

Implicit Flow Routing on Terrains with Applications to Surface Networks and Drainage Structures

Mark de Berg*

Herman Haverkort[†]

Constantinos Tsirigiannis[‡]

Abstract

Flow-related structures on terrains are defined in terms of paths of steepest descent (or ascent). A steepest descent path on a polyhedral terrain \mathcal{T} with n vertices can have $\Theta(n^2)$ complexity. The watershed of a point p —the set of points on \mathcal{T} whose paths of steepest descent reach p —can have complexity $\Theta(n^3)$. We present a technique for tracing a collection of n paths of steepest descent on \mathcal{T} implicitly in $O(n \log n)$ time. We then derive $O(n \log n)$ time algorithms for: (i) computing for each local minimum p of \mathcal{T} the triangles contained in the watershed of p and (ii) computing the surface network graph of \mathcal{T} .

We also present an $O(n^2)$ time algorithm that computes the watershed area for each local minimum of \mathcal{T} .

1 Introduction

Background and motivation. In many applications it is necessary to visualize, compute, or analyze flows on a height function defined over some 2- or higher-dimensional domain. Often the direction of flow is given by the gradient and the domain is a region in \mathbb{R}^2 . The flow of water in mountainous regions is a typical example of this. Modeling and analyzing water flow is important for predicting floods, planning dams, and other water-management issues. Hence, flow modeling and analysis has received ample attention in the GIS community [8, 9, 11, 13].

In GIS, mountainous regions are usually modeled as a DEM or as a TIN. A DEM (digital elevation model) is a uniform grid, where each grid cell is assigned an elevation. Because of the discrete nature of DEMs, it is hard to model flow in a natural and accurate way. A TIN (triangulated irregular network) is obtained by assigning elevations to the vertices of a two-dimensional triangulation; it is the model we adopt in this paper. In computational geometry, a TIN is usually referred to as a (*polyhedral*) *terrain*. One advantage of polyhedral terrains over DEMs is that one can use a non-uniform resolution, using small triangles in rugged areas and larger triangles in flat areas. Another advantage is that

the surface defined by a polyhedral terrain is continuous, which makes flow modeling more natural. Indeed, the standard flow model on polyhedral terrains is simply that water follows the direction of steepest descent. To make the flow direction well defined, it is then often assumed—and we will also make this assumption—that the direction of steepest descent is unique for every point on the terrain. For instance, the terrain should not contain horizontal triangles.¹

There are several important structures related to the flow of water on a polyhedral terrain \mathcal{T} . The simplest structure is the path that water would follow from a given point p on the terrain. This path is called the *trickle path* and, as already mentioned, in our model it is simply the path of steepest descent. Another important structure is the *watershed* of a point p on \mathcal{T} , which is the set of all points on \mathcal{T} from which water flows to p . In other words, it is the set of points whose trickle path contains p . Unfortunately, the combinatorial complexity of these structures can be quite high. For instance, De Berg *et al.* [3] showed that there are terrains of n triangles on which certain trickle paths cross $\Theta(n)$ triangles each $\Theta(n)$ times, resulting in a path of complexity $\Theta(n^2)$. McAllister [1] and McAllister and Snoeyink [2] showed that the total complexity of the watershed boundaries of all local minima can be $\Theta(n^3)$. By slightly modifying the construction provided by De Berg *et al.* we can in fact show that the boundary of a single watershed can have $\Theta(n^3)$ complexity. For *fat terrains*, where the angles of the terrain triangles are lower-bounded by a constant, the situation is somewhat better: here the worst-case complexity of a single path of steepest ascent/descent is $\Theta(n)$ [4]. The complexity of a watershed, however, can still be $\Theta(n^2)$.

It is not always necessary, however, to explicitly compute the structure of interest. For example, it may be sufficient to compute only the surface area of the watershed of a given local minimum, rather than an

*TU Eindhoven, mberg@win.tue.nl.

[†]TU Eindhoven, cs.herman@haverkort.net.

[‡]TU Eindhoven, ctsirogi@win.tue.nl. MdB and CPT were supported by the Netherlands' Organisation for Scientific Research (NWO) under project no. 639.023.301.

¹This can of course be ensured by a small perturbation of the elevations of the terrain vertices, but even small perturbations may have undesirable effects on the water flow. How to deal with horizontal triangles is therefore an important research topic in itself.

explicit description of the boundary of watershed itself. The question thus arises: is it possible to compute the surface area of the watershed of a given local minimum without explicitly computing the watershed itself, thereby avoiding a worst-case running time of $\Theta(n^3)$?

A closely related structure on a terrain is the so-called *surface network* of \mathcal{T} . This is the graph whose nodes are the critical points (local minima and maxima, and saddle points) of \mathcal{T} and whose arcs are obtained by tracing paths of steepest ascent and descent from the saddle points to the local extrema [12, 6]. This graph has linear size, but explicitly tracing the paths of steepest ascent and descent from the saddle vertices results in a procedure that is very inefficient in the worst case. The surface network is related to the so-called Morse-Smale complex [10, 15], which has not only been used in GIS applications [6] but also for example in molecular shape analysis [5] (although here the domain is no longer in \mathbb{R}^2). The Morse-Smale complex has been originally defined for smooth surfaces, and in fact transferring the concept to the piecewise linear case—for example, to polyhedral terrains—is not straightforward. (The main difficulty lies in the fact that a path of steepest descent can intersect a path of steepest ascent.) Several methods have been proposed to define and compute Morse-Smale complexes on piecewise linear surfaces; see the paper by Comić *et al.* [6] for an overview. In one way or another, these methods are always based on following certain paths of steepest descent/ascent. Sometimes an approximation is computed: the watershed of a point p (which is a cell of the unstable Morse-Smale complex), for instance, would then be represented as the union of a certain subset of the terrain triangles. Existing algorithms of this type, however, are not exact: they are not guaranteed to find exactly those triangles for which all points have a trickle path containing p .

Our results. Inspired by the above, we study the problem of implicitly tracing paths of steepest descent or ascent on a polyhedral terrain \mathcal{T} with n vertices. First, in Section 2, we give an $O(n \log n)$ algorithm that finds out where the trickle path of a given point p ends, without constructing the actual path (which would take $\Theta(n^2)$ time in the worst case). Our algorithm can also report all the triangles crossed by the path in the same amount of time. Then, in Section 3, we turn our attention to following multiple paths of steepest descent (or steepest ascent) simultaneously. We develop a mechanism for implicitly tracing n such paths in $O(n \log n)$ time in total. Using our mechanism, we can compute several of the flow-related structures mentioned above. In particular, we can in $O(n \log n)$

time:

- compute for each local minimum p of \mathcal{T} the set of terrain triangles that lie completely in the watershed of p ;
- compute the surface network of \mathcal{T} .

We also show how we can in $O(n^2)$ time compute the exact surface area of all watersheds of \mathcal{T} .

Terminology and notation. For a terrain \mathcal{T} we denote the set of its edges by E , and the set of its vertices by V . Edges in E are defined to be open, that is, they do not include their endpoints. For any point p we denote its z -coordinate by $z(p)$. For an edge $e \in E$ incident to a triangle t we call e an *out-edge* of t if e receives water from the interior of t through the direction of steepest descent. Otherwise we call e an *in-edge* of t . We call e a *valley* edge if e is an out-edge for both of its incident triangles, we call e a *transfluent* edge if e is an out-edge for only one incident triangle, and we call e a *ridge* edge if it is an in-edge for both of its incident triangles.

2 Computing the triangles crossed by a trickle path

Let \mathcal{T} be a terrain with n triangles, and let p be the point for which we want to compute the point where *trickle*(p) ends. As we only want to find where *trickle*(p) ends, we do not want to explicitly compute all intersection points between *trickle*(p) and the terrain edges. To avoid this, each time we encounter a sequence of edges that we crossed before, we jump to the first edge that we have not encountered so far. We can detect features that we already crossed, because we *mark* them the first time we hit them. Next we show how to do the above.

