Implicit flow routing on triangulated terrains

de Berg, M.T.; Haverkort, H.J.; Tsirogiannis, K.

Published in:
Abstracts 27th European Workshop on Computational Geometry (EuroCG 2011, Morschach, Switzerland, March 28-30, 2011)

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 author's 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

Citation for published version (APA):

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?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.
Implicit Flow Routing on Triangulated Terrains

Mark de Berg  Herman Haverkort  Constantinos P. Tsirogiannis*

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 $T$ with $n$ vertices can have $\Theta(n^2)$ complexity, since at worst case the path can cross $\Theta(n)$ triangles for $\Theta(n)$ times each. We present a technique for tracing a path of steepest descent on $T$ in $O(n \log n)$ time implicitly, without computing all the intersection points of the path with the terrain triangles.

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 [6, 7, 8, 9].

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 $T$. 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 $T$, which is the set of all points on $T$ 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 $\Theta(n^3)$. This is due to the fact that at worst case the boundary of a watershed can consist of $\Theta(n)$ paths of steepest gradient, each of $\Theta(n^2)$ 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 point where a path of steepest descent ends, rather than all the intersections points of the path with the terrain triangles. Is it possible thus to compute for a point $p$ on $T$ the point where the trickle path from $p$ ends without explicitly computing the path itself, thereby avoiding a worst-case running time of $\Theta(n^2)$?

Our results. Inspired by the above, we study the problem of implicitly tracing paths of steepest descent or ascent on a polyhedral terrain $T$ with $n$ vertices. We give an $O(n \log n)$ algorithm that finds out where the trickle path of a given point $p$ ends, without con-

---

\(^1\)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.
Proof. Let \( S(q) = f_1, \ldots, f_k \) and let \( f_1, \ldots, f_j \) be the last cycle of \( S(q) \). 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 \( P := \pi(r_i, r_j) \cup \pi(r_j) \). 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 \( \pi(p, r_i) \) 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. \( \square \)

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 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 \( \text{FindExit}(T_{\text{cycle}}, q) \). More precisely, if \( C = f_1, \ldots, f_k \) denotes the cycle stored in \( T_{\text{cycle}} \) and \( q \) is a point on \( f_1 \), then \( \text{FindExit}(T_{\text{cycle}}, q) \) reports a pair \((f_{\text{exit}}, q_{\text{exit}})\) such that \( f_{\text{exit}} \) is the first feature crossed by \( trickle(q) \) that is not one of the features in \( C \) and \( q_{\text{exit}} \) is the point where \( trickle(q) \) 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 \( S(q) \); these edges may have to be inserted into \( T_{\text{cycle}} \) later on. This leads to the following algorithm.

**Algorithm** \( \text{ExpandTricklePath}(T, p) \)

**Input:** A triangulated terrain \( T \) and a point \( p \) on the surface of \( T \).

**Output:** The point where \( trickle(p) \) 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 \( q \neq \text{not a local minimum and flow from } q \) does not exit the terrain
3. \( \textbf{do} \) // Invariant: \( T_{\text{cycle}} \) stores the last cycle of \( S(q) \), \( T \) 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 \( trickle(q) \) hits \( f \).
5. \( q := q' \)
6. if \( f \) is not marked
7. \( \text{then} \) Mark \( f \) and append \( f \) to \( L \).
8. \( \text{else} \) Update \( T_{\text{cycle}} \) and empty \( L \).
9. Set \((f_{\text{exit}}, q_{\text{exit}}) := \text{FindExit}(T_{\text{cycle}}, q) \), mark \( f_{\text{exit}} \), and set \( q := q_{\text{exit}} \).
10. Append \( f_{\text{exit}} \) to \( L \) (which is currently empty) and update \( T_{\text{cycle}} \).
11. \( \text{return} \ q \).
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 $S$ without cycles and assume that there is some trickle path that crosses the features in $S$ in the given order. Let first($S$) denote the first feature of $S$ and let last($S$) denote its last feature. We define the \textit{trickle function} $F_S :$ first($S$) $\rightarrow$ last($S$) of the sequence $S$ as follows. If the trickle path of a point $q \in$ first($S$) follows the sequence $S$ all the way up to last($S$), then $F_S(q)$ is the point on last($S$) where trickle($q$) hits last($S$). If, on the other hand, trickle($q$) exits $S$ before reaching last($S$), then $F_S(q)$ is undefined. We denote the domain of $F_S$ (the part of first($S$) where $F_S$ is defined) by Dom($F_S$), and we denote the image of $F_S$ by Im($F_S$). Since we assumed there is a trickle path crossing $S$, both Dom($F_S$) and Im($F_S$) are non-empty. Fig. 1(ii) illustrates these definitions. Note that Im($F_S$) is a single point when one of the features in $S$ is a vertex. The following lemma follows from elementary geometry.

