Algorithms for Massive Terrains and Graphs

Svend Christian Svendsen

PhD Dissertation

Department of Computer Science
Aarhus University
Denmark

Algorithms for Massive Terrains and Graphs

A Dissertation
Presented to the Faculty of Natural Sciences
of Aarhus University
in Partial Fulfillment of the Requirements
for the PhD Degree

by
Svend Christian Svendsen
July 29, 2021

Abstract

In this thesis, we describe several results for handling massive terrain models. Due to advances in remote sensing technology, it has become possible to acquire terrain data with high precision and speed. As a consequence of collecting and processing such data, highly detailed terrain models have become readily available with several applications for terrain analysis, such as modeling the flow of water on the surface of the terrain. Analyzing the flow of water enables the prediction of flooding during extreme weather events with heavy rain. In order to obtain a realistic and precise result of the analysis, it is crucial to use a sufficient level of detail in the model. However, highly detailed models can be orders of magnitude larger than the main memory of a computer and can be very difficult to analyze with traditional tools. This has lead to the development of I/O-efficient algorithms that process large models by keeping only a fraction of the data in main memory. By carefully designing I/O-efficient algorithms that minimize the number of accesses to disk, one can obtain tools that can process massive amounts of data efficiently, even on commodity hardware.

We first present a software framework that enables efficient implementation of I/O-efficient algorithms by providing a set of tools for easy modularization of code with low run-time overhead. We then present algorithms for the problem of estimating the rate at which water flows on the surface of the terrain during heavy rain. For each point on the terrain model, our algorithm computes a function that describes how much water flows over the point at any given time. Furthermore, we describe how this can be used to model the geometry of rivers flowing on the terrain. We then present an algorithm for efficiently dividing a large terrain into smaller regions such that the boundaries between regions are small. As an example of why such an algorithm is useful, we show how it enables us to efficiently model flow accumulation on the terrain model by scanning over the smaller regions.

Finally, we present a machine learning-based solution that efficiently detects hydrological corrections on a terrain. Hydrological corrections are modifications made on terrain models to accurately model the flow of water. This includes the identification and removal of bridges that impede the flow of water on the terrain model by forming false hydrological barriers. We demonstrate that our solution results in a highly accurate list of modifications on a given terrain.

Resumé

I denne afhandling beskriver vi adskillige algoritmer til håndtering af enorme terrænmodeller. Grundet udviklingen i fjernmålingsteknologi er det blevet muligt at indsamle terrændata med stor præcision og hastighed. Indsamling og behandling af dette data har resulteret i at meget detaljerede terrænmodeller er blevet let tilgængelige med adskillige anvendelser inden for terrænanalyse, så som modellering af vandstrømning på terrænoverflader. Analysering af vandstrømning gør det muligt at forudse oversvømmelse under ekstreme vejrfænomener med voldsom regn. For at opnå en realistisk og præcis analyse er det afgørende at anvende en terrænmodel med en tilstrækkelig detaljeringsgrad. En model med stor detaljeringsgrad kan dog være mange størrelsesordener større end den interne hukommelse af en computer og være svær at analysere med traditionelle værktøjer. Dette har ført til udviklingen af I/O-effektive algoritmer, der håndterer store modeller ved kun at opbevare en brøkdel af modellen i intern hukommelse. Ved omhyggeligt at udlede sådanne I/O-effektive algoritmer der tilgår harddisken minimalt, kan man opnå værktøjer der kan håndtere massive mængder data effektivt.

Vi præsenterer først et software framework der muliggør effektiv implementering af modulære I/O-effektive algoritmer med lille overhead. Dernæst beskriver vi algoritmer til at estimere mængden af vand der strømmer på overfladen af et terræn under voldsom regn. For hvert punkt på terrænet beregner vores algoritme en funktion der beskriver hvor meget vand der strømmer henover punktet på et givent tidspunkt. Derudover beskriver vi hvordan dette kan bruges til at modellere geometrien af floder på terrænet. Vi præsenterer dernæst en algoritme til at effektivt opdele et terræn i mindre regioner således at grænsen mellem regionerne er lille. Vi demonstrerer at dette gør det muligt effektivt at modellere *flow accumulation* ved at behandle regionerne én ad gangen.

Til sidst præsenterer vi en algoritme, baseret på *machine learning*, til effektivt at detektere hydrologiske tilpasninger på et terræn. Hydrologiske tilpasninger er modifikationer til en terrænmodel, der resulterer i en mere nøjagtig modellering af vandstrømning på modellen. Dette inkluderer identifikation og fjernelse af broer der blokerer for vandstrømning, da de danner forkerte hydrologiske barrierer i modellen. Vi demonstrerer at vores løsning resulterer i en særdeles præcis liste af tilpasninger på en given terrænmodel.

