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 $\mathcal{T}$ 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 $\mathcal{T}$ 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 $\mathcal{T}$. 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 $\Theta(n^3)$. 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 $\mathcal{T}$ 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 $\mathcal{T}$ 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-

*Dept. of Mathematics and Computer Science, Eindhoven University of Technology, mdberg@win.tue.nl, cs.herman@haverkort.net, ctsirogi@win.tue.nl

[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.

This is an extended abstract of a presentation given at EuroCG 2011. It has been made public for the benefit of the community and should be considered a preprint rather than a formally reviewed paper. Thus, this work is expected to appear in a conference with formal proceedings and/or in a journal.

structing 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.

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 $q \in$ *trickle*(p), let $\mathcal{S}(q)$ denote the EV-sequence crossed by the part of *trickle*(p) from p to q. Consider a point $q \in$ *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 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 *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 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_{cycle}, q). More precisely, if $\mathcal{C} = f_i, \ldots, f_k$ denotes the cycle stored in T_{cycle} and q is a point on f_i, then *FindExit*(T_{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 $\mathcal{S}(q)$; these edges may have to be inserted into T_{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 *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** ▷ Invariant: T_{cycle} stores the last cycle of $\mathcal{S}(q)$,
   ▷ 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. **then** Mark f and append f to L.
8. **else** Update T_{cycle} and empty L.
9. Set $(f_{\text{exit}}, q_{\text{exit}}) :=$ *FindExit*(T_{cycle}, q), mark f_{exit}, and set $q := q_{\text{exit}}$.
10. Append f_{exit} to L (which is currently empty) and update T_{cycle}.
11. **return** q.

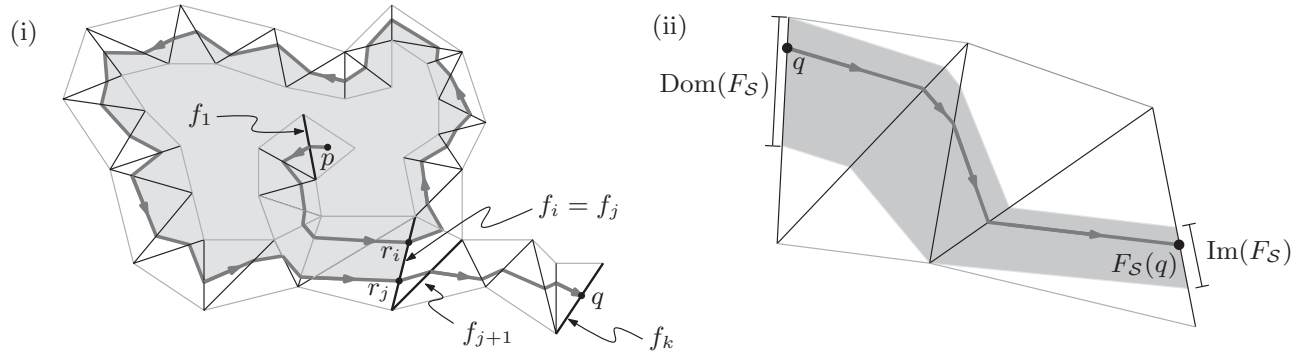

Figure 1: (i) The last cycle of the EV-sequence $\mathcal{S}(q)$ is $f_i, \ldots, f_j$, and the last chain is $f_{j+1}, \ldots, f_k$. (ii) The trickle function.

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* $F_{\mathcal{S}} : \text{first}(\mathcal{S}) \to \text{last}(\mathcal{S})$ 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 $\mathrm{Dom}(F_{\mathcal{S}})$, and we denote the image of $F_{\mathcal{S}}$ by $\mathrm{Im}(F_{\mathcal{S}})$. Since we assumed there is a trickle path crossing $\mathcal{S}$, both $\mathrm{Dom}(F_{\mathcal{S}})$ and $\mathrm{Im}(F_{\mathcal{S}})$ are non-empty. Fig. 1(ii) illustrates these definitions. Note that $\mathrm{Im}(F_{\mathcal{S}})$ is a single point when one of the features in $\mathcal{S}$ is a vertex. The following lemma follows from elementary geometry.

Lemma 2 *(i) The function $F_{\mathcal{S}}(q)$ is a linear function, and $\mathrm{Dom}(F_{\mathcal{S}})$ and $\mathrm{Im}(F_{\mathcal{S}})$ 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 $\mathcal{S}(q) = f_1 \cdots f_k$ and let $\mathcal{C} = f_i, \ldots, f_j$ 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 $f_i, \ldots, f_{j-1}$ 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. 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 3 *The function FindExit(T_{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 *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 *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) $q_\nu \in \mathrm{Dom}(F_{\mathcal{S}[lc[\nu]]})$, which means *trickle*$(q_\nu)$ completely crosses $\mathcal{S}[lc[\nu]]$, and

(ii) $F_{\mathcal{S}[lc[\nu]]}(q_\nu) \in \mathrm{Dom}(F_{\mathcal{S}'[\nu]})$, meaning *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 *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 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_{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 $i \leqslant m \leqslant j$ 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_{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 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^1_{cycle} and T^2_{cycle}, then merge these cycles trees again but this time in the opposite order (that is, putting T^1_{cycle} to the right of T^2_{cycle} 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 $\mathcal{T}$ 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 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 $\mathcal{T}$ the triangles contained in the watershed of p
- computing the surface network graph [5] of $\mathcal{T}$.

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

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

[6] A. 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.

[7] 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.

[8] 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.

[9] D. Theobald and M. Goodchild. Artifacts of TIN-Based Surface Flow Modeling. In *Proc. of GIS/LIS'90*, pages 955–964, 1990.

[10] 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.