Computing River Floods Using Massive Terrain Data

Cici Alexander
Dynamiques de l'Environnement Côtier
IFREMER, France
calexand@ifremer.fr

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

Peder Klith Bøcher
Department of Bioscience
Aarhus University, Denmark
peder.bocher@biology.au.dk

Morten Revsbæk
SCALGO
Aarhus, Denmark
morten@scalgo.com

Brody Sandel
Department of Bioscience and MADALGO*
Aarhus University, Denmark
brody.sandel@bios.au.dk

Jens-Christian Svenning
Department of Bioscience
Aarhus University, Denmark
svenning@biology.au.dk

Constantinos Tsirogiannis
Department of Bioscience and MADALGO*
Aarhus University, Denmark
constant@madalgo.au.dk

Jungwoo Yang
MADALGO*
Aarhus University, Denmark
jungwoo@madalgo.au.dk

ABSTRACT

Many times river floods have resulted in huge catastrophes. To reduce the negative outcome of such floods, it is important to predict their extent before they happen. For this reason, scientists nowadays use algorithms that model river floods on digital terrains. Yet, all existing algorithms of this kind have a major drawback; they cannot efficiently process massive terrain datasets, which have become widely available during the last years.

In this paper, we describe two algorithms that provide high-quality river flood modelling and, unlike any previous approach, efficiently handle massive terrain data. More specifically, given a raster terrain and a subset of its cells representing a river network, we describe two algorithms that for each cell in the raster estimate the height that the river should rise for the cell to get flooded. One of the proposed algorithms is a redesign of a European Union approved method that is used by authorities in Denmark for modelling river floods. We show how this algorithm can be adapted to efficiently handle massive terrain data. The other algorithm is a novel method that we introduce for modelling river floods. For an input raster that consists of N cells, and which is so large that it can only be stored in the hard disk of a computer, each of the proposed algorithms can produce its output with only $O(\text{sort}(N))$ transfers of data blocks between the disk and the main memory. Here $\text{sort}(N)$ denotes the minimum number of data transfers that are needed for sorting a set of N elements stored on disk. We have implemented both algorithms, and compared their output with data that were acquired from a real flood event. We show that both algorithms produce an output that models the actual event quite accurately. In fact, the new algorithm that we introduce produces more accurate results than the existing popular method. We evaluated the efficiency of our algorithms in practice by conducting experiments on massive datasets. We show that the two algorithms perform efficiently even for datasets of approximatelly 268 GB size.

*Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation.

1. INTRODUCTION

Throughout history, river floods have caused large disasters. Usually induced by heavy rainfall, such floods can lead to casualties and huge financial damage for the local communities. A recent example is the catastrophic flood of the Indus river in Pakistan that took place in 2010 [6]. This flood claimed approximatelly two thousand lives, and about one fifth of the total area of the country ended up covered by water. Society wants to predict such floods, so that measures can be taken in advance to reduce the harm done. Therefore, it is important for people to know which regions around a river have the highest risk of getting flooded when the level of the river rises.

Today, hydrologists use computers to model river floods; they use specialised software to simulate flood events based on digital representations of terrains and rivers. Such terrain representations are widely known as Digital Elevation Models (DEMs). The most popular type of DEMs is the so-called *grid* or *raster* DEMs. In a raster DEM the domain of the terrain is divided into square cells of equal size, and each cell is associated with an elevation value.

One method for modelling river floods on DEMs is the method introduced by Berg Sonne [11]; let $\mathcal{G}$ be a raster terrain and let $R(\mathcal{G})$ be the set of cells in $\mathcal{G}$ that represents the region covered by a river network in this terrain. Also, let x be a positive real. Given $\mathcal{G}$, $R(\mathcal{G})$ and x, the method

estimates which cells in $\mathcal{G}$ will get flooded if the level of the river $R(\mathcal{G})$ rises uniformly by x meters. Of course, a flood is a very complex phenomenon and is influenced by many factors, some of which are difficult to determine. Therefore, we cannot expect that a flood can be modelled precisely by the output of any method, no matter how involved this method is. However, the method proposed by Berg Sonne is today considered a quite accurate tool for modelling river floods. Hence, after approval by the European Union it is used by the state authorities of Denmark [11].

However, Berg Sonne's method has a major drawback; it cannot process massive DEMs. Recent advances in Lidar technology have made it possible to produce detailed and huge DEM datasets. In many cases, such a dataset is so large that it cannot fit in the main memory of a standard computer. Hence, the dataset has to be stored mainly on disk. Since the computer's processor can only handle data that appear in the main memory, blocks of data have to be transfered between the disk and the memory in order to process the dataset. We call a transfer of a single block of data between the disk and the memory an *I/O-operation*, or an I/O for short. The problem here is that a single I/O is an extremely slow operation; it can take about the same time as a million CPU operations. Therefore, when it comes to processing huge amounts of data, it is important to process the dataset in a way that we minimize the number of data transfers between the disk and the memory. Otherwise, the whole process becomes practically infeasible. To handle this issue, Aggarwal and Vitter introduced the so called *I/O-model*, which takes into account the number of I/Os between the disk and the main memory [3]. The performance of an algorithm in the I/O-model is measured as the number of I/Os that take place during its execution. This measure of performance is called the *I/O-efficiency* of the algorithm. To describe the I/O-efficiency we need three parameters; the size N of the input data, the size of the internal memory M, and the size B of a single block of data that can be transfered from and to the disk. Two basic processes that take place during the execution of most algorithms is scanning and sorting. We can scan a set of N elements stored in the disk with $O(\text{scan}(N))$ I/Os, where $\text{scan}(N) = N/B$. We can also sort a set of N records in an I/O-efficient manner with $O(\text{sort}(N))$ I/Os, where $\text{sort}(N) = N/B \log_{M/B} N/B$ [3].

Standard algorithms are often designed based on the assumption that all input data fit in the main memory. Hence, usually they cannot handle massive datasets. This is also the case for algorithms that are used to model river floods; to the best of our knowledge, there does not exist any I/O-efficient algorithm for this problem. Therefore, the users of up-to-date hydrological software are forced to choose between two approaches. In the first approach, the resolution of the input DEM is reduced (so it fits entirely in the main memory). Thus, a large amount of detail in terrain data is thrown away. Important features on the landscape, such as ditches and levees, may not be depicted anymore on the resulting terrain. When it comes to modelling a river flood, this results in wrong estimations. In the other approach, users divide the massive DEM into smaller tiles and each tile is processed independently; in this way, when processing a single tile, we do not take into account how the rest of the landscape affects the flood in that region. Therefore, there is a need for developing algorithms that, on one hand model river floods accuratelly, and on the other hand handle massive terrain datasets efficiently.

Our Results.

Inspired by the above, we designed two I/O-efficient algorithms that can be used for modelling river floods. The first algorithm is an adaptation of Berg Sonne's method that can handle massive raster terrains. The second algorithm is a novel method that we introduce for modelling river floods. For each of these algorithms, the input is a raster $\mathcal{G}$, and a subset $R(\mathcal{G})$ of cells in $\mathcal{G}$ representing the area covered by a river network. Each of our algorithms returns for each cell $c \in \mathcal{G}$ a value $\text{f}(c)$, indicating the minimum number of meters the river level should rise before c gets flooded. We call this value the *resistance* value of c. Given the resistance values $\text{f}(c)$ for every $c \in \mathcal{G}$, and a positive integer x, we can then easily extract the part of the terrain that is flooded if the river level rises uniformly by x meters. Each of the algorithms that we propose uses different criteria for computing resistance values, hence they produce different outputs. In the algorithm by Berg Sonne, the resistance value of each cell is computed in two stages; in the first stage, for each cell c the elevation difference is computed between c and its closest river cell (in the xy-plane). We call this elevation difference the *obstruction* value of c. In the second stage, the resistance value of c is computed based on the obstruction values of the cells that appear on any path connecting c with the river network. In our new algorithm, we compute the resistance values by taking into account how water flows on the terrain surface. In particular, we consider a model according to which water can flow from a cell to potentially more than one of its neighbours. Based on this model, we compute the resistance value of each cell $c \in \mathcal{G}$ as the elevation difference between c and the highest river cell where water from c can reach.

