# Implicit flow routing on terrains with applications to surface networks and drainage structures 

## Citation for published version (APA):

Berg, de, M. T., Haverkort, H. J., \& Tsirogiannis, K. (2011). Implicit flow routing on terrains with applications to surface networks and drainage structures. In D. Randall (Ed.), Proceedings 22nd Annual ACM-SIAM
Symposium on Discrete Algorithms (SODA'11, San Francisco CA, USA, January 23-25, 2011) (pp. 285-296).
Society for Industrial and Applied Mathematics (SIAM).

## Document status and date:

Published: 01/01/2011

## Document Version:

Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

## Please check the document version of this publication:

- A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
- The final author version and the galley proof are versions of the publication after peer review.
- The final published version features the final layout of the paper including the volume, issue and page numbers.
Link to publication


## General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25 fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:
www.tue.nl/taverne

## Take down policy

If you believe that this document breaches copyright please contact us at:
openaccess@tue.nl
providing details and we will investigate your claim.

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

Mark de Berg* Herman Haverkort ${ }^{\dagger} \quad$ Constantinos Tsirogiannis ${ }^{\ddagger}$


#### 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 $\Theta\left(n^{2}\right)$ complexity. The watershed of a point $p$-the set of points on $\mathcal{T}$ whose paths of steepest descent reach $p-$ can have complexity $\Theta\left(n^{3}\right)$. 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 $O\left(n^{2}\right)$ 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 higherdimensional 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

[^0]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 $\Theta\left(n^{2}\right)$. McAllister [1] and McAllister and Snoeyink [2] showed that the total complexity of the watershed boundaries of all local minima can be $\Theta\left(n^{3}\right)$. 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 $\Theta\left(n^{3}\right)$ 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 $\Theta\left(n^{2}\right)$.

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

[^1]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 $\Theta\left(n^{3}\right)$ ?

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 Čomić 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 $\Theta\left(n^{2}\right)$ 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 $O\left(n^{2}\right)$ 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 $\operatorname{trickle}(p)$ ends. As we only want to find where $\operatorname{trickle}(p)$ ends, we do not want to explicitly compute all intersection points between $\operatorname{trickle}(p)$ 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 $E V$-sequence to be the (ordered) sequence of terrain edges and vertices crossed by some path on $\mathcal{T}$. For a point $q \in \operatorname{trickle}(p)$, let $\mathcal{S}(q)$ denote the EVsequence crossed by the part of $\operatorname{trickle}(p)$ from $p$ to q. Consider a point $q \in \operatorname{trickle}(p)$ 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 $\mathcal{S}(q)=f_{1}, \ldots, f_{k}$ and let $f_{i}, \ldots, f_{j}$ 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 $\operatorname{trickle}(p)$ with $e$ that correspond to $f_{i}$ and $f_{j}$, respectively. Let $\pi\left(p, r_{i}\right)$ be the part of $\operatorname{trickle}(p)$ from $p$ to $r_{i}$ and let $\pi\left(r_{i}, r_{j}\right)$ be the part of $\operatorname{trickle}(p)$ between $r_{i}$ and $r_{j}$. Note that $\operatorname{trickle}(q) \subset \operatorname{trickle}\left(r_{j}\right)$. Define $P:=\pi\left(r_{i}, r_{j}\right) \cup \overline{r_{i} r_{j}}$. 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 $\pi\left(p, r_{i}\right)$ and trickle $\left(r_{j}\right)$ lies completely inside $P$ while the other lies completely outside $P$. This implies that a feature intersecting $\pi\left(p, r_{i}\right)$ can only intersect trickle $(q)$ if that feature intersects $\pi\left(r_{i}, r_{j}\right)$ 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 $\operatorname{trickle}(p)$ 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 $T_{\text {cycle }}$-we call this structure the cycle tree-that allows us to jump to the next new edge by performing a query $\operatorname{FindExit}\left(T_{\text {cycle }}, q\right)$. More precisely, if $\mathcal{C}=f_{i}, \ldots, f_{k}$ denotes the cycle stored in $T_{\text {cycle }}$ and $q$ is a point on $f_{i}$, then $\operatorname{FindExit}\left(T_{\text {cycle }}, q\right)$ reports a pair $\left(f_{\text {exit }}, q_{\text {exit }}\right)$ such that $f_{\text {exit }}$ is the first feature crossed by trickle $(q)$ that is not one of the features in $\mathcal{C}$ and $q_{\text {exit }}$ is the point where $\operatorname{trickle}(q)$ hits $f_{\text {exit }}$. 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 $\mathcal{S}(q)$; these edges may have to be inserted into $T_{\text {cycle }}$ later on. This leads to the following algorithm.
Algorithm ExpandTricklePath $(\mathcal{T}, p)$
Input: A triangulated terrain $\mathcal{T}$ and a point $p$ on the surface of $\mathcal{T}$.
Output: The point where $\operatorname{trickle}(p)$ ends and the edges crossed by this path.