Acknowledgments

I am immensely grateful and owe my thanks to the many people who have helped me during this journey. A special thanks to my two advisors Lars Arge and Gerth Stølting Brodal. Lars hired me as a student programmer eight years ago when I was starting my studies. His passion for research and his drive to make everything I/O-efficient helped spark my interest in research. Through Lars' deep insight, guidance, and red pen, I was introduced to many fascinating topics and the world of research. I am extremely thankful for Gerth offering to advise me after Lars passed away. His support during a very tough and difficult part of my studies has been invaluable.

In the fall of 2019, I visited Duke University where I was hosted by Pankaj K. Agarwal. I wish to thank Pankaj and his student Aaron Lowe for fruitful discussions and for welcoming me into your group.

A big thanks to the people at SCALGO for providing help on many practical issues and bringing up new ideas for research. I wish to particularly thank Mathias Rav and Jakob Truelsen for being a tremendous help.

Thanks to the community in the algorithms group and at the Department of Computer Science. I would especially like to thank Casper Freksen, who has been my office mate for most of my Ph.D. It has been great to have someone who understands and shares the struggles, frustrations, and joys experienced during a Ph.D.

A very special thanks to my parents and my sister for their love and support. You have always nurtured my curiosity and listened to my frustrations. Finally, special thanks to my friends and Århus Judo Klub for helping me take my mind off academia and beating me up on a weekly basis.

Svend Christian Svendsen,
Aarhus, July 29, 2021.

Contents

Abstract **i**

Resumé **iii**

Acknowledgments **v**

Contents **vii**

I Overview **1**

1 Introduction **3**
1.1 Outline of Thesis 5

2 Survey of Important Results **7**
2.1 External Memory Algorithms 7
2.2 Terrain Definitions 10
2.3 Flow Model Definitions 16
2.4 Terrain Analysis Algorithms 18
2.5 External Memory Algorithms in Practice 25
2.6 Our Contributions 27

II Publications **31**

3 External Memory Pipelining Made Easy With TPIE **33**
3.1 Introduction 34
3.2 An Example Problem 38
3.3 TPIE Pipelining 48

4 Practical I/O-Efficient Multiway Separators **55**
4.1 Introduction 56
4.2 Preliminaries 58
4.3 Multiway Separator Algorithm for k-ply Systems 61
4.4 Applications to Delaunay Triangulations and Terrain 67

4.5 Experiments 68
4.6 Appendix: Proof of Constant VC dimension 71
4.7 Appendix: Bound on the Total Number of Boundary Balls . . . 71
4.8 Appendix: Experimental Evaluation of Separator Size 72
4.9 Appendix: Algorithm with Larger Sample 74

5 1D and 2D Flow Routing on a Terrain 79
5.1 Introduction 79
5.2 Preliminaries & Models 84
5.3 Terrain-flow Query 88
5.4 I/O-Efficient Algorithms 96
5.5 Vertex-Flow Query 101
5.6 Extracting 2D Flow Networks 105
5.7 Experiments 114
5.8 Conclusion 124

6 Learning to Find Hydrological Corrections 127
6.1 Introduction 127
6.2 The Data 132
6.3 Segmenting Tiles with Neural Networks 133
6.4 Complete Algorithm 135
6.5 Experiments and Results 142
6.6 Conclusion and Future Work 146

Bibliography 147

Part I

Overview

Chapter 1

Introduction

Advances in technology have made it possible to acquire massive amounts of data with high precision and speed. The availability of such data provides many new opportunities for both scientific and commercial use. However, due to the sheer volume of data, processing and analyzing it can be extremely difficult and require the development of specialized tools. A particularly interesting example is the collection and analysis of terrain data. Traditionally, terrain data have been collected manually by land surveyors in the field. This process is both time-consuming and expensive. However, due to advances in remote sensing technologies such as *Light Detection and Ranging (LiDAR)*, highly detailed terrain data can be collected in a much faster and more precise process. An example is the Danish Elevation Model which has been made publicly available by the Danish Agency for Data Supply and Efficiency [50]. The model is collected using an aircraft equipped with LiDAR sensors that measure the distance between the aircraft and the surface. This technique produces a highly detailed point cloud that describes the elevation of points on the terrain. The resulting data set has an average density of 4.5 points per square meter with a vertical error of only 5 centimeters. The point cloud of Denmark proper contains 415 billion points in total.

