University, 2006.

[8] A. Danner, K. Yi, T. Mølhave, P. K. Agarwal, L. Arge, and H. Mitasova. TerraStream: From elevation data to watershed hierarchies. Manuscript, 2007.

[9] M. de Berg, O. Cheong, H. Haverkort, J.-G. Lim, and L. Toma. I/O-efficient flow modelling on fat terrains. In *Proc. 10th WADS*, pp. 239–250, 2007.

[10] H. Edelsbrunner, J. Harer, and A. Zomorodian. Hierarchical morse complexes for piecewise linear 2-manifolds. In *Proc. 17th SCG*, pp. 70–79, 2001.

[11] H. Edelsbrunner, D. Letscher, and A. Zomorodian. Topological persistence and simplification. In *Proc. 41st FOCS*, pp. 454–463, 2000.

[12] R. Fadel, K. V. Jakobsen, J. Katajainen, and J. Teuhola. Heaps and heapsort on secondary storage. *Theor. Comp. Sci.*, 220(2):345–362, 1999.

[13] S. Jenson and J. Domingue. Extracting topographic structure from digital elevation data for geographic information system analysis. *Photogrammetric Eng. and Remote Sensing*, 54(11):1593–1600, 1988.

[14] Y. Liu and J. Snoeyink. Flooding triangulated terrain. In *Proceedings of the 11th International Symposium on Spatial Data Handling*, pages 137–148, 2005.

[15] J. F. O'Callaghan and D. M. Mark. The extraction of drainage networks from digital elevation data. *Computer Vision, Graphics and Image Proc.*, 28(3):323–344, 1984.

[16] O. Palacios-Velez, W. Gandoy-Bernasconi, and B. Cuevas-Renaud. Geometric analysis of surface runoff and the computation order of unit elements in distributed hydrological models. *J. Hydrology*, 211:266–274, 1998.

[17] D. M. Theobald and M. F. Goodchild. Artifacts of TIN-based surface flow modeling. In *Proc. GIS/LIS'90*, pp. 955–964, 1990.

[18] G. Tucker, S. Lancaster, N. Gasparini, and S. Rybarczyk. An object-oriented framework for hydrology and geomorphic modeling using triangulated irregular networks. *Computers and Geosciences*, 27(8):959–973, 2001.