Define an *EV-sequence* to be the (ordered) sequence of terrain edges and vertices crossed by some path on \mathcal{T} . For a point $q \in \text{trickle}(p)$, let $\mathcal{S}(q)$ denote the EV-sequence crossed by the part of *trickle*(p) from p to q . Consider a point $q \in \text{trickle}(p)$ and let $\mathcal{S}(q) = f_1 f_2 \cdots f_k$. Let j be the largest index such that the feature f_j occurs at least twice in $\mathcal{S}(q)$, and let i be the largest index with $i < j$ such that $f_i = f_j$. We call $f_i f_{i+1} \cdots f_j$ the *last cycle* of $\mathcal{S}(q)$, and we call $f_{j+1} \cdots f_k$ the *last chain* of $\mathcal{S}(q)$; see Fig. 1(i). We need the following lemma.

LEMMA 2.1. *Let f be a feature in $\mathcal{S}(q)$ that only occurs before the last cycle of $\mathcal{S}(q)$. Then *trickle*(q) cannot cross f .*

Proof. Let $\mathcal{S}(q) = f_1, \dots, f_k$ and let f_i, \dots, f_j be the last cycle of $\mathcal{S}(q)$. Let $e = f_i = f_j$ and let r_i and

r_j be the intersection points of $trickle(p)$ with e that correspond to f_i and f_j , respectively. Let $\pi(p, r_i)$ be the part of $trickle(p)$ from p to r_i and let $\pi(r_i, r_j)$ be the part of $trickle(p)$ between r_i and r_j . Note that $trickle(q) \subset trickle(r_j)$. Define $P := \pi(r_i, r_j) \cup \overline{r_i r_j}$. Then P is the boundary of a simple polygon—see Fig.1(i), where this polygon is depicted grey. Since trickle-paths cannot self-intersect and e can be crossed in only one direction by a trickle path, one of the paths $\pi(p, r_i)$ and $trickle(r_j)$ lies completely inside P while the other lies completely outside P . This implies that a feature intersecting $\pi(p, r_i)$ can only intersect $trickle(q)$ if that feature intersects $\pi(r_i, r_j)$ and, hence, occurs in the last cycle.

Now imagine tracing $trickle(p)$ and suppose we reach an edge e that we already crossed before. Let q be the point on which $trickle(p)$ crosses e this time. After crossing e again, we may cross many more edges that we already encountered. Our goal is to skip these edges and immediately jump to the next new edge on the trickle path. By Lemma 2.1, the already crossed edges are either in the last cycle or in the last chain of $S(q)$. In fact, since q lies on an already crossed edge, the last chain is empty and so the edges we need to skip are all in the last cycle. Thus we store the last cycle in a data structure T_{cycle} —we call this structure the *cycle tree*—that allows us to jump to the next new edge by performing a query $FindExit(T_{\text{cycle}}, q)$. More precisely, if $\mathcal{C} = f_i, \dots, f_k$ denotes the cycle stored in T_{cycle} and q is a point on f_i , then $FindExit(T_{\text{cycle}}, q)$ reports a pair $(f_{\text{exit}}, q_{\text{exit}})$ such that f_{exit} is the first feature crossed by $trickle(q)$ that is not one of the features in \mathcal{C} and q_{exit} is the point where $trickle(q)$ hits f_{exit} . The cycle tree stores the last cycle encountered so far in the trickle path, thus we have to update this tree according to the changes in the last cycle.

Besides the cycle tree we also maintain a list L which stores the last chain of $S(q)$; these edges may have to be inserted into T_{cycle} later on. This leads to the following algorithm.

Algorithm *ExpandTricklePath*(T, p)

Input: A triangulated terrain \mathcal{T} and a point p on the surface of \mathcal{T} .

Output: The point where $trickle(p)$ ends and the edges crossed by this path.

1. Initialize an empty cycle tree T_{cycle} and an empty list L , and set $q := p$. If q lies on a feature f , then insert f into L .
2. **while** q is not a local minimum and flow from q does not exit the terrain
3. **do** \triangleright Invariant: T_{cycle} stores the last cycle of $S(q)$, \triangleright and L stores its last chain.
4. Let f be the first feature that $trickle(q)$ crosses after leaving from q , and let q' be the point where

5. $trickle(q)$ hits f .
6. $q := q'$
7. **if** f is not marked
8. **then** Mark f and append f to L .
9. **else** Update T_{cycle} and empty L .
10. Set $(f_{\text{exit}}, q_{\text{exit}}) := FindExit(T_{\text{cycle}}, q)$, mark f_{exit} , and set $q := q_{\text{exit}}$.
11. Append f_{exit} to L (which is currently empty) and update T_{cycle} .
11. **return** q .

It is easy to see that the invariant holds after step 1 and that it is maintained correctly, assuming T_{cycle} is updated correctly in steps 8 and 10. This implies the correctness of the algorithm. Next we describe how to implement the cycle tree.

Consider an EV-sequence \mathcal{S} without cycles and assume that there is some trickle path that crosses the features in \mathcal{S} in the given order. Let $\text{first}(\mathcal{S})$ denote the first feature of \mathcal{S} and let $\text{last}(\mathcal{S})$ denote its last feature. We define the *trickle function* $F_{\mathcal{S}} : \text{first}(\mathcal{S}) \rightarrow \text{last}(\mathcal{S})$ of the sequence \mathcal{S} as follows. If the trickle path of a point $q \in \text{first}(\mathcal{S})$ follows the sequence \mathcal{S} all the way up to $\text{last}(\mathcal{S})$, then $F_{\mathcal{S}}(q)$ is the point on $\text{last}(\mathcal{S})$ where $trickle(q)$ hits $\text{last}(\mathcal{S})$. If, on the other hand, $trickle(q)$ exits \mathcal{S} before reaching $\text{last}(\mathcal{S})$, then $F_{\mathcal{S}}(q)$ is undefined. We denote the domain of $F_{\mathcal{S}}$ (the part of $\text{first}(\mathcal{S})$ where $F_{\mathcal{S}}$ is defined) by $\text{Dom}(F_{\mathcal{S}})$, and we denote the image of $F_{\mathcal{S}}$ by $\text{Im}(F_{\mathcal{S}})$. Since we assumed there is a trickle path crossing \mathcal{S} , both $\text{Dom}(F_{\mathcal{S}})$ and $\text{Im}(F_{\mathcal{S}})$ are non-empty. Fig. 1(ii) illustrates these definitions. Note that $\text{Im}(F_{\mathcal{S}})$ is a single point when one of the features in \mathcal{S} is a vertex. The following lemma follows from elementary geometry.

LEMMA 2.2. (i) *The function $F_{\mathcal{S}}(q)$ is a linear function, and $\text{Dom}(F_{\mathcal{S}})$ and $\text{Im}(F_{\mathcal{S}})$ are intervals of $\text{first}(\mathcal{S})$ and $\text{last}(\mathcal{S})$, respectively.* (ii) *Suppose an EV-sequence \mathcal{S} is the concatenation of EV-sequences \mathcal{S}_1 and \mathcal{S}_2 . Then $F_{\mathcal{S}}$ can be computed from $F_{\mathcal{S}_1}$ and $F_{\mathcal{S}_2}$ in $O(1)$ time.*

Now consider an EV-sequence $\mathcal{S}(q) = f_1 \dots f_k$ and let $\mathcal{C} = f_i, \dots, f_j$ be the last cycle of $\mathcal{S}(q)$. The cycle tree T_{cycle} for \mathcal{C} is a balanced binary tree, defined as follows.

- The leaves of T_{cycle} store the features f_i, \dots, f_{j-1} in order.
- For an internal node ν , let $lc[\nu]$ and $rc[\nu]$ denote its left and right child, respectively. Let $\mathcal{S}[\nu]$ denote the subsequence of \mathcal{C} consisting of the features stored in the leaves below ν . Furthermore, let $\text{first}[\nu]$ and $\text{last}[\nu]$ denote the features stored in the leftmost and rightmost leaf below ν , respectively.