We have designed both of our algorithms using the I/O-model of Agarwal and Vitter [3]. To compute the resistance values on a raster that has N cells, each of our algorithms require $O(\text{sort}(N))$ I/Os in the worst case. We have implemented both algorithms and measured their efficiency in practice. We show that the two algorithms perform efficiently even for raster datasets of approximatelly 268 gigabytes size; to process a dataset of this size on a standard workstation, our adaptation for Berg Sonne's method required roughly 24 hours, and our new algorithm roughly 31 hours. We also conducted experiments to evaluate that our algorithms can model adequately real flood events. To do this, we worked as follows; we used as reference a vector dataset which outlines the river flood that took place in Pakistan in 2010 [6]. We also used each of our algorithms to compute the resistance values on a grid that models the terrain in Pakistan. Then we selected a large number of pairs of cells from this grid; each pair was selected such that one cell in the pair is covered by the flooded region in the vector dataset, and the other cell falls outside this region. On the output of each algorithm, we measured the percentage of pairs where the flooded cell of the pair scores a lower resistance value than the non-flooded cell. This percentage was 87% for Berg Sonne's method and 92% for our algorithm. We repeated the same measurements many times, each time sampling cell pairs within a different small region overlapping with the flooded area. We observed that the percentage scored by each method depends on the size of the sampling region, and the topographic heterogeneity within this region.

The lowest percentages were observed for the sampling regions of the smallest size that we considered; these regions were squares of 20 km dimension. For such regions, the mean percentage measured for Berg Sonne's method is 61%, and the mean percentage for our algorithm is 71%. For all region sizes that we used, our algorithm provided on average more accurate results than Berg Sonne's method. To understand the reasons behind this difference in the output quality, we used both algorithms to model river floods on a massive raster that represents the terrain in Denmark. Among other artefacts, the method by Berg Sonne produced flooded regions that had larger size than the ones calculated by our algorithm. As we explain, one reason for this is that Berg Sonne's method produces very small resistance values for areas along the entire coastline of the terrain.

2. PROBLEM DEFINITION AND NOTATION

Let $\mathcal{G}$ be a grid terrain that consists of N cells. For every cell $c \in \mathcal{G}$ we use $h(c)$ to indicate the elevation of the terrain at this cell. We denote the cell that appears at the i-th row and j-th column of $\mathcal{G}$ by $\mathcal{G}(i,j)$. We assume without loss of generality that the center of grid cell $\mathcal{G}(i,j)$ has xy-coordinates (j,i). For any cell $c \in G$, we denote this center point by $p(c)$. We call the *xy-distance*, or simply the *distance*, between two cells in $\mathcal{G}$ the 2D Euclidean distance between their cell centers on the xy-domain of $\mathcal{G}$. Let C be a set of cells in $\mathcal{G}$ and let c be a cell that belongs to this set. We say that c is the *closest cell* in C to another cell c' if c has the smallest xy-distance to c' compared to any other cell in C.

We use $R(\mathcal{G})$ to denote a subset of the cells in $\mathcal{G}$ that belong to a river network of the terrain. We call these cells the *river cells* of $\mathcal{G}$. The cells in $R(\mathcal{G})$ represent the river network in $\mathcal{G}$ when there is no flood. This implies that the elevation value of each cell in $R(\mathcal{G})$ approximates the average height of the river level at this location when no flood occurs. For the algorithms that we present, we assume that $R(\mathcal{G})$ is provided as part of the input.

Let h_{rise} be a positive real. We say that there is a *river rise* of h_{rise} meters, or that *the river rises* by h_{rise} meters, when for each cell $c \in R(\mathcal{G})$ the river level rises to elevation $h(c) + h_{\text{rise}}$. We call h_{rise} the *rise value*. Thus, when a river rise takes place we assume that the level of the river increases by the same amount at all river cells.

We study the following problem. Given a terrain $\mathcal{G}$ and its river network $R(\mathcal{G})$, we want to compute for every cell $c \in \mathcal{G}$ a value f(c) that estimates the minimum value h_{rise} such that c gets flooded when the river rises by h_{rise} meters. We call this value the *resistance* value of c. Each of the two algorithms that we present in this paper defines these resistance values in a different way; hence, for the same input grid the output between the two algorithms may differ substantially. For each algorithm. we provide a detailed definition for the resistance of a grid cell in the description of the algorithm. For both algorithms, it is assumed that all river cells are flooded by default. Therefore, for both approaches we imply that the resistance value of every river cell is set to zero.

3. DESCRIPTION OF THE ALGORITHMS

3.1 Adaptation of Berg Sonne's Method

The first algorithm that we describe is based on the flood modelling method introduced by Berg Sonne [11]. Originally, this method was designed to solve a more simple problem than the one that we examine. In particular, the input of the original method is a raster $\mathcal{G}$, the river network $R(\mathcal{G})$, and a rise value h_{rise}. Instead of computing flood resistance values, the method outputs the cells in $\mathcal{G}$ that are considered to get flooded when $R(\mathcal{G})$ rises by h_{rise} meters. We call this version of the method *ProximityFlood*. Below, we first explain how *ProximityFlood* calculates the flooded cells in $\mathcal{G}$ for a given rise value h_{rise}. Then, we show how we can use this method to design an I/O-efficient algorithm that computes a flood resistance value for each input cell[1].

ProximityFlood consists of two steps. In the first step, every cell $c \in \mathcal{G} \setminus R(\mathcal{G})$ gets associated with a single river cell in $R(\mathcal{G})$; this is the river cell from which we consider that c can potentially get flooded. We call this cell the *source* cell of c, and we denote this by source(c). The source cell for every $c \in \mathcal{G} \setminus R(\mathcal{G})$ is defined as the river cell $c' \in R(\mathcal{G})$ that has the smallest xy-distance from c. After calculating source(c) for every non-river cell c, the height difference between source(c) and c is computed and stored together with c. We call this value the *obstruction* value obst(c) of c.

In the second step, we extract the cells in $\mathcal{G}$ that are considered to get flooded when the river rises by h_{rise} meters. More specifically, we extract any cell c that a) has an obstruction value obst$[c] \leqslant h_{\text{rise}}$ *and* b) there exists a path of cells between c and a river cell c_R such that any non-river cell c' in this path has an obstruction value obst$(c') \leqslant h_{\text{rise}}$. Notice that, in this way, not all cells with obstruction $\leqslant h_{\text{rise}}$ are flooded.

Method *ProximityFlood* can be used to model a single flood event at a time. On the other hand, if we want to study which regions get flooded for different rise values then we have to run this method many times, once for each distinct rise value h_{rise}. To avoid this, we instead choose to compute for each cell c the minimum rise value h_{rise} for which c gets flooded according to method *ProximityFlood*. Below we describe our new I/O-efficient algorithm that does this, which we call *ProximityResistance*.

As with *ProximityFlood*, the new algorithm consists of two main steps. In the first step, we compute for each cell $c \in \mathcal{G} \setminus R(\mathcal{G})$ the source cell source(c) and the obstruction obst(c). In the second step, we calculate the flood resistance values of all cells in $\mathcal{G} \setminus R(\mathcal{G})$.

Computing the source cells and obstruction values.