1. Initialize an empty cycle tree $T_{\text {cycle }}$ 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: $T_{\text {cycle }}$ stores the last cycle of $\mathcal{S}(q)$, $\triangleright$ and $L$ stores its last chain.
4. Let $f$ be the first feature that $\operatorname{trickle}(q)$ crosses after leaving from $q$, and let $q^{\prime}$ be the point where

$$
\begin{array}{ll} 
& \text { trickle }(q) \text { hits } f . \\
5 . & q:=q^{\prime} \\
6 . & \text { if } f \text { is not marked } \\
7 . & \text { then Mark } f \text { and append } f \text { to } L . \\
8 . & \text { else Update } T_{\text {cycle }} \text { and empty } L . \\
9 . & \text { Set }\left(f_{\text {exit }}, q_{\text {exit }}\right):=\text { FindExit }\left(T_{\text {cycle }}, q\right), \\
& \text { mark } f_{\text {exit }}, \text { and set } q:=q_{\text {exit }} \\
10 . & \text { Append } f_{\text {exit }} \text { to } L(\text { which is currently } \\
& \text { empty) and update } T_{\text {cycle }} .
\end{array}
$$

## 11. return $q$.

It is easy to see that the invariant holds after step 1 and that it is maintained correctly, assuming $T_{\text {cycle }}$ 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 $\operatorname{first}(\mathcal{S})$ denote the first feature of $\mathcal{S}$ and let last $(\mathcal{S})$ denote its last feature. We define the trickle function $F_{\mathcal{S}}: \operatorname{first}(\mathcal{S}) \rightarrow \operatorname{last}(\mathcal{S})$ of the sequence $\mathcal{S}$ as follows. If the trickle path of a point $q \in \operatorname{first}(\mathcal{S})$ follows the sequence $\mathcal{S}$ all the way up to $\operatorname{last}(\mathcal{S})$, then $F_{\mathcal{S}}(q)$ is the point on $\operatorname{last}(\mathcal{S})$ where $\operatorname{trickle}(q)$ hits last $(\mathcal{S})$. If, on the other hand, $\operatorname{trickle}(q)$ exits $\mathcal{S}$ before reaching last $(\mathcal{S})$, then $F_{\mathcal{S}}(q)$ is undefined. We denote the domain of $F_{\mathcal{S}}$ (the part of $\operatorname{first}(\mathcal{S})$ where $F_{\mathcal{S}}$ is defined) by $\operatorname{Dom}\left(F_{\mathcal{S}}\right)$, and we denote the image of $F_{\mathcal{S}}$ by $\operatorname{Im}\left(F_{\mathcal{S}}\right)$. Since we assumed there is a trickle path crossing $\mathcal{S}$, both $\operatorname{Dom}\left(F_{\mathcal{S}}\right)$ and $\operatorname{Im}\left(F_{\mathcal{S}}\right)$ are nonempty. Fig. 1(ii) illustrates these definitions. Note that $\operatorname{Im}\left(F_{\mathcal{S}}\right)$ 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 $\operatorname{Dom}\left(F_{\mathcal{S}}\right)$ and $\operatorname{Im}\left(F_{\mathcal{S}}\right)$ are intervals of first $(\mathcal{S})$ and $\operatorname{last}(\mathcal{S})$, respectively. (ii) Suppose an EV-sequence $\mathcal{S}$ is the concatenation of $E V$-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 $\mathcal{S}(q)=f_{1} \cdots f_{k}$ and let $\mathcal{C}=f_{i}, \ldots, f_{j}$ be the last cycle of $\mathcal{S}(q)$. The cycle tree $T_{\text {cycle }}$ for $\mathcal{C}$ is a balanced binary tree, defined as follows.

- The leaves of $T_{\text {cycle }}$ store the features $f_{i}, \ldots, f_{j-1}$ in order.
- For an internal node $\nu$, let $l c[\nu]$ and $r c[\nu]$ 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 $\nu$. Furthermore, let first $[\nu]$ and last $[\nu]$ denote the features stored in the leftmost and rightmost leaf below $\nu$, respectively.
(i)



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

Then $\nu$ stores the trickle function $F_{\mathcal{S}[\nu]}$, and the trickle function $F_{\mathcal{S}^{\prime}[\nu]}$, where $\mathcal{S}^{\prime}[\nu]$ is the sequence $f_{\nu} f_{\nu}^{\prime}$ with $f_{\nu}=\operatorname{last}[l c[\nu]]$ and $f_{\nu}^{\prime}=$ first $[r c[\nu]]$.

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

Proof. Imagine following 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 $T_{\text {cycle }}$ as follows.

Suppose we arrive at a node $\nu$; initially $\nu$ is the root of $T_{\text {cycle }}$. We will maintain the invariant that $f^{*}$ is stored in a leaf below $\nu$. We will make sure that we have the point $q_{\nu}$ where $\operatorname{trickle}(q)$ crosses first $[\nu]$ available; initially $q_{\nu}=q$. When $\nu$ 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 $\nu$ if and only if
(i) $q_{\nu} \in \operatorname{Dom}\left(F_{\mathcal{S}[l c[\nu]]}\right)$, which means $\operatorname{trickle}\left(q_{\nu}\right)$ completely crosses $\mathcal{S}[l c[\nu]]$, and
(ii) $F_{\mathcal{S}[l c[\nu]]}\left(q_{\nu}\right) \in \operatorname{Dom}\left(F_{\mathcal{S}^{\prime}[\nu]}\right)$, meaning $\operatorname{trickle}\left(q_{\nu}\right)$ reaches first $[r c[\nu]]$ after crossing $\mathcal{S}[l c[\nu]]$.
If these two conditions are met, we set $\nu:=r c[\nu]$ and $q_{\nu}:=F_{\mathcal{S}^{\prime}[\nu]} \circ F_{\mathcal{S}[\nu]}\left(q_{\nu}\right)$, otherwise we set $\nu:=l c[\nu]$.