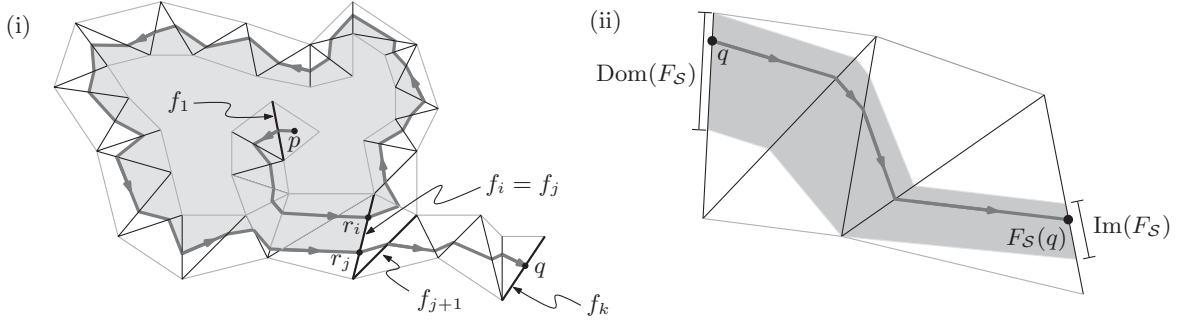


Figure 1: (i) The last cycle of the EV-sequence $\mathcal{S}(q)$ is f_i, \dots, f_j , and the last chain is f_{j+1}, \dots, f_k . (ii) The trickle function.

Then ν stores the trickle function $F_{\mathcal{S}[\nu]}$, and the trickle function $F_{\mathcal{S}'[\nu]}$, where $\mathcal{S}'[\nu]$ is the sequence $f_\nu f'_\nu$ with $f_\nu = \text{last}[lc[\nu]]$ and $f'_\nu = \text{first}[rc[\nu]]$.

LEMMA 2.3. *The function $\text{FindExit}(T_{\text{cycle}}, q)$ can be implemented to run in $O(\log |\mathcal{C}|)$ time, where $|\mathcal{C}|$ is the length of the cycle stored in T_{cycle} .*

Proof. Imagine following $\text{trickle}(q)$, starting at f_i , the first feature in \mathcal{C} . We will cross a number of features of \mathcal{C} , until we exit the cycle. (We must exit the cycle before returning to f_i again, because a trickle path cannot cross the same sequence twice without encountering another feature in between [3].) Let f^* be the feature of \mathcal{C} that we cross just before exiting. We can find f^* in $O(\log |\mathcal{C}|)$ time by descending down T_{cycle} as follows.

Suppose we arrive at a node ν ; initially ν is the root of T_{cycle} . We will maintain the invariant that f^* is stored in a leaf below ν . We will make sure that we have the point q_ν where $\text{trickle}(q)$ crosses $\text{first}[\nu]$ available; initially $q_\nu = q$. When ν is a leaf we have found f^* , otherwise we have to decide in which subtree to recurse. The feature f^* is stored in the right subtree of an internal node ν if and only if

- (i) $q_\nu \in \text{Dom}(F_{\mathcal{S}[lc[\nu]]})$, which means $\text{trickle}(q_\nu)$ completely crosses $\mathcal{S}[lc[\nu]]$, and
- (ii) $F_{\mathcal{S}[lc[\nu]]}(q_\nu) \in \text{Dom}(F_{\mathcal{S}'[\nu]})$, meaning $\text{trickle}(q_\nu)$ reaches $\text{first}[rc[\nu]]$ after crossing $\mathcal{S}[lc[\nu]]$.

If these two conditions are met, we set $\nu := rc[\nu]$ and $q_\nu := F_{\mathcal{S}'[\nu]} \circ F_{\mathcal{S}[\nu]}(q_\nu)$, otherwise we set $\nu := lc[\nu]$.

Once we have found f^* and the point q^* where $\text{trickle}(q)$ crosses f^* , we can compute the exit edge e_{exit} and point q_{exit} by inspecting the relevant triangle t incident to f^* : we just have to compute where the path of steepest descent from q^* exits t .

It remains to explain how to update T_{cycle} . First consider step 8 of *ExpandTricklePath*. Suppose that, just before q reaches f , we have $\mathcal{S}(q) = f_1 \cdots f_k$. Let

$f_i \cdots f_j$ be the last cycle of $\mathcal{S}(q)$ (which is stored in T_{cycle}) and $f_{j+1} \cdots f_k$ its last chain (which is stored in L). We know that f has been crossed before. By Lemma 2.1 this implies $f = f_m$ for some $m \geq i$. We distinguish two cases.

- If $m > j$, then f occurs in the last chain and, hence, in L . Now after crossing f the last cycle becomes $f_m \cdots f_k f$. So updating T_{cycle} amounts to first emptying T_{cycle} , and then constructing a new cycle tree on $f_m \cdots f_k f$, which can be done by a bottom-up procedure in $O(|L|)$ time.
- If $i \leq m \leq j$ then f occurs in the last cycle. Then after crossing f the last cycle becomes $f_m \cdots f_j f_{j+1} \cdots f_k f$. (In the special case that $m = j$, we in fact have $f_i = f_j = f$ and the last cycle becomes $f_j f_{j+1} \cdots f_k f$.) We can now update T_{cycle} by deleting the features $f_1 \cdots f_{m-1}$, and inserting the features $f_{j+1} \cdots f_k$. (Recall that the last feature of a cycle is not stored in the cycle tree.) Inserting and deleting elements from an augmented balanced binary tree T_{cycle} can be done in logarithmic time in a standard manner.

Next consider the updating of T_{cycle} in step 10. Let $f_i \cdots f_j$ be the last cycle before step 9, where we jump to the first new feature crossed by the trickle path. Let f_m be the last feature we cross before we exit the cycle, that is, the feature f^* in the proof of Lemma 2.3. Then after the jump, the last cycle becomes $f_m \cdots f_{j-1} f_i \cdots f_m$. (Essentially, the cycle does not change, but its starting feature changes.) Thus, to update T_{cycle} we have to split T_{cycle} between f_{m-1} and f_m into two cycle trees T_{cycle}^1 and T_{cycle}^2 , then merge these cycle trees again but this time in the opposite order (that is, putting T_{cycle}^1 to the right of T_{cycle}^2 instead of to its left). Splitting and merging can be done in logarithmic time, if we use a suitable underlying tree such as a red-black tree. We obtain the following theorem.

THEOREM 2.1. *Let \mathcal{T} be a terrain with n triangles and let p a point on the surface of \mathcal{T} . Algorithm *ExpandTricklePath*(\mathcal{T}, p) traces the trickle path of p in time $O(n \log C_{\max})$, where C_{\max} is the length of the longest cycle in the EV-sequence of *trickle*(p).*

3 Expanding multiple paths simultaneously

Our main interest is to design an efficient algorithm that can expand a collection of $\Theta(n)$ paths simultaneously. Our next step towards this direction is to present how we can expand efficiently a collection of paths that emanate from the same point. We thus design a subroutine that expands implicitly *upnet*(p), the *up-network* of a terrain point p ; this is the set of all points on \mathcal{T} reachable by a path of locally steepest ascent from p . Here the directions of locally steepest ascent are defined as follows. For a point $q \in \mathcal{T}$, let $\mathcal{B}_\epsilon(q)$ be the ball of infinitesimal radius centered at q . Let \mathcal{M}_ϵ be the set of points of locally maximum elevation in $\mathcal{B}_\epsilon(q) \cap \mathcal{T}$ whose elevation is greater than $z(q)$. Then the *directions of locally steepest ascent* at q are given by the vectors from q to each point in \mathcal{M}_ϵ . We are interested in tracing the up-network implicitly since it plays a key role in the construction of the watershed of a given point [1].

Next we describe our subroutine that expands *upnet*(p). We assume that the point p for which we want to compute the up-network is a terrain vertex². An up-network is not necessarily a path; it can split and rejoin at terrain vertices. If we remove all terrain vertices from *upnet*(p), as well as all points that lie on a ridge edge, then *upnet*(p) is broken into several components which we call *up-paths*. We want our subroutine to compute the local maxima and/or the points at the boundary of \mathcal{T} where *upnet*(p) ends.