For the first step, the main task is to compute the source cell for each non-river cell c; given this cell, it is straightforward to compute the obstruction obst(c). Calculating the source cells in $\mathcal{G}$ is similar to computing a Voronoi di-

[1]Some implementations of *ProximityFlood* include an extra preprocessing step where the heights of the river cells are adjusted to make it consistent with the rest of the terrain data. This is useful when the river dataset is acquired from a different source than the terrain raster. In that case, projecting the river data on the raster may create artefacts (such as rivers that flow upstream). In the description that we provide for *ProximityFlood* we do not include this preprocessing step; we consider that this step has to do more with configuring the datasets rather than with the method itself. Yet, the preprocessing step can be also handled in an I/O-efficient manner, given a realistic assumption on the memory size.

agram on the xy-domain of $\mathcal{G}$; the sites of the Voronoi diagram are the center-points of the river cells in $\mathcal{G}$ and for any cell $c \in \mathcal{G} \setminus R(\mathcal{G})$ it holds that source$(c) = c'$ if the center of c falls in the Voronoi region of $p(c')$. Computing the Voronoi diagram of the river cells can be done in $O(\text{sort}(N))$ I/Os [10, 2]. Then, we sweep simultaneously the diagram and grid $\mathcal{G}$. During the sweep, we maintain the diagram edges that intersect the sweep line, sorted according to the x-coordinate of their intersection point with this line. For every row of $\mathcal{G}$ that we encounter, we scan the edges that intersect the sweep line to determine the Voronoi region (and therefore the corresponding source cell) where each cell in the row belongs to. Notice that the number of edges in the sweep line cannot be more than two times the number of cells in a row. This is because there cannot be more than two river cells on a single column whose Voronoi regions intersect the same horizontal line. Therefore, scanning the raster and updating the sweep line can be done efficiently in $O(\text{sort}(N))$ I/Os in total. From this we conclude that computing the source cells on the raster can be performed in $O(\text{sort}(N))$ I/Os. But we can do this more efficiently; in the appendix we present an algorithm that computes the source cells using $O(\text{scan}(N))$ I/Os.

Computing the flood resistance values.

In the second step of method *ProximityResistance* we compute for each cell $c \in \mathcal{G}$ its flood resistance f(c). Recall that for every cell c this resistance value is equal to the minimum rise value h_{rise} such that obst$(c) \leqslant h_{\text{rise}}$ and c is connected to the river by a path of cells with obstruction $\leqslant h_{\text{rise}}$. Based on this definition, we can reduce the computation of the flood resistance values to the problem of computing the *raise elevations* on a terrain, that was described by Arge *et al.* as part of their partial flooding algorithm [8]. This problem is defined as follows; let $\mathcal{G}$ be a raster and let $\zeta_1, \ldots, \zeta_k$ be a set of cells in $\mathcal{G}$ that we call *sinks*. For any path of cells path in $\mathcal{G}$ the height of path is defined as the height of the highest cell on this path. The *raise elevation* of a cell $c \in \mathcal{G}$ is the minimum height among all paths that connect c to ζ_i for any $1 \leqslant i \leqslant k$. Arge *et al.* provide an algorithm that computes the raise elevations for all the cells on the terrain in $O(\text{sort}(N))$ I/Os [8].

We can reduce the problem of computing the flood resistance values of the cells in $\mathcal{G}$ to an instance of the raise elevation problem as follows; we create a raster $\mathcal{G}'$ that has the same number of rows and columns as $\mathcal{G}$. For any river cell $\mathcal{G}(i,j) \in R(\mathcal{G})$ we let the corresponding cell $\mathcal{G}'(i,j)$ to be a sink. For any non-river cell $\mathcal{G}(i,j)$ we let cell $\mathcal{G}'(i,j)$ have elevation equal to the obstruction value of $\mathcal{G}(i,j)$. It is know easy to see that the raise value of any cell $\mathcal{G}'(i,j)$ is equal to the flood resistance value that we want to compute for $\mathcal{G}(i,j)$. By applying the I/O-efficient algorithm of Arge *et al.* on $\mathcal{G}'$ we can compute the described flood resistance values in $O(\text{sort}(N))$ I/Os.

THEOREM 3.1. Let $\mathcal{G}$ be a raster terrain that consists of N cells, and let $R(\mathcal{G})$ be the set of all river cells in this raster. *ProximityResistance* computes the flood resistance values of all cells in $\mathcal{G}$ using $O(\text{sort}(N))$ I/Os.

3.2 Our New Method

In *ProximityResistance*, a cell c can only get flooded from the closest river cell source(c) in the xy-plane. Intuitively, this is very unnatural since the flow of water on the terrain is obviously influenced by the terrain topography. Therefore, we introduce a novel method which instead chooses source(c) based on a model that represents how water flows on the terrain. We refer to this new method as *UpstreamResistance*. Below, we first describe how source(c) is chosen in *UpstreamResistance*, and then we show how we can compute this I/O-efficiently.

For a raster $\mathcal{G}$ let $\mathcal{F}(\mathcal{G}) = (V, E)$ be the graph such that for each cell $c \in \mathcal{G}$ there exists exactly one vertex $v(c)$ in V, and there exists a directed edge in E from $v(c)$ to $v(c')$ if cells $c, c' \in \mathcal{G}$ are adjacent and $h(c) > h(c')$. We call this graph the *flow graph* of $\mathcal{G}$. For now let us assume that no adjacent cells in $\mathcal{G}$ have the same elevation value. Hence, there exists exactly one directed edge in $\mathcal{F}(\mathcal{G})$ for each pair of adjacent cells in $\mathcal{G}$, and $\mathcal{F}(\mathcal{G})$ is a DAG. The concept of the flow graph was introduced in previous works to model how water flows between cells on a DEM. It is naturally assumed that water on a cell can flow only to neighbour cells with lower height; that is modelled with a directed edge in the flow graph.

For any cell $c \in G$ water from c may flow following different routes on the raster until reaching one or more cells on the boundary of river $R(\mathcal{G})$. In method *UpstreamResistance* we choose source(c) to be one of these cells on the river boundary, that is, the river cells where the water from c reaches. More formally, let c be a cell in $\mathcal{G}$. Consider a path in $\mathcal{F}(\mathcal{G})$ that starts from vertex $v(c)$ and ends at a vertex $v(c')$ where c' is a river cell, such that the path does not contain a vertex corresponding to any other river cell. We call such a path a *downstream path* of c. Let $DC(c)$ denote the set of all river cells that belong to some downstream path of c. In method *UpstreamResistance*, source(c) is the cell in $DC(c)$ with the highest elevation value. The flood resistance of c is then defined as the height difference $h(c) - h(\text{source}(c))$.

When it comes to implementing *UpstreamResistance* I/O-efficiently, the two key tasks for computing the flood resistances are constructing the flow graph $\mathcal{F}(\mathcal{G})$, and computing the source cell for every cell in $\mathcal{G}$. If no flat areas exist on $\mathcal{G}$, we can construct $\mathcal{F}(\mathcal{G})$ straightforwardly in $O(\text{scan}(N))$ I/Os. As for computing the source cells, observe that for any cell c it holds that source$(c) =$ source(c') for some c' such that there exists an edge in $\mathcal{F}(\mathcal{G})$ from $v(c)$ to $v(c')$. Therefore, we can compute source(c) by first computing the source cells for those neighbours of c that appear downstream in $\mathcal{F}(\mathcal{G})$, and then use these to infer source(c). Arge *et al.* describe an I/O-efficient algorithm that computes the number of upstream cells for every cell on a raster in $O(\text{sort}(N))$ I/Os [4]. Their algorithm can be easily modified for computing the source cells in $\mathcal{G}$. Therefore, we can perform this computation in $O(\text{sort}(N))$ I/Os.

Handling flat areas.

Terrain datasets often contain large connected regions of cells that have exactly the same elevation. Given raster $\mathcal{G}$ that contains such flat areas, we have to perform two extra steps; first, we have to outline all distinct flat areas in $\mathcal{G}$, and then we have to model how water flows on each such area. For the second step, we will have to modify the definition of the flow graph so as to represent flow between cells in a flat area. To outline the flat areas in $\mathcal{G}$ we have to compute the connected components of cells in the raster that have the same elevation. This can be done I/O-efficiently in $O(\text{sort}(N))$ I/Os. Let $A \subseteq \mathcal{G}$ be a flat area in $\mathcal{G}$, and

let c be a cell on the boundary of A. We say that c is a *spill point* of A if c is adjacent to at least one cell that has elevation lower than $h(c)$. When modelling water flow on A the goal is to route flow so that every cell in A drains to at least one spill point of this area (if such a spill point exists). Let A be a flat area that has at least one spill point. Every cell $c \in A$ can be routed to all the spill points in A. Therefore, all cells in A should have exactly the same source cell and also the same flood resistance. To represent this appropriatelly, we modify slightly the way we construct the flow graph such that instead of representing each cell in A by exactly one vertex, we use a single vertex to represent the entire flat area. We denote this vertex by $v(A)$. The in-edges of $v(A)$ connect this vertex with all vertices $v(c)$ such that c is a cell adjacent to A, and c has a higher elevation than A. Similarly, the out-edges of $v(A)$ connect to all lower elevation vertices in the graph representing cells adjacent to A.