Once we have found $f^{*}$ and the point $q^{*}$ where trickle $(q)$ crosses $f^{*}$, we can compute the exit edge $e_{\text {exit }}$ and point $q_{\text {exit }}$ 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 $T_{\text {cycle }}$. 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 $T_{\text {cycle }}$ ) 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 $m \geqslant i$. 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 $T_{\text {cycle }}$ amounts to first emptying $T_{\text {cycle }}$, 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 $i \leqslant m \leqslant j$ 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 $T_{\text {cycle }}$ 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 $T_{\text {cycle }}$ can be done in logarithmic time in a standard manner.

Next consider the updating of $T_{\text {cycle }}$ 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 $T_{\text {cycle }}$ we have to split $T_{\text {cycle }}$ between $f_{m-1}$ and $f_{m}$ into two cycle trees $T_{\text {cycle }}^{1}$ and $T_{\text {cycle }}^{2}$, then merge these cycles trees again but this time in the opposite order (that is, putting $T_{\text {cycle }}^{1}$ to the right of $T_{\text {cycle }}^{2}$ 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 $O\left(n \log C_{\max }\right)$, 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 $\mathcal{M}_{\epsilon}$ 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 $\mathcal{M}_{\epsilon}$. 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 $\operatorname{upnet}(p)$. We assume that the point $p$ for which we want to compute the up-network is a terrain vertex ${ }^{2}$. An up-network is not necessarily a path; it can split and rejoin at terrain vertices. If we remove all terrain vertices from $\operatorname{upnet}(p)$, as well as all points that lie on a ridge edge, then $\operatorname{upnet}(p)$ 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 $\operatorname{upnet}(p)$ 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 $\operatorname{top}(\pi)$ denote the point up to which we have traced an up-path $\pi \in P_{z}$; the point $\operatorname{top}(\pi)$ lies on or above $h_{z}$, and it will always lie on an edge. We associate $\pi$ with the edge on which $\operatorname{top}(\pi)$ 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 $P_{z}(e)=\pi_{1}, \ldots, \pi_{k}$; here and in the sequel we number the up-paths in $P_{z}(e)$

[^2]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 $\pi_{i}$ 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 $\pi_{i}$. There can be several uppaths 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 $D_{\text {contour. }}$. Later we will explain how to implement $D_{\text {contour }}$, 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 $\operatorname{rank}(v)$, the rank of a vertex $v$, be the $z$-coordinate of $v$, and let 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 $\operatorname{rank}\left(f_{1}\right)<\operatorname{rank}\left(f_{2}\right)$, or $f_{1}$ is a vertex and $f_{2}$ is an edge and $\operatorname{rank}\left(f_{1}\right)=\operatorname{rank}\left(f_{2}\right)$. 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 $\operatorname{upnet}(p)$ ends and the edges crossed by $\operatorname{upnet}(p)$.

1. Set $z:=z(p)$, initialize $D_{\text {contour }}$ with all edges intersecting $h_{z}$, and create an event queue $Q$ storing only $p$.
while $Q$ is not empty
do Remove from $Q$ the feature $f$ that is minimal in the $\prec$-order. Set $z:=\operatorname{rank}(f)$ and update $D_{\text {contour }}$. if $f$ is a vertex, $v$
then if $v$ is a local maximum then output $v$. else $\triangleright$ Expand $v$ :

For each up-path $\pi$ starting at $v$, let $e_{\pi}$ be the first edge hit by $\pi$. If $e_{\pi}$ is incident to $v$ then insert the other vertex $w$ of $e_{\pi}$ into $Q$. If $e_{\pi}$ is not incident to $v$, then add $\pi$ to $P\left(e_{\pi}\right)$, insert $e_{\pi}$ into $Q$, and mark and report $e_{\pi}$.
10. if $f$ is an edge, $e$
11.
12.
13.
14.
then if $e$ is an edge on the boundary of $\mathcal{T}$ then Output the tops of the paths stored in $P_{z}(e)$. else $\triangleright$ Jump from $e$ :

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 $P_{z}\left(f_{\text {exit }}(b)\right)$.
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 uppath 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 $\operatorname{upnet}(p)$ 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 $e_{\text {in }}$ and $e_{\text {out }}$ be an in-edge and an outedge, respectively, of a terrain triangle $t$. Let $p, q \in e_{\text {out }}$ with $z(p)>z(q)$, and let $p^{\prime}, q^{\prime} \in e_{\text {in }}$ be such that $\overline{p^{\prime} p}$ and $\overline{q^{\prime} q}$ are parallel to the direction of steepest descent. Then $z\left(q^{\prime}\right)>z\left(p^{\prime}\right)$ if and only if the highest vertex of $e_{\text {out }}$ is the lowest vertex of $e_{\mathrm{in}}$.