Our algorithm is a space-sweep algorithm. Let h_z be the horizontal plane at elevation z and let P_z denote the set of up-paths intersecting h_z . We will maintain P_z as we move h_z upwards from p , meanwhile marking all the edges and triangles crossed by any of the up-paths. The difficulty in doing so is that an edge can be crossed by many up-paths and moreover that a single up-path can cross an edge many times.

To overcome these problems we proceed as follows. Let $\text{top}(\pi)$ denote the point up to which we have traced an up-path $\pi \in P_z$; the point $\text{top}(\pi)$ lies on or above h_z , and it will always lie on an edge. We associate π with the edge on which $\text{top}(\pi)$ lies. We denote the set of up-paths associated with an edge e when the sweep plane is at elevation z by $P_z(e)$. Let $P_z(e) = \pi_1, \dots, \pi_k$; here and in the sequel we number the up-paths in $P_z(e)$

²If this is not the case we can just add p in V and re-triangulate the terrain.

in increasing order of the z -coordinate of their tops. During the algorithm we will maintain each set $P_z(e)$ in an augmented tree according to this order. How this *bundle tree* is implemented will be discussed later. The idea is now to jump with each π_i to the first point where it crosses a terrain feature that lies completely above h_z . This feature can be either an edge or a vertex and we call it the *exit feature* of π_i . There can be several up-paths in $P_z(e)$ with the same exit edge. We call the collection of all such up-paths a *bundle* and we will make sure that we can jump with an entire bundle to the common exit edge. To facilitate the jumping, we store the edges currently intersecting h_z in a data structure similar to the cycle tree of the previous section. We call our new structure a *contour structure* and we denote it by D_{contour} . Later we will explain how to implement D_{contour} , but first we return to the overall algorithm.

We define an order on the terrain vertices and edges, that specifies the order in which they are handled. Let $\text{rank}(v)$, the *rank of a vertex* v , be the z -coordinate of v , and let $\text{rank}(e)$, the *rank of an edge* e , be the z -coordinate of the lower endpoint of e . This implies that when we jump from an edge e , we jump to the first feature with rank greater than the elevation of h_z . For two features f_1, f_2 we define $f_1 \prec f_2$ if either $\text{rank}(f_1) < \text{rank}(f_2)$, or f_1 is a vertex and f_2 is an edge and $\text{rank}(f_1) = \text{rank}(f_2)$. We extend this partial order to a total order in an arbitrary manner. An event queue will store vertices and edges in \prec -order. The global algorithm is now as follows. (When we write “insert this feature into Q ” we actually first check whether the feature is already present in Q and only do the insertion when this is not the case.)

Algorithm *ExpandUpNetwork*(\mathcal{T}, p)

Input: A triangulated terrain \mathcal{T} and a vertex p of \mathcal{T} .

Output: The local maxima/boundary points on \mathcal{T} where *upnet*(p) ends and the edges crossed by *upnet*(p).

1. Set $z := z(p)$, initialize D_{contour} with all edges intersecting h_z , and create an event queue Q storing only p .
2. **while** Q is not empty
3. **do** Remove from Q the feature f that is minimal in the \prec -order.
4. Set $z := \text{rank}(f)$ and update D_{contour} .
5. **if** f is a vertex, v
6. **then if** v is a local maximum
7. **then** output v .
8. **else** \triangleright Expand v :
9. For each up-path π starting at v , let e_π be the first edge hit by π . If e_π is incident to v then insert the other vertex w of e_π into Q . If e_π is not incident to v , then add π to $P(e_\pi)$, insert e_π into Q , and mark and report e_π .

10. **if** f is an edge, e
11. **then if** e is an edge on the boundary of \mathcal{T}
12. **then** Output the tops of the paths stored in $P_z(e)$.
13. **else** \triangleright Jump from e :
14. Split $P_z(e)$ into bundles. For each bundle b , proceed as follows: Let $f_{\text{exit}}(b)$ be the first feature crossed by b that lies completely above the sweep plane h_z . Mark and report any unmarked edges crossed by b . Insert $f_{\text{exit}}(b)$ into Q , and if $f_{\text{exit}}(b)$ is an edge then add b to $P_z(f_{\text{exit}}(b))$.

The correctness of the algorithm can be seen as follows. By induction we can argue that all up-paths are created. When we trace the first link of an up-path (step 9) we mark the crossed edge, and when we extend an up-path as part of a bundle (step 14) we mark all newly crossed edges. Furthermore, an up-path continues to be extended until it ends. Hence, all edges crossed by $upnet(p)$ are marked and all reached local maxima and boundary points are reported if the steps are implemented correctly.

Before we explain the various steps of the algorithm in more detail, we discuss some properties of the paths and bundles generated by the algorithm. We start with the next basic lemma.

LEMMA 3.1. *Let e_{in} and e_{out} be an in-edge and an out-edge, respectively, of a terrain triangle t . Let $p, q \in e_{\text{out}}$ with $z(p) > z(q)$, and let $p', q' \in e_{\text{in}}$ be such that $\overline{pp'}$ and $\overline{qq'}$ are parallel to the direction of steepest descent. Then $z(q') > z(p')$ if and only if the highest vertex of e_{out} is the lowest vertex of e_{in} .*

Proof. Since $z(p) > z(q)$ we know that p lies closer than q to the vertex incident to e_{out} with the highest elevation. Let v' be this vertex.

Let v be the vertex incident to e_{out} and e_{in} . We consider two cases:

- $v = v'$: Since $\overline{pp'}$ and $\overline{qq'}$ are parallel, $dist(p, v) < dist(q, v)$ implies that $dist(p', v) < dist(q', v)$. But then we can only have $z(q') > z(p')$ if v is the lowest vertex of e_{in} .
- $v \neq v'$: Now q lies closer to v on e_{out} and as $\overline{pp'}$ and $\overline{qq'}$ are parallel then q' lies on e_{in} closer to v than p' . Let v'' be the other vertex incident to e_{in} . v'' has higher elevation than v' otherwise v and v'' are the vertices of lowest elevation in t and e_{in} cannot be an in-edge. p' lies on e_{in} closer to v'' than q' so p' has a higher elevation than q' .

Consider a point q on an up-path π . We denote the part of π up to q by $tail_{\pi}(q)$. We define $rank(tail_{\pi}(q))$ to be the maximum rank of any edge crossed by $tail_{\pi}(q)$.

LEMMA 3.2. *Let π and π' be two up-paths that cross the same transfluent edge e , and let q and q' be the points where they cross e . If $rank(tail_{\pi}(q)) > rank(tail_{\pi'}(q'))$ then $z(q) > z(q')$.*

Proof. Assume for a contradiction that $z(q') > z(q)$. Imagine tracing $tail_{\pi}(q)$ and $tail_{\pi'}(q')$ downwards as long as they follow the same EV-sequence. Let $\mathcal{S} = e_1, \dots, e_k$ be this EV-sequence. Note that $e_1 = e$. Let q_i and q'_i denote the points where $tail_{\pi}(q)$ and $tail_{\pi'}(q')$ cross e_i , respectively—see Fig. 2(a).

Consider two consecutive edges e_i and e_{i+1} . Then the lowest vertex incident to e_i cannot be the highest vertex incident to e_{i+1} . Otherwise, e_i has a higher rank than any other edge following it, contradicting $rank(tail_{\pi}(q)) > rank(tail_{\pi'}(q'))$. The assumption $z(q') > z(q)$ thus implies by Lemma 3.1 that $z(q'_i) > z(q_i)$ for all $1 \leq i \leq k$.