After building the flow graph in this manner, we proceed with the computation of the resistance values. We processing a vertex that represents a flat region A, we distinguish two cases depending on whether A contains river cells or not. In the case that A does not contain any river cell, we find the highest river cell c that appears on a downstream path from $v(A)$, and for every cell in A we set the flood resistance to the elevation difference between A and c. In the case that A contains river cells, we consider that all cells in this area are flooded by default. Hence, for every cell in A we set the flood resistance value to zero. In that case, vertex $v(A)$ is treated in the flow graph in the same way as a vertex that represents a single river cell; for each cell c such that $v(A)$ appears in a downstream path from c we use the elevation of A to determine the flood resistance of c, as if A was a single river cell.

Note that not all flat areas have a spill point. In that case, a flat area is a region of locally minimum elevation in $\mathcal{G}$. Let A be such an area in $\mathcal{G}$. If A does not contain any river cell then we consider that A corresponds to a spurious pit. We remove all such pits by raising the elevation of the terrain within and around this region. The removal of the spurious pits can be done as a preprocessing step before constructing the actual flow graph on $\mathcal{G}$. We can do this I/O-efficiently in $O(\text{sort}(N))$ I/Os, using the partial flooding algorithm described by Danner *et al.* [8]. On the other hand, if A contains river cells, then we process this area in the same way as we did with flat areas that contain both river cells and spill points; the flood resistance of all cells in A are set to zero, and the entire area is represented by a single vertex $v(A)$ in the flow graph.

From the above description, we conclude that constructing the flow graph for a raster $\mathcal{G}$ boils down to computing the connected components of flat regions in $\mathcal{G}$, and then adding in the described way the graph edges between the vertices of the flat regions and the adjacent cells. We can compute the connected components of flat regions on $\mathcal{G}$ using the batched union-find algorithm of Agarwal *et al.* [1] which uses $O(\text{sort}(N))$ I/Os. Constructing the modified flow graph, and processing the vertices that represent flat areas can be done straightforwardly in $O(\text{sort}(N))$ I/Os.

THEOREM 3.2. Let $\mathcal{G}$ be a raster terrain that consists of N cells, and let $R(\mathcal{G})$ be the set of river cells in this raster. *UpstreamResistance* computes the flood resistance values of all cells in $\mathcal{G}$ using $O(\text{sort}(N))$ I/Os.

4. IMPLEMENTATIONS AND EXPERIMENTS

We implemented both algorithms described in Section 3 in order to evaluate how fast they perform in practice, as well as how good they model real flood events.

4.1 Description of Implementations

We implemented both algorithms in C++, using the open source library TPIE that provides I/O-efficient algorithms for sorting and scanning data [13]. We used the GNU g++ compiler (version 4.8.2), and the experiments were ran on a Linux Ubuntu operating system (release 14.04).

When implementing *ProximityResistance* we made two modifications compared to the description in Section 3. First, when computing the source cells on $\mathcal{G}$ using a sweepline approach, we used the $O(\text{scan}(N))$ approach (described in the appendix) and we made the practically realistic assumption that a constant number of rows in $\mathcal{G}$ can fit in main memory. Thus, instead of performing an external scan of each row and maintaining an I/O-efficient stack during the sweep, we simply store the two last rows that we swept in memory and perform all computations internally. Second, when computing the raise elevations we did not use the $O(\text{sort}(N))$ batched union-find algorithm by Agarwal *et al.* [1] (that is quite involved), but instead a much simpler $O(\text{sort}(N)\log(N/M))$ algorithm also proposed by Agarwal *et al.*. Agarwal *et al.* [1] and Danner *et al.* [8] have showed that this simple union-find algorithm performs very well in practice.

When implementing *UpstreamResistance* we accurately followed the description in Section 3. The only difference was that we again used the practical union-find algorithm of Agarwal *et al.* (that requires $O(\text{sort}(N)\log(N/M))$ I/Os), this time for computing the connected components of flat areas in $\mathcal{G}$, and for removing flat areas that correspond to spurious pits.

4.2 Measuring I/O-Efficiency in Practice

To measure the practical efficiency of each method, we ran our implementations on a massive raster dataset that represents the terrain surface of the entire country of Denmark. Publicly available through the website of the Danish Ministry of Environment [7], this raster consists of roughly 66.4 billion cells, arranged in 287500 rows and 231250 columns. Each cell represents a region of 1.6×1.6 meters on the terrain and is assigned an elevation value which is a 4-byte floating point number. The total size of the uncompressed dataset is 268 gigabytes. We refer to this dataset as `denmark`.

Raster `denmark` does not include any river data, and therefore we had to extract the river cells before conducting the experiments. To do so, we first preprocessed the raster by removing all shallow pits. Then, we selected the river cells based on the size of their upstream area. For this reason, we computed the flow graph of `denmark` as described in Section 3 except that for each cell c we included at most one outgoing edge. This outgoing edge points to the vertex $v(c')$ such that c' is a neighbour of c and the vector from $p(c)$ to $p(c')$ has the steepest downward slope. Then, we computed for each cell c the size of its upstream area; this is the area that is covered by all cells c' such that there exists a path from $v(c')$ to $v(c)$ in the flow graph. We extracted the river cells by selecting all cells whose upstream area was larger than 12.5 km^2. We picked this threshold since the resulting river network resembles better the actual shape of the rivers in Denmark, according to available orthophotos.

We ran both of our algorithms on the `denmark` raster and the extracted cells, using a workstation that has a Xeon CPU (W3565), a four-core processor with 3.2GHz per core. The workstation had 48 Gigabytes of main memory, and a raid (redundant array of independent disks) that consists of nineteen disks, with 3 Terrabytes capacity in total. The maximum amount of main memory that was available at any point during the execution of our implementations was 22 Gigabytes. The total time taken by the implementation of *ProximityResistance* was roughly 24.2 hours; only 2.4 hours was used for computing the source cell for each non-river cell, and the rest 21.8 hours were spent on computing the resistance values. For the implementation of *UpstreamResistance*, the total execution time was approximatelly 31.1 hours. The first stage of this method, where the flow graph of the input raster is computed, took 12.5 hours. The rest 18.6 hours were spent for delineating the flat areas on the terrain, and computing the resistance values.

From the above, it is clear that the implementations of both methods have a very good performance even for an enormous dataset such as `denmark`. Each method took less than one and a half day to process this dataset, using an amount of main memory which corresponds to roughly 8% of the datasets total size.

4.3 Evaluating the Quality of Flood Modelling

In the second set of experiments we used an actual flood event to evaluate the quality of the output produced by the two methods, namely the catastrophic flood of the Indus river that took place in Pakistan in 2010 [6].

For the experiments we used a raster terrain extracted from the SRTM grid, a DEM that represents the earth surface from 60° North to 56° South [12]. The extracted raster covers a square region of approximately 2160×2160 kilometers and includes the entire Indus river basin–see Fig. 1. The raster consists of 24,000×24,000 cells, and the dimension of each square cell is approximately 90 meters. We refer to this dataset as `indus`.

Since the `indus` DEM does not contain any river data, we extracted the river cells based on the upstream area of each cell, in the same way as we did for the `denmark` dataset in Section 4.2. In this case we used an upstream area of 300 km^2 since it produces a visual result that matches the shape of the local river network, as it appears in orthophotos acquired when there was no flood in the region.