The terrain impacts numerous areas of our lives, and therefore accurate elevation models have a wide area of applications. An example of this is modeling how water flows on the surface of a terrain. During extreme weather events with heavy rain, water might exceed the capacity of sewer and drainage systems. In this case, water starts to collect on the surface of the terrain and accumulate in depressions. Being able to predict where water accumulates on the terrain is extremely important, as it enables emergency personnel and property owners to mitigate damage from flooding. The rate at which water accumulates in a depression on the terrain depends on several aspects. Initially, the fill rate depends on the volume of the depression and the amount of rain falling on the *upstream area*. The upstream area of a depression is the area from which water flows into a depression. However, once a depression becomes full,

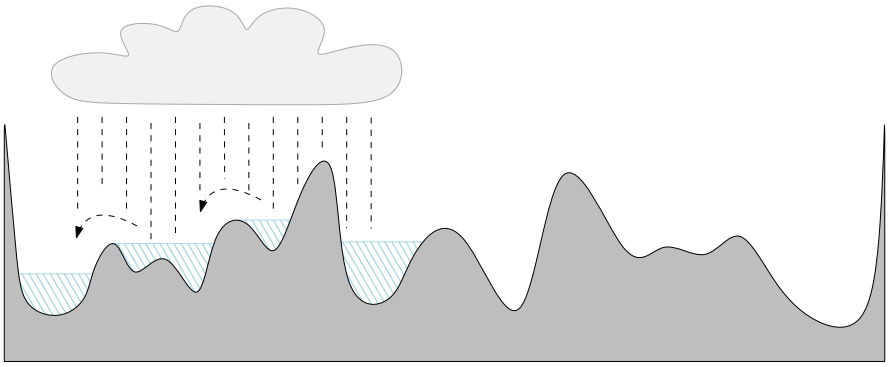


Figure 1.1: A flash flood event where rain falls on part of a terrain which causes depressions to fill and spill into adjacent depressions.

the water will start spilling into an adjacent depression, effectively increasing the size of the upstream area of the depression into which it spills. This can lead to a large and sudden increase in the rate at which a depression fills, also known as a *flash flood event*. See Figure 1.1 for illustration.

In order to predict flash flood events, one can use data from previous flooding events combined with forecasts on future extreme weather events. However, the paths in which water flows can change significantly with new construction projects such as bridges and highways. Thus, in order to efficiently mitigate flooding, one must be able to predict the impact of such projects before construction begins. By using a digital elevation model of the terrain, one can compute such predictions. However, to obtain realistic results, it is crucial that the model has a sufficient level of detail. If the model is not detailed enough, hydrological barriers and waterways might be missing from the model, leading to inaccurate predictions. Furthermore, the elevation model must be large enough to cover the entire upstream area of the area of interest. Thus, using models that cover a smaller area might not be sufficient.

When processing such an elevation model on a computer, the model is first moved to internal memory before computation can happen on the data. On modern hardware, the external memory is orders of magnitude slower than internal memory. Therefore, in order to speed up the computation, the hardware performs so-called *block accesses* that transfers data from and to external memory in *blocks* of consecutive data elements. The transfer of a block between the hard disk and internal memory is also known as an *I/O operation*. When working with highly detailed elevation models, the size of the data can be orders of magnitude larger than what fits in the internal memory of commodity hardware. This leads to many traditional algorithms encountering the problem of *thrashing* where the internal memory is full and blocks are frequently moved from and to external memory. This causes the performance of the algorithm to collapse. In order to prevent thrashing, algorithms can be engineered to keep only a small fraction of the data in internal memory and

minimize the number of I/O operations performed. In order to make the most use of a block, it is important to store data on the hard disk such that related elements are not spread out. That is, elements that are accessed at the same time by the algorithm must be laid out consecutively such that they can be read using a single I/O operation. By doing this, the algorithm can minimize the number of I/O operations performed by making the most out of each block. Such algorithms are called *I/O-efficient algorithms*.

1.1 Outline of Thesis

In this thesis, we develop algorithms for massive terrain data. The thesis is divided into two parts. In Part I, we provide a survey of relevant preliminaries within the research area and describe how our results contribute to the research area. Part II consists of publications as follows:

Chapter 3
External Memory Pipelining Made Easy With TPIE [19]
Lars Arge, Mathias Rav, Svend C. Svendsen, and Jakob Truelsen.
2017 IEEE International Conference on Big Data, Big Data 2017

Chapter 4
Practical I/O-Efficient Multiway Separators [94]
Svend C. Svendsen.
Manuscript

Chapter 5
1D and 2D Flow Routing on a Terrain [16]
Lars Arge, Aaron Lowe, Svend C. Svendsen, and Pankaj K. Agarwal.
Invited to ACM Transactions on Spatial Algorithms and Systems "Best Papers Special Issue of the ACM SIGSPATIAL 2020"

Chapter 6
Learning to Find Hydrological Corrections [14]
Lars Arge, Allan Grønlund, Svend C. Svendsen, and Jonas Tranberg.
Proc. of the 27th ACM International Conference on Advances in Geographic Information Systems, SIGSPATIAL 2019

Chapters 3 and 6 use the full arXiv versions of the papers [15, 20]. Additionally, we remark that a conference version of Chapter 5 has already been published [79]. The papers are included in their entirety with changes only to formatting, typesetting, and minor typos. The author of this thesis has contributed significantly to all papers. Finally, in accordance with the rules of the Graduate School of Natural Sciences, we inform the reader that Chapter 3 and Chapter 6 were also used in the progress report for the qualifying examination of the author.