Let t be the triangle entered by $tail_{\pi}(q)$ and $tail_{\pi'}(q')$ after crossing e_k , and let v be the vertex of t not incident to e_k . We assume for simplicity that neither $tail_{\pi}(q)$ nor $tail_{\pi'}(q')$ crosses v ; adapting the argument is straightforward. Let e_{k+1} be the edge of t incident to the two lowest vertices of t . Note that v is one of these two vertices. Since $z(q'_k) > z(q_k)$, we know that $tail_{\pi}(q)$ crosses e_{k+1} . Let q_{k+1} denote the point where this crossing takes place. Since the endpoints of e_{k+1} are the two lowest vertices of t , either e_k or e'_{k+1} (the third edge of t) lies strictly above the interior points of e_{k+1} . But then any edge crossed by $tail_{\pi}(q_{k+1})$ has a lower rank than either e_k or e'_{k+1} , and the latter two edges are crossed by $tail_{\pi'}(q')$. Hence, $rank(tail_{\pi'}(q')) \geq rank(tail_{\pi}(q))$, and we reach a contradiction.

Lemma 3.2 is used to prove that bundles cannot interleave, so that splitting a set $P_z(e)$ into bundles and adding these bundles to the sets $P_z(f_{\text{exit}})$ of their respective exit features can be done efficiently. Next we make this non-interleaving property precise.

Suppose that the algorithm jumps from an edge e in step 14. Note that two or more bundles in $P_z(e)$ may first follow the same edge sequence for some time before they split. For an edge e' , we denote by $B_{\mathcal{S}}(e, e')$ the set of bundles that follow the same edge-sequence \mathcal{S} from e to e' when $P_z(e)$ is processed. We call $B_{\mathcal{S}}(e, e')$ a *multi-bundle*. The tops of the up-paths when they reach e' after traversing \mathcal{S} are called the tops of the multi-bundle.

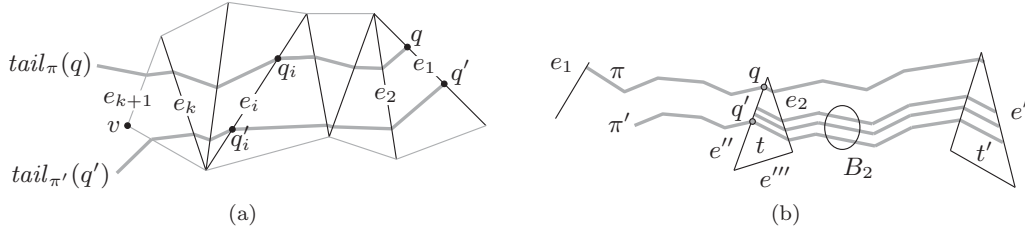


Figure 2: (a) Illustration for the proof of Lemma 3.2. (b) Illustration for the proof of Lemma 3.3.

LEMMA 3.3. (i) Let b be a bundle of $P_z(e)$. Then the paths in b are consecutive in $P_z(e)$.

(ii) Let $B_1 := B_{S_1}(e_1, e')$ and $B_2 := B_{S_2}(e_2, e')$ be two multi-bundles crossing the same transfluent edge e' . Then B_1 and B_2 do not interleave on e' , that is, there is a point on e' separating the tops of B_1 from the tops of B_2 .

Proof. To prove part (i), let π_1 and π_2 be the two outermost up-paths in b . Since up-paths don't cross, any up-path starting in between π_1 and π_2 follows the same edge-sequence as π_1 and π_2 up to $f_{\text{exit}}(b)$ and, hence, is an up-path in b .

To prove part (ii), assume without loss of generality that e_1 was handled before e_2 . Thus $\text{rank}(e_1) \leq \text{rank}(e_2)$. We will show that no up-path $\pi \in B_1$ can separate B_2 , that is, $\text{top}(\pi)$ cannot lie in between the tops of the outermost paths π_1 and π_2 in B_2 . Showing that no up-path in B_2 can separate B_1 can be done in a similar, yet not symmetric, way.

If $\text{rank}(e_1) < \text{rank}(e_2)$, then according to Lemma 3.2 the tops of B_2 lie above $\text{top}(\pi)$, so π does not separate B_2 .

Now consider the case³ $\text{rank}(e_1) = \text{rank}(e_2)$. Let z be the z -coordinate corresponding to this rank (so h_z is the plane through the lower endpoints of e_1 and e_2). Since e' is a transfluent edge, the paths in B_2 and π cross the same triangle t' before encountering e' . We can assume that B_2 and π enter t' through the same edge, as in Fig. 2(b), otherwise π surely cannot separate B_2 . Now imagine following π backwards from e' as long as it follows the same edge-sequence as B_2 . If π lies in between π_1 and π_2 , the path π must follow the same edge sequence until either e_1 or e_2 , whichever comes first. In fact, we can argue that e_2 must come first—otherwise, when π jumped to e_1 it would actually have stopped at e_2 . We claim that π crosses e_2 above any of

³The argument for the case $\text{rank}(e_1) = \text{rank}(e_2)$ also applies when $e_1 = e_2$. This special case may happen when an up-path traverses some edges intersecting h_z in a cyclic way. It is then possible that some up-paths in $P_z(e_1)$ cross a sequence S' of edges before hitting e' , while others first traverse a cycle of all edges intersecting h_z , before crossing S' and hitting e' .

the paths $\pi_i \in B_2$, which then implies part (ii) of the lemma. Let t be the triangle that π and the paths in B_2 cross just before e_2 and let e'', e''' be the other two edges incident to t . Suppose π enters t through e'' , as in Fig. 2(b). There are two cases.

- First consider a path $\pi' \in B_2$ that also crosses e'' when it jumped to e_2 . Let q be the point where π crosses e'' , and let q' be the intersection point between π' and e'' . Since $tail_\pi(q)$ crosses e_1 and $tail_{\pi'}(q')$ does not cross any edge with rank higher or equal to $\text{rank}(e_2)$ we have that $\text{rank}(tail_\pi(q)) > \text{rank}(tail_{\pi'}(q'))$. By Lemma 3.2 we get that q lies above q' on e'' . Thus the top of π on the forthcoming encounter with e_2 also lies above the top of π' on e_2 according also to Lemma 3.1, as claimed.
- Now consider a path π' that did not reach e_2 through e'' , but through edge e''' . The lower vertex of e_2 is intersected by h_z , and e'' or a vertex of e'' is intersected by h_z since there is a path from e_1 that crosses e'' , namely π , before hitting an exit feature. Then e''' must either lie completely below or above h_z , otherwise t is horizontal. Since π' crosses e''' before ever hitting e_2 then e''' can only lie below h_z . The fact that e''' lies below h_z and π crosses e'' above h_z implies that top of π on e_2 lies above the top of π' on e_2 , as claimed.

We now return to the algorithm, and show how it can be implemented efficiently.

The contour structure. Consider a situation where h_z does not contain a vertex. Then $h_z \cap \mathcal{T}$ consists of a number of simple, closed, polygonal curves, called *contours*. Let C_1, C_2, \dots be the contours, and let \mathcal{S}_i denote the (cyclic) edge sequence corresponding to C_i . We give each edge $e \in \mathcal{S}_i$ that can be hit in clockwise direction by an up-path a label CW, and each edge that can be hit in counterclockwise direction a label CCW. Note that ridge edges get two labels, transfluent edges get one label, and valley edges get no label. We partition \mathcal{S}_i into maximal subsequences \mathcal{S}_i^j of edges with the

same label; we call them CW-subsequences and CCW-subsequences depending on their common label. A ridge edge will be part of two subsequences (one CW-subsequence, and one CCW-subsequence), a transfluent edge will be part of one subsequence, and a valley edge will not be part of any subsequence.

Each subsequence \mathcal{S}_i^j will be stored in an augmented tree $D(\mathcal{S}_i^j)$, which is the same as the cycle tree of the previous section, except for the following. First, the trickle functions should be reversed, meaning that they should specify how an up-path (rather than a trickle path) can traverse a sequence. Second, each internal node $\nu \in D(\mathcal{S}_i^j)$ stores a boolean *unmarked* $[\nu]$ indicating whether any of the edges stored in the subtree rooted at ν is still unmarked. This way, when we jump over some edges of \mathcal{S}_i^j to the first encountered edge above the sweep plane, we can mark all unmarked edges in logarithmic time per unmarked edge.

Inserting an edge or deleting an edge from the contour can be done in logarithmic time. Moreover, we can merge and split any of the structures $D(\mathcal{S}_i^j)$ in logarithmic time; this is necessary when we hit a saddle vertex, for instance, since then two contours split.