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

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

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

Lemma 3.2. Let $\pi$ and $\pi^{\prime}$ be two up-paths that cross the same transfluent edge $e$, and let $q$ and $q^{\prime}$ be the points where they cross e. If $\operatorname{rank}\left(\operatorname{tail}_{\pi}(q)\right)>\operatorname{rank}\left(\operatorname{tail}_{\pi^{\prime}}\left(q^{\prime}\right)\right)$ then $z(q)>z\left(q^{\prime}\right)$.

Proof. Assume for a contradiction that $z\left(q^{\prime}\right)>z(q)$. Imagine tracing $\operatorname{tail}_{\pi}(q)$ and $\operatorname{tail}_{\pi^{\prime}}\left(q^{\prime}\right)$ downwards as long as they follow the same EV-sequence. Let $\mathcal{S}=$ $e_{1}, \ldots, e_{k}$ be this EV-sequence. Note that $e_{1}=e$. Let $q_{i}$ and $q_{i}^{\prime}$ denote the points where $\operatorname{tail}_{\pi}(q)$ and $\operatorname{tail}_{\pi^{\prime}}\left(q^{\prime}\right)$ 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 $\operatorname{rank}\left(\operatorname{tail}_{\pi}(q)\right)>\operatorname{rank}\left(\operatorname{tail}_{\pi^{\prime}}\left(q^{\prime}\right)\right)$. The assumption $z\left(q^{\prime}\right)>z(q)$ thus implies by Lemma3.1 that $z\left(q_{i}^{\prime}\right)>$ $z\left(q_{i}\right)$ for all $1 \leqslant i \leqslant k$.

Let $t$ be the triangle entered by $\operatorname{tail}_{\pi}(q)$ and $\operatorname{tail}_{\pi^{\prime}}\left(q^{\prime}\right)$ after crossing $e_{k}$, and let $v$ be the vertex of $t$ not incident to $e_{k}$. We assume for simplicity that neither $\operatorname{tail}_{\pi}(q)$ nor $\operatorname{tail}_{\pi}\left(q^{\prime}\right)$ 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 $z\left(q_{k}^{\prime}\right)>z\left(q_{k}\right)$, we know that $\operatorname{tail}_{\pi}(q)$ 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 $e_{k+1}^{\prime}$ (the third edge of $t$ ) lies strictly above the interior points of $e_{k+1}$. But then any edge crossed by $\operatorname{tail}_{\pi}\left(q_{k+1}\right)$ has a lower rank than either $e_{k}$ or $e_{k+1}^{\prime}$, and the latter two edges are crossed by $\operatorname{tail}_{\pi^{\prime}}\left(q^{\prime}\right)$. Hence, $\operatorname{rank}\left(\operatorname{tail}_{\pi^{\prime}}\left(q^{\prime}\right)\right) \geqslant \operatorname{rank}^{\left(\operatorname{tail}_{\pi}(q)\right) \text {, }}$ 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 $P_{z}\left(f_{\text {exit }}\right)$ 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 $e^{\prime}$, we denote by $B_{\mathcal{S}}\left(e, e^{\prime}\right)$ the set of bundles that follow the same edge-sequence $\mathcal{S}$ from $e$ to $e^{\prime}$ when $P_{z}(e)$ is processed. We call $B_{\mathcal{S}}\left(e, e^{\prime}\right)$ a multi-bundle. The tops of the up-paths when they reach $e^{\prime}$ after traversing $\mathcal{S}$ are called the tops of the multi-bundle.

(a)

(b)

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 $B_{1}:=B_{\mathcal{S}_{1}}\left(e_{1}, e^{\prime}\right)$ and $B_{2}:=B_{\mathcal{S}_{2}}\left(e_{2}, e^{\prime}\right)$ be two multi-bundles crossing the same transfluent edge $e^{\prime}$. Then $B_{1}$ and $B_{2}$ do not interleave on $e^{\prime}$, that is, there is a point on $e^{\prime}$ separating the tops of $B_{1}$ from the tops of $B_{2}$.
Proof. To prove part (i), let $\pi_{1}$ and $\pi_{2}$ be the two outermost up-paths in $b$. Since up-paths don't cross, any up-path starting in between $\pi_{1}$ and $\pi_{2}$ follows the same edge-sequence as $\pi_{1}$ and $\pi_{2}$ 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 $\operatorname{rank}\left(e_{1}\right) \leqslant$ $\operatorname{rank}\left(e_{2}\right)$. We will show that no up-path $\pi \in B_{1}$ can separate $B_{2}$, that is, $\operatorname{top}(\pi)$ cannot lie in between the tops of the outermost paths $\pi_{1}$ and $\pi_{2}$ 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 $\operatorname{rank}\left(e_{1}\right)<\operatorname{rank}\left(e_{2}\right)$, then according to Lemma 3.2 the tops of $B_{2}$ lie above $\operatorname{top}(\pi)$, so $\pi$ does not separate $B_{2}$.