In Chapter 3, we present a software framework for the implementation of I/O-efficient algorithms and data structures. This framework streamlines the implementation of I/O-efficient algorithms by enabling re-use and modularization of code. Furthermore, we provide implementations of commonly used data structures and algorithms. In Chapter 4, we present a practical algorithm for computing a division of a planar graph into regions such that each region fits in the internal memory of a computer. Furthermore, we show how this construction can be applied to the problem of efficiently modeling the flow of water on a terrain. In Chapter 5, we present a model for computing functions that estimate the amount of water that flows on points of the terrain. For each point on the terrain model, our algorithm computes the rate at which water flows over the point at any given time. We present both internal and external memory algorithms for this problem. Furthermore, we present a model for modeling the geometry of streams formed on the terrain during heavy rain events. In Chapter 6, we consider the problem of identifying *hydrological corrections* on a terrain. Loosely stated, a hydrological correction is a modification performed on the terrain which ensures that the flow of water is not impeded by a false hydrological barrier. This includes the removal of bridges and culverts which otherwise appear as dams in the data since elevation models, such as the Danish Elevation Model [50], are recorded from a Bird's-eye view. The identification of hydrological corrections has traditionally been performed by hand which is both an expensive and slow process. We present a machine learning-based solution that automatically and accurately detects and extracts the hydrological corrections of the terrain. The solution detects most of the manually labelled corrections and an additional number of corrections missing from our input data.

Chapter 2

Survey of Important Results

In this chapter, we provide a survey on I/O-efficient algorithms and terrain analysis. Additionally, we detail the contributions of the thesis. The chapter is structured as follows. In Section 2.1, we describe the I/O model of computation and several fundamental results within the model. In Section 2.2, we formally state the mathematical definitions used when describing algorithmic problems on terrain. We proceed in Section 2.3 by defining several models used for modeling the flow of water on the surface of a terrain. Section 2.4 surveys algorithmic problems and results related to terrain analysis and modeling the flow of water on a terrain. Finally, in Section 2.5, we describe the implementation of I/O-efficient algorithms, and in Section 2.6, we summarize the contributions of this thesis.

2.1 External Memory Algorithms

In theoretical computer science, it is important to accurately model the performance of algorithms. This is typically done by stating a *model of computation* that describes an abstraction with certain assumptions on how data is accessed. A widely used example of this is the *RAM model* which assumes that data is stored in an infinitely large internal memory that supports random access. That is, the time it takes to access a data element in internal memory is constant and does not depend on the location of the data element. The model supports operations on any data element in one time step. We define the performance of an algorithm as the *computation time* which is the total number of time steps used. This results in a model that strikes a good balance between simplicity and results that are applicable in practice.

In practice, computers do not have an infinitely large internal memory since internal memory can be quite expensive. Instead, modern hardware have a small amount of fast internal memory combined with a large but slow external memory, e.g. a hard disk or solid-state drive (SSD). Additionally, modern hardware have multiple levels of cache between the central processing

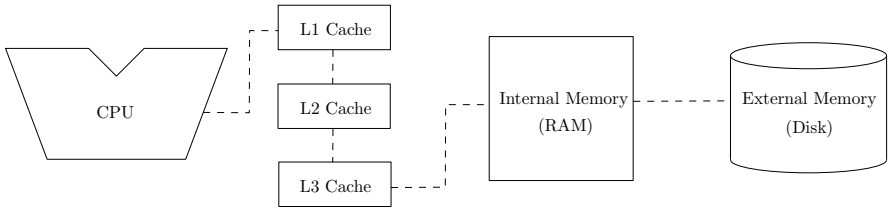


Figure 2.1: Illustration of the memory hierarchy of modern hardware when ignoring parallelism.