The bundle tree. Consider an edge e stored in the event queue with $P_z(e) = \pi_1, \dots, \pi_k$. Let $\text{top}_{s_z}(e) = \tau_1, \dots, \tau_k$ be the tops of these up-paths on e . The bundle tree $T_{\text{bundle}}(e)$ stored with e is a balanced binary tree that we define as follows.

- The leaves of $T_{\text{bundle}}(e)$ store the tops $\tau_2, \dots, \tau_{k-1}$ in order. Let $\text{dist}(\tau_i, \tau_j)$ denote the distance between the tops τ_i and τ_j . A leaf node ν that stores the top τ_r also stores the ratio $\frac{\text{dist}(\tau_r, \tau_{r+1})}{\text{dist}(\tau_{r-1}, \tau_r)}$. This ratio remains the same when we expand the bundle upwards as long as the two paths incident to π_r follow the same sequence of edges.
- For an internal node ν , let $\text{first}[\nu]$ and $\text{last}[\nu]$ denote the tops stored in the leftmost and rightmost leaf below ν , respectively. Let $\text{pred}[\nu]$ be the top that comes before $\text{first}[\nu]$, and $\text{suc}[\nu]$ the top that follows $\text{last}[\nu]$. Then ν stores the ratios $r_1[\nu] = \frac{\text{dist}(\text{first}[\nu], \text{last}[\nu])}{\text{dist}(\text{pred}[\nu], \text{first}[\nu])}$ and $r_2[\nu] = \frac{\text{dist}(\text{last}[\nu], \text{suc}[\nu])}{\text{dist}(\text{pred}[\nu], \text{first}[\nu])}$.
- We store with $T_{\text{bundle}}(e)$ the coordinates of τ_1 and τ_k , and $\text{dist}(\tau_1, \tau_2)$ and $\text{dist}(\tau_{k-1}, \tau_k)$.

Updates on a bundle tree, and merging and splitting, can be done in logarithmic time.

Next we show how to compute, given a point $p \in e$, which tops of $P_z(e)$ lie on each side of p . In other words, we have to determine the maximum j such that $\tau_j \in P_z(e)$ lies below p . We start by setting $\nu := \text{root}(T_{\text{bundle}}(e))$. We maintain the invariant that

τ_j is stored in a leaf under ν , or $j = 1$, or $j = k$. Define $d := \text{dist}(\text{pred}[\nu], p)$ and $\delta := \text{dist}(\text{pred}[\nu], \text{first}[\nu])$. Initially we have $d = \text{dist}(\tau_1, p)$ and $\delta = \text{dist}(\tau_1, \tau_2)$. Also define $\delta_1 := \text{dist}(\text{pred}[\nu], \text{last}[lc[\nu]])$ and $\delta_2 := \text{dist}(\text{pred}[\nu], \text{first}[rc[\nu]])$. Note that $\delta_1 = \delta \cdot (1 + r_1(lc[\nu]))$ and $\delta_2 = \delta_1 + \delta \cdot r_2(lc[\nu])$. Using the information stored in $T_{\text{bundle}}(e)$, we can maintain $d, \delta, \delta_1, \delta_2$ in constant time as we descend in $T_{\text{bundle}}(e)$. To determine to which child to proceed, we distinguish three cases:

- if $d < \delta_1$ then τ_j is stored in a leaf below $lc[\nu]$ or it is τ_1 , and so we set $\nu := lc[\nu]$.
- if $\delta_1 < d < \delta_2$ then τ_j is $\text{last}(lc[\nu])$, and we are done.
- if $\delta_2 < d$ then τ_j is stored in a leaf below $rc[\nu]$ or it is τ_k , and so we set $\nu := rc[\nu]$.

The process to find τ_j takes logarithmic time. After finding τ_j , we can split $T_{\text{bundle}}(e)$ in logarithmic time into a bundle tree T_{bundle}^1 for π_1, \dots, π_j and a bundle tree T_{bundle}^2 for π_{j+1}, \dots, π_k .

Details of the algorithm. Now that we have described D_{contour} and T_{bundle} , we can explain steps 4, 9, and 14 of *ExpandUpNetwork* in more detail.

Step 4: Updating the contour structure. Whenever we move the sweep plane h_z upward to some new elevation z^* , we have to update D_{contour} : we must delete all edges whose top endpoint now lies on or below h_z , and we must insert all edges whose bottom endpoint lies on h_z . Updates can be done in $O(\log n)$, so in total they take $O(n \log n)$ time.

Step 9: expanding a vertex v . The number of up-paths emanating from v is at most the degree of v . Each up-path may require updating Q and then updating some set $P_z(e_\pi)$, which takes $O(\log n)$ time. Hence, the vertex expansions take $O(n \log n)$ time in total.

Step 14: jumping from an edge e . To split $P_z(e)$ into bundles and jump with each bundle to its exit edge, we proceed as follows. Let $P_z(e) = \pi_1, \dots, \pi_k$, let $\tau_1, \dots, \tau_k \in e$ be the tops of these up-paths, and let \mathcal{S}_i^j denote the subsequence (in the current set of contours) containing e . Recall that $\text{FindExit}(D(\mathcal{S}_i^j), q)$ reports, given a point q on an edge e intersecting the sweep plane h_z , the first feature f_{exit} crossed by q 's up-path that lies completely above h_z .

We first perform a query $\text{FindExit}(D(\mathcal{S}_i^j), \tau_1)$, giving us the exit feature $f_{\text{exit}}(\pi_1)$. Let $F_1 : e \rightarrow f_{\text{exit}}(\pi_1)$ be the function that maps a point $q \in \text{Dom}(F_1)$ to the point on f_{exit} that we reach when we follow an up-path from q . We modify $\text{FindExit}(D(\mathcal{S}_i^j), q)$ such that it also

returns F_1 and $\text{Dom}(F_1)$. Let $T_{\text{bundle}}(e)$ be the tree storing $P_z(e)$. Using $T_{\text{bundle}}(e)$ we determine the largest j such that $\tau_j \in \text{Dom}(F_1)$ and we split $T_{\text{bundle}}(e)$ into two bundle trees T_{bundle}^1 and T_{bundle}^2 , as describe above. By Lemma 3.3(i) the paths π_1, \dots, π_j follow the same edge sequence from e to $f_{\text{exit}}(\pi_1)$, thus forming the first bundle of $P_z(e)$.

We repeat the process with the remainder of $P_z(e)$, now stored in T_{bundle}^2 , until we have determined all the bundles, and for each bundle b its exit feature $f_{\text{exit}}(b)$. For each bundle we then mark all newly crossed edges—this will take $O(\log n)$ per marked edge—and if $f_{\text{exit}}(b)$ is an edge we insert b into $P_z(f_{\text{exit}}(b))$. The latter operation takes $O(\log n)$, since by Lemma 3.3(ii) b does not interleave with the up-paths already stored in $P_z(f_{\text{exit}}(b))$, which means we can add T_{bundle}^1 to $T_{\text{bundle}}(f_{\text{exit}})$ by one splitting and two merging operations. In the case that b hits a ridge edge, we discard b and insert in Q the upper vertex of this edge.

THEOREM 3.1. *Algorithm `ExpandUpNetwork`(\mathcal{T}, p) computes the up-network of a point p on a terrain with n vertices in $O(n \log n)$ time.*

Proof. To prove the time bound, it suffices to argue that there are $O(n)$ bundles generated. When handling an edge e , a bundle is split off when the paths of $P_z(e)$ enter a triangle t through one edge e_1 , but leave t through different edges e_2 and e_3 . Let v be the common vertex of e_2 and e_3 . According to Lemma 3.3 the up-paths of some other set $P_{z'}(e')$ do not interleave with $P_z(e)$ on e_1 , and thus only one multi-bundle can split around v .