\textbf{Lemma 2} \hspace{0.3cm} (i) The function $F_S(q)$ is a linear function, and Dom($F_S$) and Im($F_S$) are intervals of first($S$) and last($S$), respectively. (ii) Suppose an EV-sequence $S$ is the concatenation of EV-sequences $S_1$ and $S_2$. Then $F_S$ can be computed from $F_{S_1}$ and $F_{S_2}$ in $O(1)$ time.

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

- The leaves of $T_{cycle}$ store the features $f_i, \ldots, f_{j-1}$ in order.
- For an internal node $\nu$, let $lc[\nu]$ and $rc[\nu]$ denote its left and right child, respectively. Let $S[\nu]$ denote the subsequence of $C$ 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. Then $\nu$ stores the trickle function $F_{S[\nu]}$, and the trickle function $F_{S'[\nu]}$, where $S'[\nu]$ is the sequence $f_i f'_\nu$ with $f'_\nu = \text{last}[lc[\nu]]$ and $f'_\nu = \text{first}[rc[\nu]]$.

\textbf{Lemma 3} \hspace{0.3cm} The function FindExit($T_{cycle}, q$) can be implemented to run in $O(\log |C|)$ time, where $|C|$ is the length of the cycle stored in $T_{cycle}$.

\textbf{Proof.} Imagine following trickle($q$), starting at $f_1$, the first feature in $C$. We will cross a number of features of $C$, until we exit the cycle. (We must exit the cycle before returning to $f_1$ again, because a trickle path cannot cross the same sequence twice without encountering another feature in between [5]). Let $f^*$ be the feature of $C$ that we cross just before exiting. We can find $f^*$ in $O(\log |C|)$ time by descending down $T_{cycle}$ as follows.

Suppose we arrive at a node $\nu$; initially $\nu$ is the root of $T_{cycle}$. We will maintain the invariant that $f^*$ is stored in a leaf below $\nu$. We will make sure that we have the point $q_0$ where trickle($q$) crosses first[\nu] available; initially $q_0 = q$. 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) $q_0 \in \text{Dom}(F_{S[lc[\nu]]})$, which means trickle($q_0$) completely crosses $S[lc[\nu]]$, and

(ii) $F_{S[lc[\nu]]}(q_0) \in \text{Dom}(F_{S[rc[\nu]]})$, meaning trickle($q_0$) reaches first[rc[\nu]] after crossing $S[lc[\nu]]$.

If these two conditions are met, we set $\nu := \text{rc[\nu]}$ and $q_0 := F_{S[lc[\nu]]} \circ F_{S[rc[\nu]]}(q_0)$, otherwise we set $\nu := \text{lc[\nu]}$.

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 $t$. \hfill $\Box$

It remains to explain how to update $T_{cycle}$. First consider step 8 of ExpandTricklePath. Suppose that,
just before $q$ reaches $f$, we have $S(q) = f_1 \cdots f_k$. Let $f_1 \cdots f_j$ be the last cycle of $S(q)$ (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 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 $T_{\text{cycle}}$ 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 $i \leq m \leq j$ 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 $T_{\text{cycle}}$ by deleting the features $f_1 \cdots f_{m-1}$, and inserting the features $f_j f_{j+1} \cdots f_k$. (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 $f_1 \cdots f_j$ 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 3. Then after the jump, the last cycle becomes $f_m \cdots f_j f_{j+1} \cdots f_m$. (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 $T_1$ and $T_2$. Then merge these cycles trees again but this time in the opposite order (that is, putting $T_1$ to the right of $T_2$ 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 4** Let $T$ be a terrain with $n$ triangles and let $p$ a point on the surface of $T$. Algorithm ExpandTricklePath($T, p$) traces the trickle path of $p$ in time $O(n \log C_{\text{max}})$, where $C_{\text{max}}$ is the length of the longest cycle in the EV-sequence of trickle($p$).

### 3 Applications to Other Drainage Structures

In the full version of this paper we show how we can use the presented method so as to derive an efficient mechanism that expands a collection of $\Theta(n)$ paths simultaneously. This mechanism combines the data structures that we describe above with a space-sweep algorithm. Based on this we derive $O(n \log n)$ time algorithms for:

- computing for each local minimum $p$ of $T$ the triangles contained in the watershed of $p$

We have also designed an $O(n^2)$ time algorithm that computes the watershed area for each local minimum of $T$.

### References