To evaluate the ability of our algorithms to accurately model floods, we used a vector dataset that shows the actual flooded regions around the river during the Indus river flood. This dataset was released by the Dartmouth Flood Observatory, and contains data acquired with MODIS (Moderate-resolution Imaging Spectroradiometer) technology [9]. We refer to this dataset as `flood`. The `flood` dataset was constructed based on several satellite photos of the Indus region, acquired during the period from the 1st to the 5th of August of 2010. It represents with polygons all the regions that were flooded in at least one day during this period. The bounding box of `flood` covers a rectangular region that spans approximatelly 1118 and 911 kilometers on the langitudinal and the longitudinal axes respectively. It contains 4294 polygons, and the total area covered by these polygons is approximatelly 30483 km^2. Refer to Fig. 1.

We ran the implementations of *ProximityResistance* and *UpstreamResistance* algorithms on the `indus` DEM and the extracted river cells, and we evaluated the output of each algorithm using a method that resembles the *Area-Under-the-Curve* (also known as AUC) measure, which is one of the most popular measures for model testing [5]. In particular, we overlayed `flood` with `indus` and extracted the cells in `indus` whose centers lie in the interior of a polygon in `flood`. We refer to these cells as the *flooded* cells of `indus`. In total, we identified 4045544 flooded cells. Next we selected at random a large set of pairs of cells. Each pair was selected so that it consists of one flooded cell and one non-flooded cell. We denote this set of pairs by $\mathcal{P}$. For each of our methods, we determined for each pair $pr \in \mathcal{P}$ if the flooded cell in pr scores a higher resistance value than the non-flooded cell, and calculated the percentage of the pairs in $\mathcal{P}$ for which this condition holds. We call this percentage the *output quality* of the method. The value of the output quality is an estimation of the AUC measure; the output quality value is equal to the AUC if $\mathcal{P}$ consists of all possible pairs of flooded/non-flooded cells in the region of interest. For our study, we chose 10^5 pairs, considering that this is a sufficient number for estimating the value of the AUC. For method *ProximityResistance* the output quality is 87%, while for *UpstreamResistance* the output quality is 92%. This shows clearly that both of the methods produce flood resistances that are highly consistent with the actual event.

To measure how the two methods perform on a more local scale, we calculated their output quality within several smaller regions. More specifically, within the xy-region covered by `flood` we extracted three sets of square windows, each set consisting of windows of certain size. In the first set each window is a square with dimension 20 km, in the second set each window has dimension 40 km, and the third set consists of windows of 80 km dimension. The windows of each set were picked in the following way. Within the region covered by `flood` we extracted at random 500 windows of the same size. Then we used a greedy algorithm to select a subset of these windows, so that there is no pair of windows in the subset that overlap with each other, and so that each window contains at least 500 flooded and at least 500 non-flooded cells. Thus, we ended up with a subset of 119 windows for the first set, and forty-five and twenty-two windows for the second and third set respectively. From each window, we selected 10^5 cell pairs, again so that each pair contains one flooded and one non-flooded cell. We then calculated the output quality of our methods for each window. Figure 4.3 shows the results for the windows of 20 km dimension, where the mean output quality was 61% for *ProximityResistance* and 71% for *UpstreamResistance*. For windows of 40 km dimension, *ProximityResistance* attained mean output quality 69% and *UpstreamResistance* mean output quality 81%. For the third set of windows, the values were 76% and 85%, respectively.

Therefore, for each window size *UpstreamResistance* has higher mean output quality than *ProximityResistance*. For both methods the output quality increases as the window size becomes larger. Yet, we observed that for all examined window sizes, there exist windows were at least one of the methods has an output quality value of less than 50%.

To examine the above further, we investigated if there is a correlation between the output quality values and the two following factors: heterogeneity of the terrain (variability of elevation values) and the number of flooded cells inside

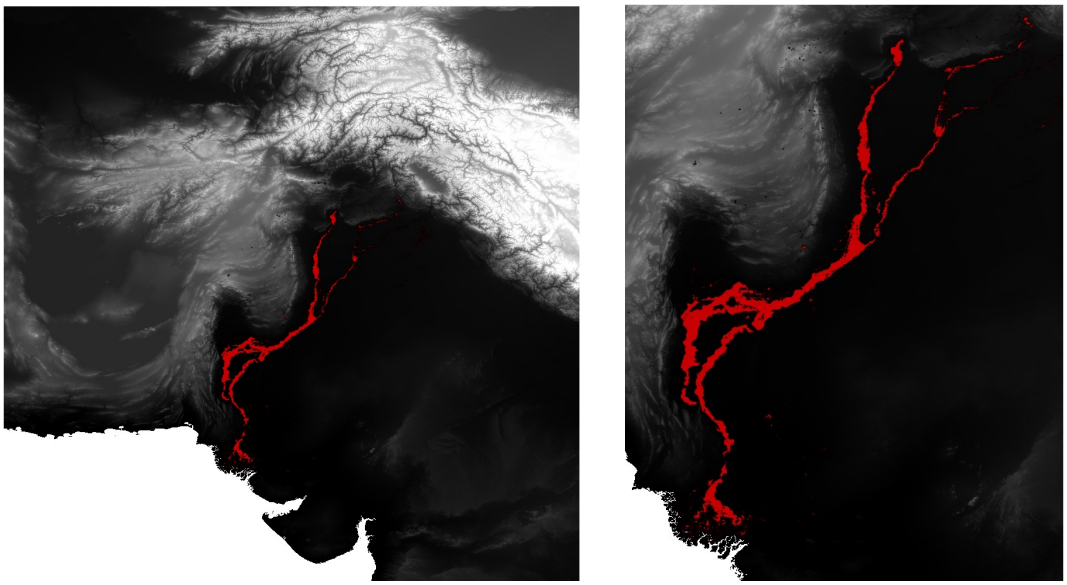


Figure 1: (a) An illustration of the `indus` DEM together with the `flood` vector dataset. The cells of the DEM appear in grayscale colours, shaded according to their elevation values; cells of higher elevation are indicated by lighter shades. The polygons of the `flood` dataset appear in red colour. (b) A closer view of the flooded regions.

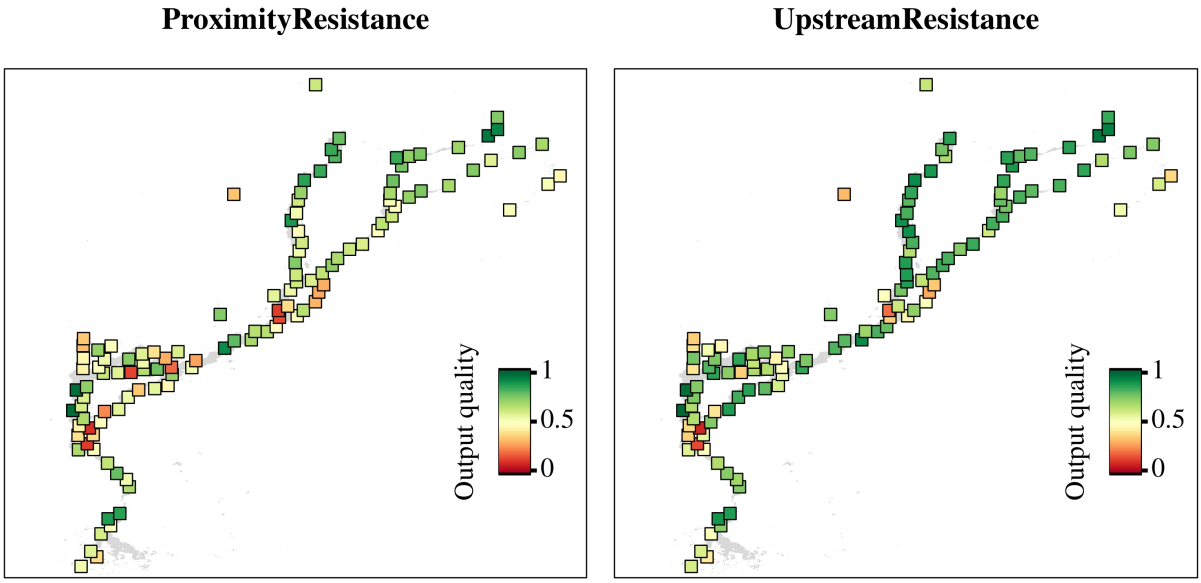


Figure 2: The locations for the selected windows of 20 km dimension. In each subfigure, a window is represented by a colored box. Each box is colored according to the output quality value achieved by one of the methods for the corresponding window. The relative size of the boxes in the figure is larger than the size of the original windows. We did this to make the color of each box more visible. The xy-regions of the original windows do not overlap with each other. Left: boxes colored based on the output quality values for `ProximityResistance`. Right: boxes colored according to the output quality values of `UpstreamResistance`.