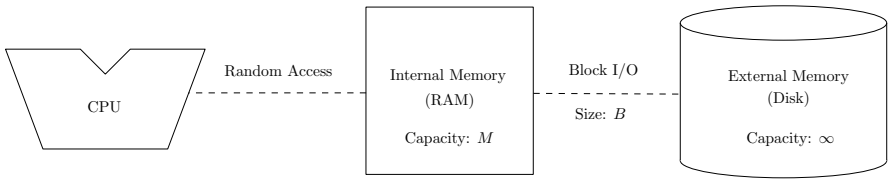


Figure 2.2: Illustration of the memory hierarchy in the I/O model.

unit (CPU) and the internal memory. See Figure 2.1 for an illustration of a *memory hierarchy* consisting of several levels of cache and memory. When performing computation on a data element, the computer retrieves the element from external memory into internal memory and cache. The movement of data from and to external memory can be quite slow and can become a bottleneck of the computation on very large data sets. In order to counter this problem, the movement of data from and to external memory is done in large *blocks* of consecutive data elements. Therefore, when working with large data sets, it is important to use a model of computation that represents the movement of blocks such that algorithms can be engineered to minimize this. Motivated by this, Aggarwal and Vitter introduced the *I/O model* of computation [6]. In the I/O model, the computer is equipped with a simplified memory hierarchy that has an infinitely large external memory and an internal memory of bounded size M. The data is initially stored consecutively on external memory and consists of N elements in total. In order to perform computation on a data element, it must first be transferred into internal memory. Data is transferred from and to external memory in blocks of B consecutive elements. We refer to this operation as an *I/O*. See Figure 2.2 for an illustration of the memory hierarchy in the I/O model. The performance of an algorithm in the I/O model is the total number of I/Os performed. Algorithms that minimize the number of I/Os performed are called *I/O-efficient algorithms* or *external memory algorithms*.

2.1.1 Sorting and Permutation

Since the introduction of the I/O model, results have been presented for many fundamental problems [11, 99]. It immediately follows from the model that

N consecutive elements can be read from external memory using $\text{Scan}(N) = O(N/B)$ I/Os by fetching blocks of B consecutive elements at a time. Aggarwal and Vitter [6] presented tight lower and upper bounds for the fundamental problems of sorting and permuting N elements. They presented an algorithm that sorts N consecutive elements using $\text{Sort}(N) = \Theta\big(\frac{N}{B}\log_{M/B}(N/B)\big)$ I/Os and proved the matching lower bound by assuming the comparison model for data elements in internal memory. Similarly, they showed that permuting N elements stored consecutively on external memory requries $\Theta(\min(N, \text{Sort}(N)))$ I/Os. In the RAM model, N elements can be permuted in $O(N)$ time by trivially swapping elements. Thus, it seems that the techniques and solutions applied in the RAM model do not trivially extend to the I/O model. We now proceed to describe techniques commonly used to solve fundamental graph problems.

2.1.2 Time-Forward Processing

A fundamental technique often used in the I/O model is the *time-forward processing technique* introduced by Chiang *et al.* [42]. In Section 2.4.5, we will describe an example of the time-forward processing technique. For now, we state various results obtained using the technique. Chiang *et al.* used the time-forward processing technique to solve the circuit evaluation problem, in which we are given a boolean circuit represented as an acyclic directed graph. We assume that the graph is topologically ordered and that each node represents a function to be evaluated on the values of the incoming edges of the node. Chiang *et al.* showed that the functions of all nodes can be evaluated using $O(\text{Sort}(N))$ I/Os if $\sqrt{M/2B}\log(M/2B) \geq 2\log(2N/M)$. The assumption on M/B is due to the I/O-efficient priority queue that Chiang *et al.* relies on in their result. This assumption was removed by Arge [10] when he introduced the *buffer tree*. The buffer tree is an I/O-efficient priority queue that supports N insertions and deletions using $O(\text{Sort}(N))$ I/Os with no assumptions on the size of M.

Another priority queue implementation was described by Brodal *et al.* [37]. Their result supports B consecutive operations using at most $O\big(\log_{M/B}(N/M)\big)$ I/Os. Additionally, insertion and deletion operations perform $O(\log_2(N))$ comparisons in internal memory. This result improves upon the buffer tree by stating non-amortized worst-case guarantees and by bounding the number of comparisons performed in internal memory.

2.1.3 List Ranking

Chiang *et al.* [42] presented an I/O-efficient solution to the *list-ranking* problem. In the list-ranking problem, we are given a linked list where each node contains a pointer to its successor in the list. The objective is to compute the distance from each node to the end of the list. Chiang *et al.* presented

an algorithmic framework that solves the problem using $O(\text{Sort}(N))$ I/Os if $\sqrt{M/2B}\log(M/2B) \geq 2\log(2N/M)$. When combined with the buffer tree by Arge [10], we obtain an optimal $O(\text{Sort}(N))$ I/O solution without the assumption on M/B. Furthermore, Chiang *et al.* [42] demonstrated how the list-ranking problem can be applied to other graph-theoretic problems such as computing depth-first search numberings and finding least common ancestors in trees.