Computing steepest descent/ascent paths between critical points, and assigning triangles to watersheds. To construct the surface network graph of \mathcal{T} we need the following more general version of the algorithm `ExpandUpNetwork`. Let P_{saddle} be the set of the $O(n)$ saddle points on \mathcal{T} . Then we can compute the edge-set of the surface network graph in $O(n \log n)$ time; first we initialise the event queue Q in Step 1 of `ExpandUpNetwork` with the points of P_{saddle} . At every saddle point, we expand an up-path of steepest ascent for each *wedge in its upper star* [7]. When we initiate an up-path π , we associate π with the critical point $v[\pi]$ from which the path emanates. An up-path is terminated when it hits a terrain feature that is a vertex or a ridge edge. If this feature is a critical point u we add an edge $(v[\pi], u)$ in the surface network graph, otherwise we propagate the tag $v[\pi]$ to the path of steepest ascent that starts from this feature. To compute the rest of the edges of the graph we use an algorithm `ExpandMultiTricklePath`, which is essentially the same as `ExpandUpNetwork` except that it traces paths downwards

instead of upwards. In the proof of the following theorem we also show how we can compute in $O(n \log n)$ time the triangles contained in the watershed of each local minimum on \mathcal{T} .

THEOREM 3.2. *Let \mathcal{T} be a terrain with n triangles and let P be the set of local minima on \mathcal{T} . We can compute the surface network graph of \mathcal{T} , and assign to each minimum $p \in P$ the triangles that are entirely contained in the watershed of p in $O(n \log n)$ time.*

Proof. Consider a local minimum p of \mathcal{T} and let t be a triangle that is entirely contained in the watershed of p . That means that the trickle path from every point in the interior of t ends in p . For this to happen it can only be that these trickle paths (except maybe a discrete subset of these paths) contain also one or more valley edges. Hence, in order to compute the watershed of p we have to find which valley edges send water to p and then find the triangles that send water to these edges. Thus we proceed as follows.

We use `ExpandMultiTricklePath` to compute for each terrain vertex v the first valley edge hit by `trickle`(v); the algorithm can also compute the points where the trickle paths hit their first valley edge. Now consider a valley edge e whose lowest endpoint sheds water to a local minimum p , and suppose e is the first valley edge hit by the trickle paths of vertices v_1, \dots, v_k . Let $q_i \in e$ be the point where `trickle`(v_i) hits e , and assume $z(q_1) < z(q_2) < \dots < z(q_k)$. Define q_0 and q_{k+1} to be the lowest and highest endpoints of e , respectively. The points q_i for $0 \leq i \leq k$ are the lowest vertices of the *strips* [14] incident to the edge e . A strip is a maximal subset of the terrain surface extending between a segment s of a valley edge and a segment of a ridge edge such that all up-paths starting from s traverse the same sequence of edges. For $0 \leq i \leq k$, let $p_i \in e$ be a point that we pick arbitrarily between q_i and q_{i+1} . Imagine tracing an up-path from each p_i , leaving in the direction where `trickle`(v_i) comes from, until a ridge edge is reached. It can be shown [14] that the triangles containing a point q for which e is the first valley edge hit by `trickle`(q), are precisely the triangles crossed by one of these up-paths. We collect the points p_i, q_i over all valley edges in a set Q , and then apply to Q a modified version of `ExpandUpNetworkTriangle`. In this version of the algorithm we associate each terrain edge e with a tag that indicates if all the trickle paths starting from points on e end at the same local minimum or not.

Let e be a valley edge and let v the lowest vertex incident to e . We tag e with the local minimum where `trickle`(e) ends. We tag each up-path in Q with the same local minimum as the valley edge where it comes from. A triangle t is contained in the watershed of a

local minimum p if and only if the valley and transfluent edges of t are intersected only by up-paths in Q that are tagged with p . If the valley and transfluent edges of t are intersected by up-paths that have different tags then t is a border triangle.

For each bundle tree T_{bundle} that is generated during the sweep we maintain a tag $\text{tag}[T_{\text{bundle}}]$ in the following manner: if a bundle tree T_{bundle} stores up-paths that are all tagged with the same local minimum p then we have $\text{tag}[T_{\text{bundle}}] = "p"$ otherwise this tag has a symbolic value "MULTIPLE". We store also such a tag for every node of T_{bundle} , maintaining this information for each subtree of T_{bundle} . In this way, whenever a new bundle tree T'_{bundle} is generated from splitting or merging other bundle trees then the value of $\text{tag}[T'_{\text{bundle}}]$ can be computed in $O(\log n)$ time.

We also change the fields stored with each node ν of a tree $D(\mathcal{S}_i^j) \in D_{\text{contour}}$ slightly. Instead of a boolean $\text{unmarked}[\nu]$, we store a tag $\text{tag}[\nu]$. If ν is a leaf node, then ν represents an edge crossed by h_z . Let $e[\nu]$ be this edge. The value stored in $\text{tag}[\nu]$ may be of three possible kinds:

- If $e[\nu]$ has not been crossed so far by any up-path then $\text{tag}[\nu]$ stores a symbolic value "NONE".
- If $e[\nu]$ has been crossed only by up-paths that were tagged to the same local minimum p then $\text{tag}[\nu] = "p"$.
- If $e[\nu]$ has been crossed by up-paths that were tagged to different local minima then $\text{tag}[\nu] = "MULTIPLE"$.

For an internal node $\nu \in D(\mathcal{S}_i^j)$ let T_ν be the subtree of $D(\mathcal{S}_i^j)$ with root ν . If all the leaves in T_ν have the same tag value then $\text{tag}[\nu]$ is also set to this value. Otherwise, we distinguish two more cases. If the only tags that appear in the leaves of T_ν are "MULTIPLE" and " p " for only one local minimum p , then $\text{tag}[\nu] = "MULTIPLE \text{ AND } p"$. In any other case $\text{tag}[\nu] = "MIXED"$. Notice that $\text{tag}[\nu] = "MULTIPLE"$ means that each valley edge represented by a leaf node in T_ν has been crossed by up-paths that were tagged with different local minima. However, $\text{tag}[\nu] = "MIXED"$ implies that there are two or more leaf nodes in T_ν that have different flags with each other; for example there may exist a leaf node ν' with tag " p " and a leaf node ν'' with $\text{tag}[\nu''] = "q"$ because $e[\nu']$ was crossed only by up-paths tagged with " p " while $e[\nu'']$ was crossed only by up-paths tagged with " q ".

Suppose that we execute a query $\text{FindExit}(D(\mathcal{S}_i^j), \tau)$ for some up-path τ and for some tree $D(\mathcal{S}_i^j) \in D_{\text{contour}}$ that stores a CW or CCW

subsequence. Let T_{bundle} be the bundle tree that is generated after this query and which stores τ . Let $\nu \in D(\mathcal{S}_i^j)$ be a node encountered during this query such that τ was found to traverse symbolically all the edges stored in the subtree with root ν . We distinguish the following cases:

- If $\text{tag}[\nu] = "NONE"$ then we simply store at $\text{tag}[\nu]$ the tag value of T_{bundle} and we do the same for all the nodes in T_ν .
- If $\text{tag}[\nu] = "MULTIPLE"$ then we do not change anything.
- If $\text{tag}[\nu]$ corresponds to a local minimum p then we check the tag of T_{bundle} ; If also $\text{tag}[T_{\text{bundle}}] = "p"$ then we do not change anything, otherwise we set $\text{tag}[\nu] = "MULTIPLE"$ for all the nodes in T_ν .
- If $\text{tag}[\nu] = "MULTIPLE \text{ AND } p"$ then if $\text{tag}[T_{\text{bundle}}] = "p"$ we do not change anything, otherwise we set $\text{tag}[\nu] = "MULTIPLE"$ and we recurse with the children of ν .
- If $\text{tag}[\nu] = "MIXED"$ then if $\text{tag}[T_{\text{bundle}}] = "MULTIPLE"$ we set to "MULTIPLE" the tag for all the nodes in the subtree with root ν . Otherwise, if $\text{tag}[T_{\text{bundle}}] = "p"$ for some local minimum p we recurse with the children of ν .