each window. To measure the heterogeneity of the terrain within each window w, we computed the logarithm of the standard deviation for the elevations of the cells in w. We call this value the *topographic heterogeneity* of w. In order to examine visually the relation between the output quality and the topographic heterogeneity among the different windows, we created a scatter plot for each method. Each scatter plot contains a 2-dimensional point $p(w)$ for every window w; the horizontal coordinate of $p(w)$ is equal to the topographic heterogeneity of w, and the vertical coordinate of this point is equal to the output quality of the method for w. Figure 4.4 shows the scatter plots that we produced for windows of 20 km dimension. In a similar way, we created a scatter plot for each method where the horizontal coor-

dinates of the presented points are equal to the number of flooded cells in the windows that we examine. These scatter plots appear also in Figure 4.4. It becomes evident that both of the methods score higher output quality values for windows of intermediate topographic heterogeneity. Most of the low output quality values appear on windows of small heterogeneity. Regions that consist mainly of flat areas belong to this category. Also, there does not appear to be any relation between the output quality of the methods and the number of flooded cells within each window. The visualisations that we produced for the windows of larger size showed similar patterns.

4.4 Comparing the Output of the Methods

To gain more insight about *ProximityResistance* and *UpstreamResistance*, we visually examined the output that the two methods produced for the `denmark` dataset. During this examination, for various rise values ρ we extracted the regions in the output of each method which consisted of all cells with flood resistance $\leqslant \rho$. The first observation we made was that for the same rise value the size of the flooded area that appears in the output of *ProximityResistance* is larger than in the output produced by *UpstreamResistance*. Refer to Figure 4(a) and Figure 4(b). One of the reasons that contributes to this difference is how the two methods estimate river floods around coastlines; in the output of *ProximityResistance*, almost the entire coastline of the terrain appears flooded even for very small rise values. Refer to Figure 4(c). We can explain this as follows. Recall that with *ProximityResistance* a cell c gets flooded for a rise value ρ if a) this cell has a height difference $\leqslant \rho$ from the closest river cell on the xy-domain (the obstruction value), and b) if there is a path from c to any river cell such that the obstruction values of all cells in the path is $\leqslant \rho$. The terrain cells which appear close to the coastline have low height values, since they lie almost on the sea level. Therefore, for each such cell the height difference from the closest river cell is either very small or negative, which means that the coastline constitutes a path of cells that connects to the river and all cells in this path have very low obstruction values. As a consequence, even for small rise values all cells in this path are flooded when using *ProximityResistance*. On the other hand, in the output of the *UpstreamResistance* method, coastlines do not appear flooded even for large rise values. The reason is that for a coastline cell there is usually no flow path that connects this cell with a river cell. Hence, cell c can not get flooded whichever the river-rise value. We also observed one more artefact in the output of method *ProximityResistance*; in some cases, this method produces flooded regions with long linear boundaries that do not correspond to actual obstacles on the elevation profile of the terrain. Refer to Figure 4(d). These artefacts are the result of assigning obstruction values to non-river cells based on the Voronoi diagram of the river cells on the xy-domain of the terrain. In an area that extends between two different river streams, this step may produce two regions of cells that have a large difference in their obstruction values. The boundary between these two regions follows the boundaries between Voronoi regions of river cells that belong to different streams. As a consequence, for certain rise values there appear flooded areas in the output whose boundary follows the boundary between the Voronoi regions of the river cells.

5. REFERENCES

[1] P. K. Agarwal, L. Arge, and K. Yi. I/O-Efficient Batched Union-Find and its Applications to Terrain Analysis. In *Proc. 22nd Annual Symposium on Computational Geometry*, pages 167–176, 2006.

[2] P. K. Agarwal, L. Arge, and K. Yi. I/O-Efficient Construction of Constrained Delaunay Triangulations. In *Proc. 13th Annual European Symposium on Algorithms*, pages 355–366, 2005.

[3] A. Aggarwal and J. S. Vitter. The Input/Output Complexity of Sorting and Related Problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[4] L. Arge, L. Toma, and J. S. Vitter. I/O-Efficient Algorithms for Problems on Grid-Based Terrains. *ACM Journal on Experimental Algorithmics*, 6(1), 2001.

[5] U. Brefeld and T. Scheffer. AUC maximizing support vector learning. In *Proc. ICML Workshop on ROC Analysis in Machine Learning*, 2005.

[6] Encyclopaedia Britannica Webpage, Pakistan Floods of 2010. `http://www.britannica.com/EBchecked/topic/1731329/Pakistan-Floods-of-2010`.

[7] Elevation Model of Denmark, Geodata Agency of the Danish Ministry of Environment. `http://gst.dk/emner/frie-data/hvilke-data-er-omfattet/hvilke-data-er-frie/dhm-danmarks-hoejdemodel/`.

[8] A. Danner, T. Mølhave, K. Yi, P.K. Agarwal, L. Arge, and H. Mitasova. TerraStream: From Elevation Data to Watershed Hierarchies. In *Proc. 15th ACM International Symposium on Advances in Geographic Information Science (ACM GIS)*, pages 212–219, 2007.

[9] The Dartmouth Flood Observatory Webpage. `http://floodobservatory.colorado.edu/`.

[10] M.T. Goodrich, J.J. Tsay, D.E. Vengroff and J.S. Vitter. External-Memory Computational Geometry. In *Proc. 34th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pages 714–723, 1993.

[11] B. Kronvang, M. Søndergaard, C.C. Hoffmann, H. Thodsen, N. Bering Ovesen, M. Stjernholm, C. B. Nielsen, C. Kjærgaard, B. Schønfeldt, and B. Levesen. Etablering af P-Ådale. Technical report 840, National Environmental Research Institute of Denmark, 2011.

[12] E. Rodriguez, C.S. Morris, and J.E. Belz. (2006). A Global Assessment of the SRTM Performance. *Photogrammetric Engineering & Remote Sensing*, 72(3):249–260, 2006.

[13] TPIE, the Templated Portable I/O Environment. `http://www.madalgo.au.dk/tpie/`.