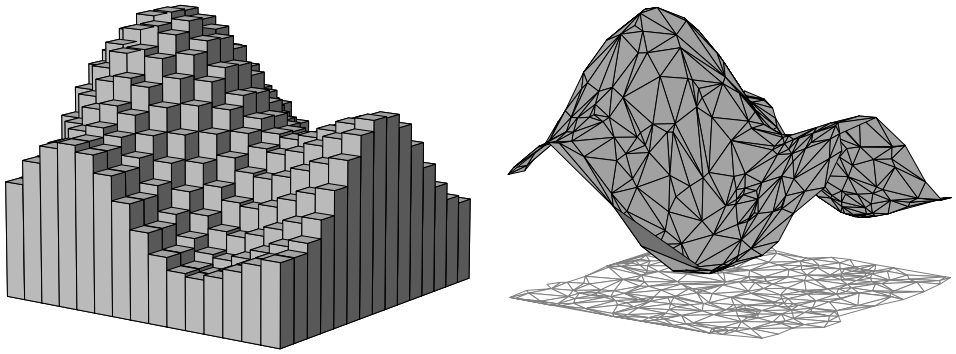
2.1.4 Multiway Separators

Given a planar graph with N vertices and a parameter r, a *multiway planar* separator divides the graph into $O(N/r)$ regions (not necessarily disjoint) of size at most r. We say that each region consists of two types of vertices: *boundary vertices* and *internal vertices*. An internal vertex is a vertex that is in only one region and is adjacent only to vertices in the same region. A boundary vertex is adjacent to vertices in more than one region and can be contained in multiple regions. Since the graph is planar, it can be shown that a multiway planar separator always exists such that each region contains $O(\sqrt{r})$ boundary vertices [53]. It follows that the total number of boundary vertices is $O\big(N/\sqrt{r}\big)$. Maheshwari *et al.* [80] presented an algorithm for I/O-efficiently computing a multiway planar separator that divides a planar separator into N/M regions using $O(\text{Sort}(N))$ I/Os. The concept of multiway separators is extremely useful and can be used to solve many fundamental planar graph problems in the I/O model, such as breadth-first search, single-source shortest paths, depth-first search, strong connectivity, and topological sorting [26, 29]. Arge *et al.* [28] furthered the study of multiway planar separators by presenting an algorithm that uses $O(\text{Sort}(N))$ I/Os and $O(N \log N)$ time in internal memory. Additionally, Arge *et al.* [28] showed that their result can be used to derive algorithms for finding single-source shortest paths, topological sorting, and finding strongly connected components using $O(\text{Sort}(N))$ I/Os and $O(N \log N)$ internal memory computation time. Maheshwari *et al.* [80] did not provide any bounds on internal memory computation time for their results.

2.2 Terrain Definitions

In order to precisely describe computational problems on terrain models, we state a more formal definition of what a terrain is and how water flows on the terrain. In this section, we state two commonly used definitions of terrain. These precise definitions are often omitted from publications due to page constraints. Thus, this section is intended to serve as a general introduction to the area of terrain analysis. The definitions in this section loosely follow those of [17, 23, 77].

The terrain models used in this thesis are *digital elevation models (DEMs)* that describe the height of each point on the surface of the terrain. A digital



(a) An example of a grid DEM.

(b) A section of a triangulated irregular network DEM.

Figure 2.3: Two digital elevation models representing the surface of a section of terrain.

elevation model is given by a *mesh* $\mathbb{M} \subseteq \mathbb{R}^2$ along with a *height function* $h : \mathbb{M} \to \mathbb{R}$ that assigns a height to each point on $\mathbb{M}$. Given $\mathbb{M}$ and h, the *terrain* $\Sigma = (\mathbb{M}, h)$ is defined as $\Sigma = \big\{(x, y, z) \in \mathbb{R}^3 \mid (x, y) \in \mathbb{M}, h(x, y) = z\big\}$. There are two types of terrain commonly used in publications; *triangulated irregular networks* and *grid* digital elevation models. A triangulated irregular network DEM (TIN DEM) represents the surface as a triangulated continuous surface such that the height of a point in each triangular face is defined by linear interpolation on the vertices of the face. A grid DEM represents the terrain as a rectangular area subdivided into square cells such that the height of each square cell is a constant function. See Figure 2.3 for illustration. We proceed by describing each type of digital elevation model in more detail.

2.2.1 Triangulated Irregular Network DEM

In the context of a triangulated irregular network (TIN), we let $\mathbb{M}$ be a triangulation of a set of vertices $\mathbb{V}$ in the plane. Often, a *Delaunay triangulation* of the vertices are used. A Delaunay triangulation is a triangulation such that no vertex of a triangle is contained in the interior of a circumcircle of any triangle in the triangulation.