Now consider the case ${ }^{3} \operatorname{rank}\left(e_{1}\right)=\operatorname{rank}\left(e_{2}\right)$. Let $z$ be the $z$-coordinate corresponding to this rank (so $h_{z}$ is the plane through the lower endpoints of $e_{1}$ and $\left.e_{2}\right)$. Since $e^{\prime}$ is a transfluent edge, the paths in $B_{2}$ and $\pi$ cross the same triangle $t^{\prime}$ before encountering $e^{\prime}$. We can assume that $B_{2}$ and $\pi$ enter $t^{\prime}$ through the same edge, as in Fig. 2(b), otherwise $\pi$ surely cannot separate $B_{2}$. Now imagine following $\pi$ backwards from $e^{\prime}$ as long as it follows the same edge-sequence as $B_{2}$. If $\pi$ lies in between $\pi_{1}$ and $\pi_{2}$, the path $\pi$ 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 firstotherwise, when $\pi$ jumped to $e_{1}$ it would actually have stopped at $e_{2}$. We claim that $\pi$ crosses $e_{2}$ above any of

[^3]the paths $\pi_{i} \in B_{2}$, which then implies part (ii) of the lemma. Let $t$ be the triangle that $\pi$ and the paths in $B_{2}$ cross just before $e_{2}$ and let $e^{\prime \prime}, e^{\prime \prime \prime}$ be the other two edges incident to $t$. Suppose $\pi$ enters $t$ through $e^{\prime \prime}$, as in Fig. 2(b). There are two cases.

- First consider a path $\pi^{\prime} \in B_{2}$ that also crosses $e^{\prime \prime}$ when it jumped to $e_{2}$. Let $q$ be the point where $\pi$ crosses $e^{\prime \prime}$, and let $q^{\prime}$ be the intersection point between $\pi^{\prime}$ and $e^{\prime \prime}$. Since $\operatorname{tail}_{\pi}(q)$ crosses $e_{1}$ and $\operatorname{tail}_{\pi^{\prime}}\left(q^{\prime}\right)$ does not cross any edge with rank higher or equal to $\operatorname{rank}\left(e_{2}\right)$ we have that $\operatorname{rank}_{\left(\operatorname{tail}_{\pi}(q)\right)>}$ $\operatorname{rank}\left(\operatorname{tail}_{\pi^{\prime}}\left(q^{\prime}\right)\right)$. By Lemma 3.2 we get that $q$ lies above $q^{\prime}$ on $e^{\prime \prime}$. Thus the top of $\pi$ on the forthcoming encounter with $e_{2}$ also lies above the top of $\pi^{\prime}$ on $e_{2}$ according also to Lemma 3.1, as claimed.
- Now consider a path $\pi^{\prime}$ that did not reach $e_{2}$ through $e^{\prime \prime}$, but through edge $e^{\prime \prime \prime}$. The lower vertex of $e_{2}$ is intersected by $h_{z}$, and $e^{\prime \prime}$ or a vertex of $e^{\prime \prime}$ is intersected by $h_{z}$ since there is a path from $e_{1}$ that crosses $e^{\prime \prime}$, namely $\pi$, before hitting an exit feature. Then $e^{\prime \prime \prime}$ must either lie completely below or above $h_{z}$, otherwise $t$ is horizontal. Since $\pi^{\prime}$ crosses $e^{\prime \prime \prime}$ before ever hitting $e_{2}$ then $e^{\prime \prime \prime}$ can only lie below $h_{z}$. The fact that $e^{\prime \prime \prime}$ lies below $h_{z}$ and $\pi$ crosses $e^{\prime \prime}$ above $h_{z}$ implies that top of $\pi$ on $e_{2}$ lies above the top of $\pi^{\prime}$ 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 $C_{1}, C_{2}, \ldots$ 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 CCWsubsequences depending on their common label. A ridge edge will be part of two subsequences (one cwsubsequence, 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 $D\left(\mathcal{S}_{i}^{j}\right)$, 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 $\nu \in D\left(\mathcal{S}_{i}^{j}\right)$ stores a boolean unmarked $[\nu]$ indicating whether any of the edges stored in the subtree rooted at $\nu$ 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 $D\left(\mathcal{S}_{i}^{j}\right)$ 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 $P_{z}(e)=\pi_{1}, \ldots, \pi_{k}$. Let $\operatorname{tops}_{z}(e)=$ $\tau_{1}, \ldots, \tau_{k}$ 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 $\tau_{2}, \ldots, \tau_{k-1}$ in order. Let $\operatorname{dist}\left(\tau_{i}, \tau_{j}\right)$ denote the distance between the tops $\tau_{i}$ and $\tau_{j}$. A leaf node $\nu$ that stores the top $\tau_{r}$ also stores the ratio $\frac{\operatorname{dist}\left(\tau_{r}, \tau_{r+1}\right)}{\operatorname{dist}\left(\tau_{r-1}, \tau_{r}\right)}$. This ratio remains the same when we expand the bundle upwards as long as the two paths incident to $\pi_{r}$ follow the same sequence of edges.
- For an internal node $\nu$, let first $[\nu]$ and last $[\nu]$ denote the tops stored in the leftmost and rightmost leaf below $\nu$, respectively. Let $\operatorname{pred}[\nu]$ be the top that comes before first $[\nu]$, and $\operatorname{suc}[\nu]$ the top that follows last $[\nu]$. Then $\nu$ stores the ratios $r_{1}[\nu]=$ $\frac{\operatorname{dist}(\text { first }[\nu], \text { last }[\nu])}{\operatorname{dist}(\operatorname{pred}[\nu], \text { first }[\nu])}$ and $r_{2}(\nu)=\frac{\operatorname{dist}(\operatorname{last}[\nu], \operatorname{suc}[\nu])}{\operatorname{dist}(\operatorname{pred}[\nu], \text { first }[\nu])}$.
- We store with $T_{\text {bundle }}(e)$ the coordinates of $\tau_{1}$ and $\tau_{k}$, and $\operatorname{dist}\left(\tau_{1}, \tau_{2}\right)$ and $\operatorname{dist}\left(\tau_{k-1}, \tau_{k}\right)$.
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 $\nu:=\operatorname{root}\left(T_{\text {bundle }}(e)\right)$. We maintain the invariant that
$\tau_{j}$ is stored in a leaf under $\nu$, or $j=1$, or $j=k$. Define $d:=\operatorname{dist}(\operatorname{pred}[\nu], p)$ and $\delta:=\operatorname{dist}(\operatorname{pred}[\nu], \operatorname{first}[\nu])$. Initially we have $d=\operatorname{dist}\left(\tau_{1}, p\right)$ and $\delta=\operatorname{dist}\left(\tau_{1}, \tau_{2}\right)$. Also define $\delta_{1}:=\operatorname{dist}(\operatorname{pred}[\nu], \operatorname{last}[l c[\nu]])$ and $\delta_{2}:=$ $\operatorname{dist}(\operatorname{pred}[\nu]$, first $[r c[\nu]])$. Note that $\delta_{1}=\delta \cdot\left(1+r_{1}(l c[\nu])\right)$ and $\delta_{2}=\delta_{1}+\delta \cdot r_{2}(l c[\nu])$. 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:
(i) if $d<\delta_{1}$ then $\tau_{j}$ is stored in a leaf below $l c[\nu]$ or it is $\tau_{1}$, and so we set $\nu:=l c[\nu]$.
(ii) if $\delta_{1}<d<\delta_{2}$ then $\tau_{j}$ is $\operatorname{last}(l c[\nu])$, and we are done.
(iii) if $\delta_{2}<d$ then $\tau_{j}$ is stored in a leaf below $r c[\nu]$ or it is $\tau_{k}$, and so we set $\nu:=r c[\nu]$.