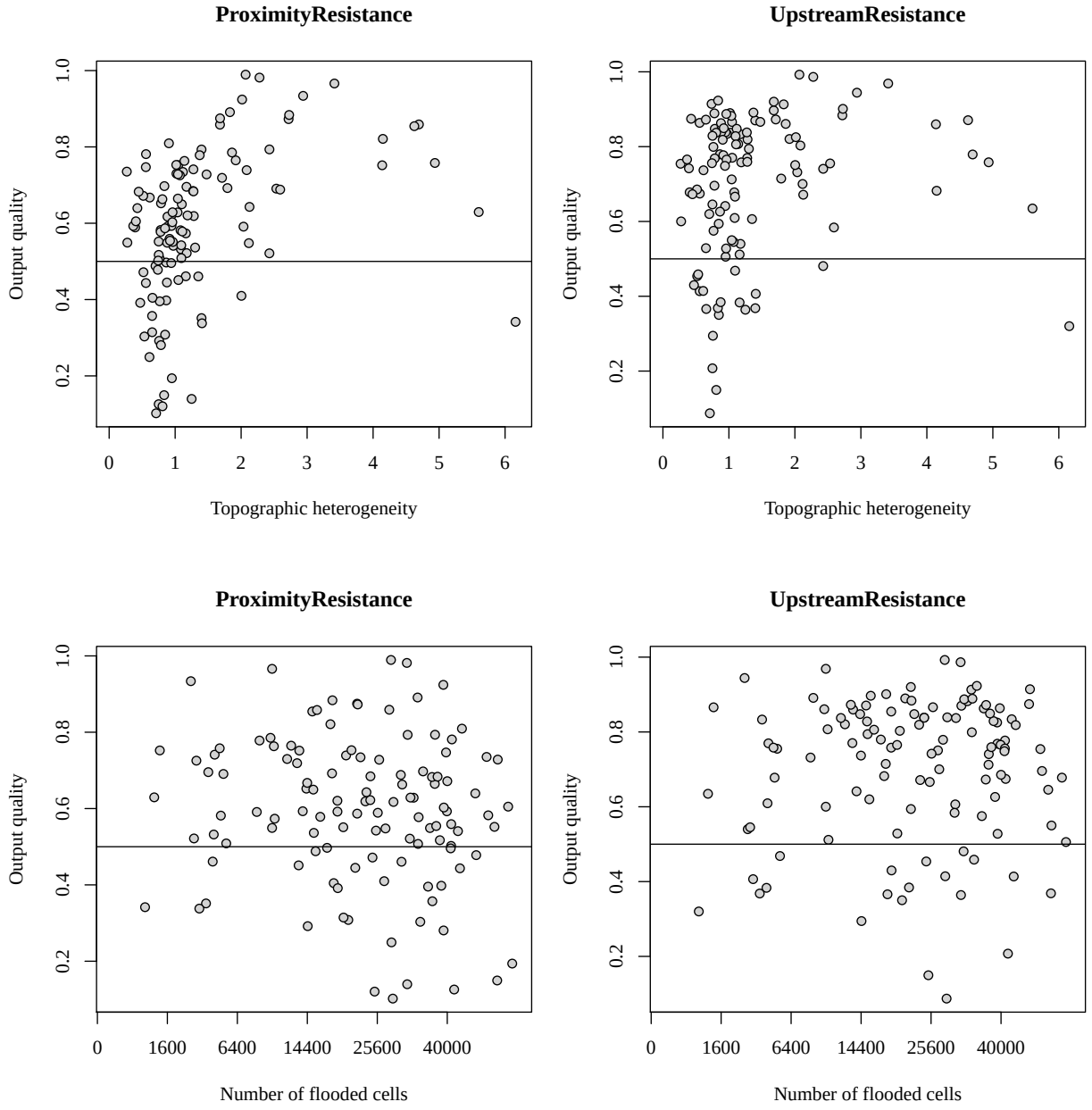
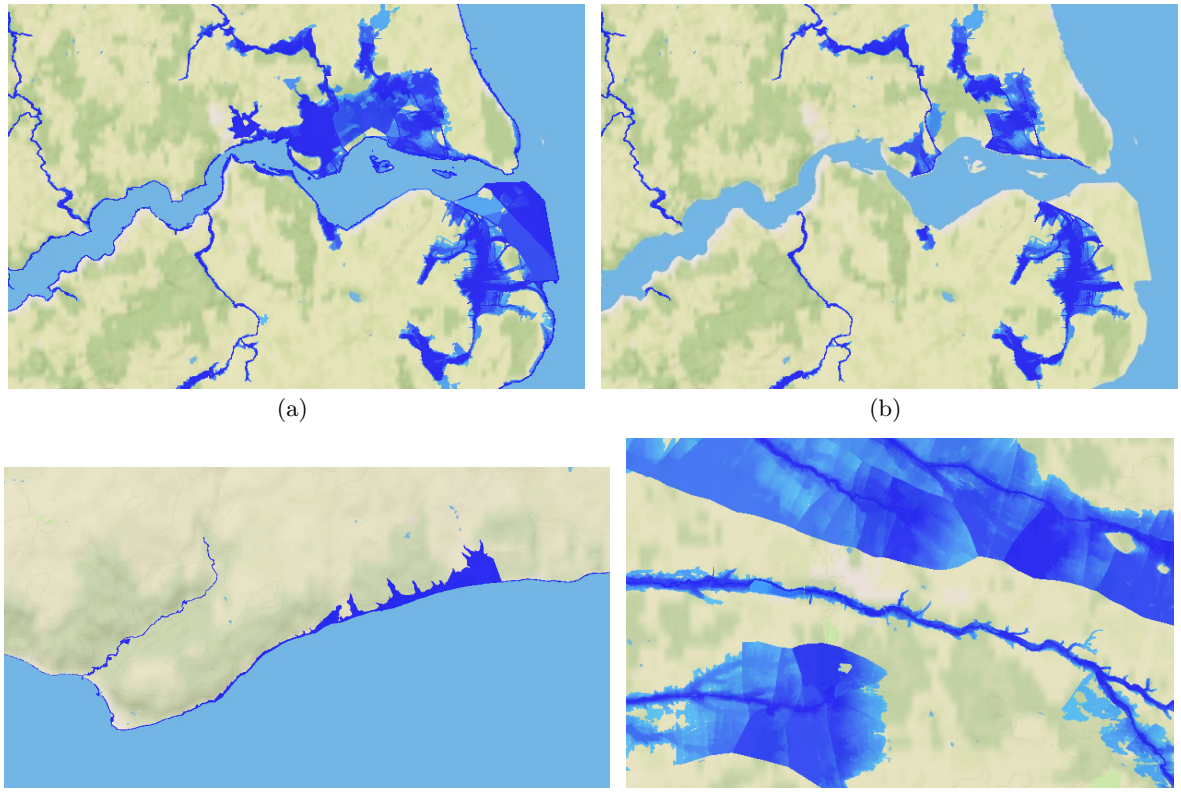


Figure 3: Scatter plots that show the relation between the output quality of each method and two features of the examined windows. Each point corresponds to a window of 20 km dimension. Top: The relation between output quality and topographic heterogeneity. Bottom: The relation between output quality and the number of flooded cells in each window.



(a) (b)

(c) (d)

Figure 4: An illustration of the outputs of *ProximityResistance* and *UpstreamResistance* for the `denmark` dataset. Flooded regions are indicated by dark blue color. (a) The output of the *ProximityResistance* around Hadsund town (north-east Jutland) for a rise value of half a meter. (b) The output of *UpstreamResistance* for the same region and rise value. (c) The output of *ProximityResistance* close to Vejle city with a rise value of just one milimeter. The entire coast appears flooded, with wide flooded areas at certain places. (d) The output of the *ProximityResistance* on a region with several river streams, for a rise value of 2.8 meters.

APPENDIX

In Section 3.1 we claimed that, given a grid $\mathcal{G}$ and the set of river cells $R(\mathcal{G})$ on this grid, we can compute for every cell $c \in \mathcal{G}$ its source cell using only $O(\text{scan}(N))$ I/Os. Recall that the source cell of c is the cell in $R(\mathcal{G})$ that has the closest xy-distance to c. Next we describe in detail how we can do this computation while achieving the claimed performance bound.

Instead of computing explicitly the Voronoi diagram of centers of the cells in $R(\mathcal{G})$, we use a simpler approach. For every cell $c = \mathcal{G}(i, j)$ we compute three cells: The closest river cell $\mathcal{G}(k, l)$ such that $k < i$, the closest river cell $\mathcal{G}(k', l')$ such that $k' > i$, and the closest river cell $\mathcal{G}(k'', l'')$ such that $k'' = i$. We indicate these cells by $\text{source}_<(c)$, $\text{source}_>(c)$, and $\text{source}_=(c)$, respectively. Obviously, one of these three cells is the source cell of c; given those cells we can determine $\text{source}(c)$ by simply comparing their xy-distances from c. We can calculate these three cells for every cell in $\mathcal{G}$ in $O(\text{scan}(N))$ I/Os using a sweep-line technique. Next we show how we can do this for $\text{source}_<(c)$ and $\text{source}_=(c)$; the process for computing $\text{source}_>(c)$ is quite similar to the one for $\text{source}_<(c)$.