In each face of $\mathbb{M}$, the height function $h : \mathbb{M} \to \mathbb{R}$ is restricted to be a linear interpolation of the height of the corners. The triangulation may contain a vertex v_∞ at infinity such that each edge (u, v_∞) is a ray from u [77]. In this case, the triangles in $\mathbb{M}$ incident to v_∞ are unbounded and we let h approach ∞ at v_∞. It follows that $\Sigma = (\mathbb{M}, h)$ is a continous surface in $\mathbb{R}^3$.

Critical Vertices

Given a terrain and a vertex $v \in \mathbb{V}$, we say a vertex u is *adjacent to* v or *a neighbor of* v if there is an edge (v, u) of $\mathbb{M}$. Throughout this section, we assume that two adjacent vertices v and u in $\mathbb{M}$ satisfy $h(v) \neq h(u)$. In latter sections, we discuss how to avoid this assumption. We say that an adjacent vertex u is an *upslope* neighbor of v if $h(u) > h(v)$. Correspondingly, u is a *downslope* neighbor of v if $h(u) < h(v)$. It follows that all adjacent vertices of a vertex v will be either downslope or upslope. If a vertex v has no downslope neighbors, then v is a *minimum*. Correspondingly, if v has no upslope neighbors, then v is a *maximum*. Mimima and maxima are also referred to as *sinks* and *peaks*, respectively. We say that a vertex v is a *saddle* if v has a sequence of four neighbors u_1, u_2, u_3, u_4 in clockwise order such that $\max\big(h(u_1), h(u_3)\big) < h(v) < \min\big(h(u_2), h(u_4)\big)$. A vertex that is either a sink, a peak, or a saddle is called a *critical vertex*.

Contours and Depressions

Given a height ℓ, the ℓ-level set of h is the set $h_{=\ell} = \big\{x \in \mathbb{M} \mid h(x) = \ell\big\}$. Correspondingly the ℓ-sublevel set of h is the set $h_{<\ell} = \big\{x \in \mathbb{M} \mid h(x) < \ell\big\}$. The connected components of $h_{=l}$ are called *contours* at height ℓ and the connected components of $h_{<l}$ are called *depressions* at height ℓ. We observe that the boundary of a depression β in $h_{<\ell}$ is formed by one or more contours from $h_{=\ell}$. Furthermore, a contour is not a simple polygonal cycle if it contains a saddle vertex.

Given a vertex $v \in \mathbb{M}$ with height ℓ, we say that a depression β_v of $h_{<\ell}$ is *delimited by* v if v lies on the boundary of β_v. We say that a depression β is *maximal* if every depression $\beta' \supset \beta$ contains strictly more sinks than β. Note that each maximal depression is delimited by a saddle. If two depressions β_1 and β_2 are delimited by a saddle v, then v is a *negative saddle* and β_1 and β_2 are *sibling depressions*. A maximal depression containing only one sink is called an *elementary depression*. See Figure 2.4 for illustration. The *volume* of a depression β in $h_{<\ell}$ is

$$\mathrm{Vol}(\beta) = \int_{\beta} \big(\ell - h(x)\big)\, dx \ .$$

Merge Tree

Suppose we sweep a horizontal plane from $-\infty$ to ∞ over a given terrain Σ. As the height ℓ of the plane changes, so do the depressions in the ℓ-sublevel set $h_{<\ell}$. The depressions change continuously, however, depressions appear and disappear only at certain critical vertices. Observe that whenever the sweep plane crosses a sink, a new elementary depression is added to $h_{<\ell}$. Whenever