The process to find $\tau_{j}$ takes logarithmic time. After finding $\tau_{j}$, we can split $T_{\text {bundle }}(e)$ in logarithmic time into a bundle tree $T_{\text {bundle }}^{1}$ for $\pi_{1}, \ldots, \pi_{j}$ and a bundle tree $T_{\text {bundle }}^{2}$ for $\pi_{j+1}, \ldots, \pi_{k}$.
Details of the algorithm. Now that we have described $D_{\text {contour }}$ and $T_{\text {bundle }}$, 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 $D_{\text {contour }}$ : 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 uppath may require updating $Q$ and then updating some set $P_{z}\left(e_{\pi}\right)$, 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 $P_{z}(e)=\pi_{1}, \ldots, \pi_{k}$, let $\tau_{1}, \ldots, \tau_{k} \in e$ 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 FindExit $\left(D\left(\mathcal{S}_{i}^{j}\right), q\right)$ reports, given a point $q$ on an edge $e$ intersecting the sweep plane $h_{z}$, the first feature $f_{\text {exit }}$ crossed by $q$ 's up-path that lies completely above $h_{z}$.

We first perform a query FindExit $\left(D\left(\mathcal{S}_{i}^{j}\right), \tau_{1}\right)$, giving us the exit feature $f_{\text {exit }}\left(\pi_{1}\right)$. Let $F_{1}: e \rightarrow f_{\text {exit }}\left(\pi_{1}\right)$ be the function that maps a point $q \in \operatorname{Dom}\left(F_{1}\right)$ to the point on $f_{\text {exit }}$ that we reach when we follow an up-path from $q$. We modify $\operatorname{FindExit}\left(D\left(\mathcal{S}_{i}^{j}\right), q\right)$ such that it also
returns $F_{1}$ and $\operatorname{Dom}\left(F_{1}\right)$. Let $T_{\text {bundle }}(e)$ be the tree storing $P_{z}(e)$. Using $T_{\text {bundle }}(e)$ we determine the largest $j$ such that $\tau_{j} \in \operatorname{Dom}\left(F_{1}\right)$ and we split $T_{\text {bundle }}(e)$ into two bundle trees $T_{\text {bundle }}^{1}$ and $T_{\text {bundle }}^{2}$, as describe above. By Lemma 3.3(i) the paths $\pi_{1}, \ldots, \pi_{j}$ follow the same edge sequence from $e$ to $f_{\text {exit }}\left(\pi_{1}\right)$, thus forming the first bundle of $P_{z}(e)$.