Let $\text{row}(i)$ denote the i-th row in $\mathcal{G}$, where $\text{row}(1)$ is the bottom row in the raster. Let $\mathcal{VD}(i)$ indicate the Voronoi diagram on the xy-domain of $\mathcal{G}$ where the sites are the centers of all river cells $\mathcal{G}(k, l)$ such that $k < i$. For every cell $c \in \text{row}(i)$, the cell $\text{source}_<(c)$ corresponds to the site in $\mathcal{VD}(i)$ whose Voronoi region contains the center of c. Based on this observation, to compute $\text{source}_<(c)$ for every cell $c \in \mathcal{G}$ we scan $\mathcal{G}$ row by row, starting from the bottom row of the raster. During this process, we maintain a sweepline l_s parallel to the y-axis. When we process $\text{row}(i)$, we consider that the sweepline has the same y-coordinate as the centers of the cells of this row. We use $l_s(i)$ to indicate the sweepline at that moment. Line $l_s(i)$ intersects the Voronoi regions of certain sites in $\mathcal{VD}(i)$ and is thus subdived into horizontal segments, each contained in a different region. To represent this we compute a list of points L_i; these are the intersection points between $l_s(i)$ and the voronoi boundaries of the sites that appear below $l_s(i)$. With every point $p \in L_i$ we also store at most two river cells $rc_{\text{left}}(p)$ and $rc_{\text{right}}(\text{p})$. Cell $rc_{\text{left}}(p)$ is the river cell whose Voronoi region intersects l_s to the left of p, while $rc_{\text{right}}(p)$ is the river cell whose Voronoi region intersects l_s to the right of this point. The points in L_i are stored in order of increasing x-coordinate. Given the list L_i, it is easy to compute the source cells for every cell c in $\text{row}(i)$; we scan this row and L_i simultaneously, while keeping track of the cell centers in $\text{row}(i)$ that fall inside the Voronoi region delimited by two consecutive points in L_i.

Computing L_i for each row is a bit more involved. We can do this efficiently as follows; to calculate the elements in L_i we look at the list L_{i-1} that we computed for the previous row. First, we construct an intermediate list L_{temp} that has a structure similar to L_i except that it stores the intersections points between $l_s(i)$ and the Voronoi regions in $\mathcal{VD}(i-1)$. Then, we update this intermediate list so that also the regions of the sites in $\text{row}(i-1)$ are taken into account.

To construct L_{temp} we need to find where the boundaries of the Voronoi regions of $\mathcal{VD}(i-1)$ intersect $l_s(i)$. To do so for every region boundary that intersects $l_s(i-1)$ we could simply compute where this boundary intersects $l_s(i)$. However, not all region boundaries in $\mathcal{VD}(i-1)$ that intersect $l_s(i-1)$ intersect also $l_s(i)$; this is the case when there exist regions in $\mathcal{VD}(i-1)$ whose y-span ends somewhere between $l_s(i)$ and $l_s(i-1)$. This can be handled in the following way; we scan list L_{i-1} and we maintain an I/O-efficient stack ST that stores are the bisectors between Voronoi sites in $\mathcal{VD}(i-1)$ whose regions potentially intersect $l_s(i)$. More specifically, let p_{curr} be the element that is currently scanned in L_{i-1}. We maintain the invariant that ST stores the bisectors for only those sites in $\mathcal{VD}(i-1)$ whose Voronoi regions: a) intersect $l_s(i-1)$ to the left of p_{curr}, and b) would intersect $l_s(i)$ if we did not consider the sites in $\mathcal{VD}(i-1)$ whose regions intersect $l_s(i-1)$ to the right of p. We also maintain that for any two lines l_1 and l_2 in the stack, line l_1 is stored below l_2 if and ony if l_1 crosses $l_s(i)$ to the left of l_2. With each line l that we store in ST we also maintain the two sites $rc_{\text{left}}(l)$ and $rc_{\text{right}}(l)$ for which l is the bisector. Initially ST is empty. Recall that p_{curr} is the intersection point between the sweepline $l_s(i-1)$ and the bisector line of two neighbouring sites $rc_{\text{left}}(p)$ and $rc_{\text{right}}(p)$. Let l_{bisect} be this line for p_{curr}, the element currently processed in L_{i-1}. Let $\text{top}(ST)$ indicate the bisector currently stored at the top of the stack ST. We compute the intersection point between l_{bisect} and $\text{top}(ST)$; if this point has a y-coordinate that falls above $l_s(i)$, or below $l_s(i-1)$ we push l_{bisect} into the stack and we continue with the next element in L_{i-1}. Otherwise, we compute the bisector between $rc_{\text{left}}(\text{top}(ST))$ and $rc_{\text{right}}(p)$ and we set l_{bisect} to represent this line. Then, we remove the line at the top of the stack and compute the intersection point between the current l_{bisect} and the line that is now stored $\text{top}(ST)$. We repeat this process until either l_{bisect} and $\text{top}(ST)$ intersect at a point above $l_s(i)$ or below $l_s(i-1)$, or ST becomes empty. It is easy to prove that after processing all the elements in L_{i-1}, stack ST stores the support lines of exactly those Voronoi boundaries in $\mathcal{VD}(i-1)$ that intersect $l_s(i)$.

Given the data stored in ST we can easily construct the intermediate list L_{temp}; recall that this is the list that stores the intersection points between $l_s(i)$ and the bisectors of the Voronoi regions in $\mathcal{VD}(i-1)$. To construct L_i, we then process L_{temp} together with the river cells in $\text{row}(i-1)$. First, we compute the Voronoi diagram of the river cells in $\text{row}(i-1)$; this diagram is trivial to compute since all sites have the same y-coordinate, and the bisectors between the Voronoi regions are lines parallel to the y-axis. For each river cell c in $\text{row}(i-1)$ we compute the interval which corresponds to the x-span of the Voronoi region of c in this diagram. We store each such interval together with corresponding cell in a list L_{int}, in increasing order of the x-coordinates of their endpoints. Next, we use the intervals in L_{int} and the information stored in L_{temp} in order to compute the elements of L_i, the intersection points between the river cells in $\mathcal{VD}(i)$ and $l_s(i)$. For this, we need to partition $l_s(i)$ into intervals, such that each interval corresponds to the intersection of $l_s(i)$ with a Voronoi region in $\mathcal{VD}(i)$. Each such interval is either a subset of an interval in L_{int}, or a subset of an interval represented by two consecutive elements in L_{temp}. Therefore, we can compute L_i by simultaneously scanning L_{temp} and L_{int}, and substituting, inserting, or deleting points from L_{temp} using the information stored in L_{int}.

Let k be the number of cells in a single row of $\mathcal{G}$. From the above description, we conclude that constructing each list L_i requires $O(\text{scan}(k))$ I/Os since we need to scan $\text{row}(i-1)$,

list L_{i-1}, and L_{temp} a constant number of times. We also have to insert and extract at most $O(k)$ elements to and from the I/O-efficient stack ST. We can insert or extract a single element from such a stack in $O(1/B)$ I/Os, which sums up to $O(\text{scan}(k))$ I/Os for processing all cells in a row.

Processing L_i to determine the values $\text{source}_<$ also requires one scan, which adds up in total to $O(\text{scan}(N))$ I/Os for handling the corresponding lists for all rows in $\mathcal{G}$.

Computing $\text{source}_=(c)$ is a simple task; let c be any cell in $\mathcal{G}$, and let r be the row in $\mathcal{G}$ which contains c. Cell $\text{source}_=(c)$ is either the closest river cell on r that appears on the left side of c, or the closest river cell from the right side of c. Therefore, to determine $\text{source}_=(c)$ we scan each row of $\mathcal{G}$ independently, and for each cell that we are currently scanning we keep track of the nearest river cell from each side on this row. Hence, we can compute cell $\text{source}_=(c)$ for every cell $c \in \mathcal{G}$ in $O(\text{scan}(N))$ I/Os. From the above description, we conclude that we can compute $\text{source}_>(c)$, $\text{source}_<(c)$, $\text{source}_=(c)$, and therefore the source cell $\text{source}(c)$ for every cell $c \in \mathcal{G} \setminus R(\mathcal{G})$ in $O(\text{scan}(N))$ I/Os.