According to the above, changing the values of the $\text{tag}[\cdot]$ fields of the nodes takes in total $O(\log n)$ steps for each leaf node that was updated. The tag of each leaf node in the contour structure will be updated at most twice during the execution of the algorithm which takes $O(n \log n)$ time in total.

After executing the modified version of *ExpandUpNetwork* we check for each terrain triangle the tags of its incident edges and accordingly assign this triangle to a watershed of a local minimum or classify it as a border triangle.

We can use a variant of *ExpandMultiTricklePath* to compute the exact watershed area for each local minimum on \mathcal{T} in $O(n^2)$ as explained in the following theorem.

THEOREM 3.3. *Let \mathcal{T} be a terrain with n triangles and let P be the set of local minima on \mathcal{T} . The exact measure of the area covered by the watershed of each point $p \in P$ can be computed in $O(n^2)$ time.*

Proof. Let p, q be two points on the interior of an edge $e_1 \in \mathcal{T}$ and let π_p and π_q be the up-paths that start from these points respectively. Suppose that these two up-paths cross a common sequence of edges $\mathcal{S} = e_1 e_2 \dots e_k$ and suppose \mathcal{S} does not contain multiple elements. Let

p', q' be respectively the intersection points of π_p and π_q with e_k . Let L be the part of \mathcal{T} that is bounded by \overline{pq} , $\overline{p'q'}$, π_p , and π_q . The area of L can be expressed as a quadratic function G_S on the coordinates of p and q . We call this function the *area function* of \mathcal{S} . It is important to note that the value of G_S does not depend only on the length of \overline{pq} but on the exact position of p, q .

To compute the area of the watershed of each local minimum in P we proceed as follows. We use *ExpandMultiTricklePath* to compute for each valley edge e the intersection points of e with the paths of locally steepest descent that start from vertices of \mathcal{T} . Let $q_1(e), q_2(e), \dots, q_k(e)$ be the intersections points of e with these paths. Assume $z(q_1(e)) < z(q_2(e)) < \dots < z(q_k(e))$, and let $q_0(e)$ and $q_{k+1}(e)$ to be the lowest and highest endpoints of e respectively. The segments $\overline{q_i q_{i+1}}$ for every $0 \leq i \leq k$ bound from below the strips [14] that are incident to e . As it is shown by Yu et al [14], each strip is a region entirely contained to the watershed of some local minimum. Our approach will be to compute the area of each of the strips simultaneously and then sum the computed values of the strips that are associated with the same local minimum. For $0 \leq i \leq k$, let $p_i(e)$ be a point that we pick in an arbitrary way on the interior of $\overline{q_i q_{i+1}(e)}$.

For each valley edge $e \in \mathcal{T}$ we insert all the points $p_i(e)$ that we constructed to an initially empty queue Q . We maintain for each $p_i(e)$ a quadratic function $G[p_i(e)]$ that is initially set to zero, and we apply a new version of *ExpandUpNetwork* to Q .

For this version of the algorithm we store two extra quadratic functions with each node ν of a tree $D(\mathcal{S}_i^j) \in D_{\text{contour}}$ that stores a CW/CCW subsequence. In detail, node ν stores the quadratic function $G_{\mathcal{S}[\nu]}$ and the quadratic function $G_{\mathcal{S}'[\nu]}$ with $\mathcal{S}[\nu]$ and $\mathcal{S}'[\nu]$ defined as in Section 2. The following formula shows how we can compute $G_{\mathcal{S}[\nu]}$ in constant time given the satellite data of the children of ν :

$$G_{\mathcal{S}[\nu]} = G_{\mathcal{S}[\text{lc}[\nu]]} + G_{\mathcal{S}'[\nu]}(F_{\mathcal{S}[\text{lc}[\nu]]}) + G_{\mathcal{S}[\text{rc}[\nu]]}(F_{\mathcal{S}'[\nu]} \circ F_{\mathcal{S}[\text{lc}[\nu]]})$$

Consider a call *FindExit*($D(\mathcal{S}_i^j), \tau$) for some tree ($D(\mathcal{S}_i^j) \in D_{\text{contour}}$ that stores a CW/CCW subsequence, and some up-path τ). Let \mathcal{S} be the sequence of edges that τ traversed during this call. In this new version of *FindExit* we compute also the area function G_S as a sum of quadratic functions stored with at most $O(\log n)$ nodes in $D(\mathcal{S}_i^j)$. Let T_{bundle} be the bundle that contains τ . At the the end of the call of *FindExit* we add G_S to $G[p_i(e)]$ for every $p_i(e)$ which is the starting point of an up-path in T_{bundle} . This takes $O(n)$ time for each generated bundle instead of $O(\log n)$ which was the case for the basic version of *FindExit*. Thus the overall

running time of *ExpandUpNetwork* becomes $O(n^2)$.

After the execution of *ExpandUpNetwork* we associate with each local minimum $p \in P$ a watershed area value $A[p]$ initially set to zero. We apply each function $G[p_i(e)]$ to the points $q_i(e), q_{i+1}(e)$ and then add the computed value to $A[p]$, where p is the local minimum at which *trickle*($q_i(e)$) and *trickle*($q_{i+1}(e)$) end. The resulting value $A[p]$ is the exact watershed area of each local minimum $p \in \mathcal{T}$.

4 Concluding Remarks

We presented algorithms that compute efficiently certain flow-related structures on terrains and their characteristics: the surface network, an approximation of the watersheds of all local minima and the exact area for the watersheds of all local minima on the terrain. Our algorithms are much more efficient in the worst case than previous approaches that involve computing explicitly paths of steepest ascent/descent on the terrain. The techniques we developed may also be useful for computing approximate representations of other flow-related structures. An interesting problem for future research is to prove if it is possible to compute in subquadratic time the exact area for the watersheds of all local minima on the terrain. A positive solution to this problem may then provide a general mechanism to evaluate efficiently also other quantities related to drainage structures.

References

- [1] M. McAllister. A Watershed Algorithm for Triangulated Terrains. In *Proc. 11th Canadian Conference on Computational Geometry*, pages 103–106, 1999.
- [2] 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.
- [3] 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.
- [4] M. de Berg, O. Cheong, H. Haverkort, J. Lim and L. Toma. I/O-Efficient Flow Modeling on Fat Terrains. In *Proc. 10th Workshop on Algorithms and Data Structures*, pages 239–250, 2007.
- [5] F. Cazals, F. Chazal, and T. Lewiner. Molecular shape analysis based upon the morse-smale complex and the connolly function. In *Proc. 19th ACM Symposium on Computational Geometry*, pages 351–360, 2003.
- [6] 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.

- [7] H. Edelsbrunner, J. Harer and A. Zomorodian. Hierarchical MorseSmale Complexes for Piecewise Linear 2-Manifolds. *Discrete & Computational Geometry*, 30(1):87–107, 2003.
- [8] A. Frank, B. Palmer and V. Robinson. Formal Methods for the Accurate Definition of Some Fundamental Terms in Physical Geography. In *Proc. 2nd International Symposium Spatial Data Handling*, pages 585–599, 1986.
- [9] S. Mackay and L. Band. Extraction and Representation of Nested Catchment Areas from Digital Elevation Models in Lake-Dominated Topography. *Water Resources Research Journal*, 34(4):897–901, 1998.
- [10] J. Milnor. *Morse Theory*. Princeton University Press, New Jersey, 1963.
- [11] O. Palacios-Velez and B. Cuevas-Renaud. Automated River-Course, Ridge and Basin Delineation from Digital Elevation Data. *Journal of Hydrology*, 86:299–314, 1986.
- [12] J. Pfaltz. Surface Networks. *Geographical Analysis Journal*, 8:77–93, 1976.
- [13] D. Theobald and M. Goodchild. Artifacts of TIN-Based Surface Flow Modeling. In *Proc. of GIS/LIS'90*, pages 955–964, 1990.
- [14] 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.
- [15] X. Zhu, R. Sarkar and J. Gao. Topological data processing for distributed sensor networks with Morse-Smale decomposition. In *Proc. 28th Annual IEEE Conference on Computer Communications (INFOCOM09)*, mini-conference, pages 2911–2915, 2009.