We repeat the process with the remainder of $P_{z}(e)$, now stored in $T_{\text {bundle }}^{2}$, 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 egdesthis will take $O(\log n)$ per marked edge - and if $f_{\text {exit }}(b)$ is an edge we insert $b$ into $P_{z}\left(f_{\text {exit }}(b)\right)$. The latter operation takes $O(\log n)$, since by Lemma 3.3(ii) $b$ does not interleave with the up-paths already stored in $P_{z}\left(f_{\text {exit }}(b)\right)$, which means we can add $T_{\text {bundle }}^{1}$ to $T_{\text {bundle }}\left(f_{\text {exit }}\right)$ 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 $P_{z^{\prime}}\left(e^{\prime}\right)$ 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 $P_{\text {saddle }}$ 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 $P_{\text {saddle }}$. 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 $\pi$, we associate $\pi$ 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 $v_{1}, \ldots, v_{k}$. Let $q_{i} \in e$ be the point where trickle $\left(v_{i}\right)$ hits $e$, and assume $z\left(q_{1}\right)<z\left(q_{2}\right)<\ldots z\left(q_{k}\right)$. Define $q_{0}$ and $q_{k+1}$ to be the lowest and highest endpoints of $e$, respectively. The points $q_{i}$ for $0 \leqslant i \leqslant k$ 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 $0 \leqslant i \leqslant k$, 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 $\operatorname{trickle}\left(v_{i}\right)$ 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 $T_{\text {bundle }}$ that is generated during the sweep we maintain a tag $\operatorname{tag}\left[T_{\text {bundle }}\right]$ in the following manner: if a bundle tree $T_{\text {bundle }}$ stores uppaths that are all tagged with the same local minimum $p$ then we have $\operatorname{tag}\left[T_{\text {bundle }}\right]=$ " $p$ " otherwise this tag has a symbolic value "Multiple". We store also such a tag for every node of $T_{\text {bundle }}$, maintaining this information for each subtree of $T_{\text {bundle }}$. In this way, whenever a new bundle tree $T_{\text {bundle }}^{\prime}$ is generated from splitting or merging other bundle trees then the value of $\operatorname{tag}\left[T_{\text {bundle }}^{\prime}\right]$ can be computed in $O(\log n)$ time.

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

- If $e[\nu]$ has not been crossed so far by any up-path then $\operatorname{tag}[\nu]$ 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 $\operatorname{tag}[\nu]=" p "$.
- If $e[\nu]$ has been crossed by up-paths that were tagged to different local minima then $\operatorname{tag}[\nu]=$ "MultiPLE".

For an internal node $\nu \in D\left(\mathcal{S}_{i}^{j}\right)$ let $T_{\nu}$ be the subtree of $D\left(\mathcal{S}_{i}^{j}\right)$ with root $\nu$. If all the leaves in $T_{\nu}$ have the same tag value then $\operatorname{tag}[\nu]$ is also set to this value. Otherwise, we distinguish two more cases. If the only tags that appear in the leaves of $T_{\nu}$ are "Multiple" and " $p$ " for only one local minimum $p$, then $\operatorname{tag}[\nu]=$ "Multiple And $p$ ". In any other case $\operatorname{tag}[\nu]=$ "Mixed". Notice that $\operatorname{tag}[\nu]=$ "Multiple" means that each valley edge represented by a leaf node in $T_{\nu}$ has been crossed by up-paths that were tagged with different local minima. However, $\operatorname{tag}[\nu]=$ "Mixed" implies that there are two or more leaf nodes in $T_{\nu}$ that have different flags with each other; for example there may exist a leaf node $\nu^{\prime}$ with tag " $p$ " and a leaf node $\nu^{\prime \prime}$ with $\operatorname{tag}\left[\nu^{\prime \prime}\right]=$ " $q$ " because $e\left[\nu^{\prime}\right]$ 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 Find$\operatorname{Exit}\left(D\left(\mathcal{S}_{i}^{j}\right), \tau\right)$ for some up-path $\tau$ and for some tree $D\left(\mathcal{S}_{i}^{j}\right) \in D_{\text {contour }}$ that stores a CW or CCW
subsequence. Let $T_{\text {bundle }}$ be the bundle tree that is generated after this query and which stores $\tau$. Let $\nu \in$ $D\left(\mathcal{S}_{i}^{j}\right)$ be a node encountered during this query such that $\tau$ was found to traverse symbolically all the edges stored in the subtree with root $\nu$. We distinguish the following cases:

- If $\operatorname{tag}[\nu]=$ "NONE" then we simply store at $\operatorname{tag}[\nu]$ the tag value of $T_{\text {bundle }}$ and we do the same for all the nodes in $T_{\nu}$.
- If $\operatorname{tag}[\nu]=$ "Multiple" then we do not change anything.
- If $\operatorname{tag}[\nu]$ corresponds to a local minimum $p$ then we check the tag of $T_{\text {bundle }}$; If also $\operatorname{tag}\left[T_{\text {bundle }}\right]=" p "$ then we do not change anything, otherwise we set $\operatorname{tag}[\nu]=$ "Multiple" for all the nodes in $T_{\nu}$.
- If $\operatorname{tag}[\nu]=" M u l t i p l e ~ A N D ~ p " ~ t h e n ~ i f ~$ $\operatorname{tag}\left[T_{\text {bundle }}\right]=" p$ " we do not change anything, otherwise we set $\operatorname{tag}[\nu]=$ "Multiple" and we recurse with the children of $\nu$.
- If $\operatorname{tag}[\nu] \quad=$ "Mixed" then if $\operatorname{tag}\left[T_{\text {bundle }}\right]=$ "Multiple" we set to "MultiPLE" the tag for all the nodes in the subtree with root $\nu$. Otherwise, if $\operatorname{tag}\left[T_{\text {bundle }}\right]=" p "$ for some local minimum $p$ we recurse with the children of $\nu$.

According to the above, changing the values of the $\operatorname{tag}[\cdot]$ 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 minim on $\mathcal{T}$ in $O\left(n^{2}\right)$ 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 $O\left(n^{2}\right)$ time.

Proof. Let $p, q$ be two points on the interior of an edge $e_{1} \in \mathcal{T}$ and let $\pi_{p}$ and $\pi_{q}$ be the up-paths that start from these points respectively. Suppose that these two uppaths cross a common sequence of edges $\mathcal{S}=e_{1} e_{2} \ldots e_{k}$ and suppose $\mathcal{S}$ does not contain multiple elements. Let
$p^{\prime}, q^{\prime}$ be respectively the intersection points of $\pi_{p}$ and $\pi_{q}$ with $e_{k}$. Let $L$ be the part of $\mathcal{T}$ that is bounded by $\overline{p q}$, $\overline{p^{\prime} q^{\prime}}, \pi_{p}$, and $\pi_{q}$. The area of $L$ can be expressed as a quadratic function $G_{\mathcal{S}}$ 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 $G_{\mathcal{S}}$ does not depend only on the length of $\overline{p q}$ 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 $q_{1}(e), q_{2}(e), \ldots, q_{k}(e)$ be the intersections points of $e$ with these paths. Assume $z\left(q_{1}(e)\right)<z\left(q_{2}(e)\right)<\ldots<$ $z\left(q_{k}(e)\right)$, 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 $0 \leqslant i \leqslant k$ 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 $0 \leqslant i \leqslant k$, let $p_{i}(e)$ be a point that we pick in an arbitrary way on the interior of $\overline{q_{i}(e) q_{i+1}(e)}$.

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 $G\left[p_{i}(e)\right]$ 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 $\nu$ of a tree $D\left(\mathcal{S}_{i}^{j}\right) \in D_{\text {contour }}$ that stores a CW/CCW subsequence. In detail, node $\nu$ stores the quadratic function $G_{\mathcal{S}[\nu]}$ and the quadratic function $G_{\mathcal{S}^{\prime}[\nu]}$ with $\mathcal{S}[\nu]$ and $\mathcal{S}^{\prime}[\nu]$ 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 $\nu$ :

$$
G_{\mathcal{S}[\nu]}=G_{\mathcal{S}[l c[\nu]]}+G_{\mathcal{S}^{\prime}[\nu]}\left(F_{\mathcal{S}[l c[\nu]]}\right)+G_{\mathcal{S}[r c[\nu]]}\left(F_{\mathcal{S}^{\prime}[\nu]} \circ F_{\mathcal{S}[l c[\nu]]}\right)
$$

Consider a call $\operatorname{Find} \operatorname{Exit}\left(D\left(\mathcal{S}_{i}^{j}\right), \tau\right)$ for some tree $\left(D\left(\mathcal{S}_{i}^{j}\right) \in D_{\text {contour }}\right.$ that stores a CW/CCW subsequence, and some up-path $\tau$. Let $\mathcal{S}$ be the sequence of edges that $\tau$ traversed during this call. In this new version of FindExit we compute also the area function $G_{\mathcal{S}}$ as a sum of quadratic functions stored with at most $O(\log n)$ nodes in $D\left(\mathcal{S}_{i}^{j}\right)$. Let $T_{\text {bundle }}$ be the bundle that contains $\tau$. At the the end of the call of FindExit we add $G_{\mathcal{S}}$ to $G\left[p_{i}(e)\right]$ for every $p_{i}(e)$ which is the starting point of an up-path in $T_{\text {bundle }}$. 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 $O\left(n^{2}\right)$.
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 $G\left[p_{i}(e)\right]$ 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 $\operatorname{trickle}\left(q_{i}(e)\right)$ and $\operatorname{trickle}\left(q_{i+1}(e)\right)$ 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 E 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 585599, 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 TINBased 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 MorseSmale decomposition. In Proc. 28th Annual IEEE Conference on Computer Communications (INFOCOM09), mini-conference, pages 2911-2915, 2009.


[^0]:    *TU Eindhoven, mdberg@win.tue.nl.
    ${ }^{\dagger}$ 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.

[^1]:    ${ }^{1}$ This 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.

[^2]:    ${ }^{2}$ If this is not the case we can just add $p$ in $V$ and re-triangulate the terrain.

[^3]:    ${ }^{3}$ The argument for the case $\operatorname{rank}\left(e_{1}\right)=\operatorname{rank}\left(e_{2}\right)$ 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 $P_{z}\left(e_{1}\right)$ cross a sequence $\mathcal{S}^{\prime}$ of edges before hitting $e^{\prime}$, while others first traverse a cycle of all edges intersecting $h_{z}$, before crossing $\mathcal{S}^{\prime}$ and hitting $e^{\prime}$.