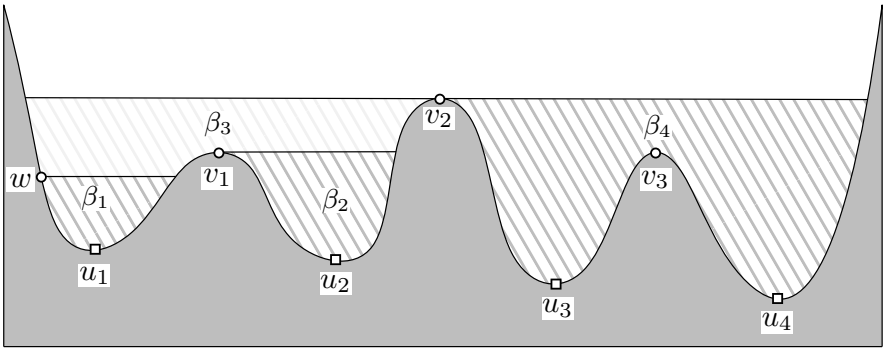


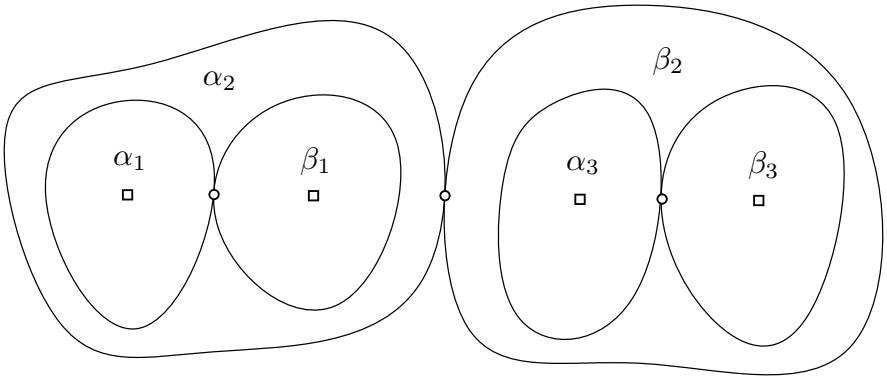
Figure 2.4: Illustration of the depressions of a terrain viewed from the side. The vertices u_1, u_2, u_3, u_4 are sinks and the vertices v_1, v_2, v_3 are negative saddles. The depression β_1 is a (non-maximal) elementary depression delimited by w and β_2 is maximal elementary depression delimited by v_1. Depressions β_3 and β_4 are maximal sibling depressions delimited by v_2.

the sweep plane crosses a peak, a hole in a depression is removed. Finally, when the sweep plane crosses a negative saddle, two or more sibling depressions are merged. The *merge tree* T of Σ is a data structure used to track these changes. The leaves of T correspond to the sinks of Σ, and the internal nodes correspond to the negative saddles of Σ. Each edge (u, v) of T corresponds to the maximal depression β_u delimited by u that contains v. See Figure 2.5 an illustration. For simplicity, we assume T is a binary tree. That is, we assume each negative saddle delimits at most two depressions. Such negative saddles are called *simple*. Non-simple saddles can be unfolded into simple saddles as described by Edelsbrunner *et al.* [49].

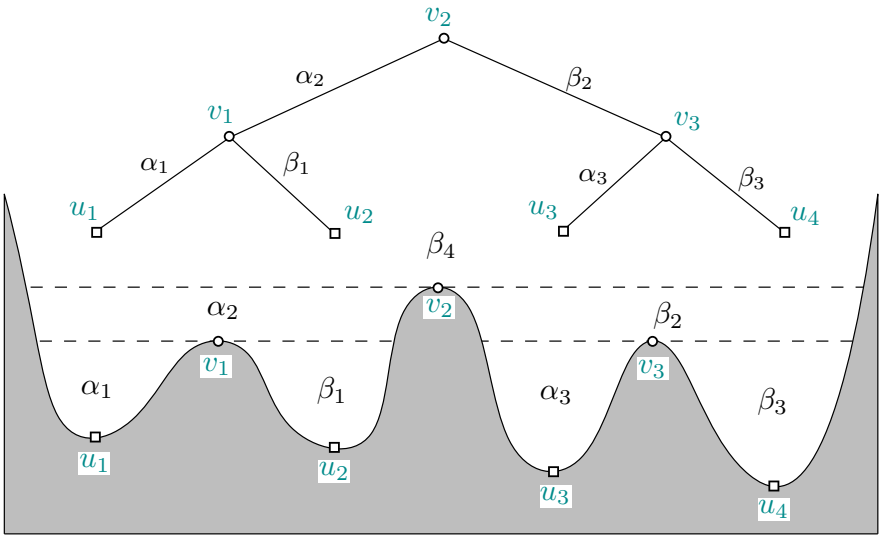
The *extended merge tree* is obtained by mapping each vertex $v \in \mathbb{V}$ to the edge of T corresponding to the smallest maximal depression containing v. For each edge of T we store the list of mapped vertices in non-decreasing order of height. The extended merge tree is also referred to as the *augmented merge tree*.

2.2.2 Grid DEM

Terrains can also be represented using a *grid digital elevation model (grid DEM)*, also known as a *raster DEM*. A grid DEM is represented by a rectangular area subdivided into N square cells of equal size. This representation has the advantage of being very simple and often leads to elegant and efficient algorithms. Furthermore, the model can be stored and transferred quite efficiently due to the grid structure. However, triangulated irregular networks might be preferred for terrains when collected terrain data is highly non-uniform and non-grid-like. We proceed by stating relevant definitions on grid terrains. These definitions are analog to those of the TIN digital elevation model and many follow from



(a) Bird's-eye view showing how the depressions of a terrain are nested.



(b) The merge tree of a terrain. The leaves of the merge tree correspond to sinks u_1, u_2, u_3, and u_4. The internal vertices of the merge tree correspond to negative saddle vertices v_1, v_2, and v_3. Observe how each edge in the merge tree corresponds to a maximal depression.

Figure 2.5: The depressions and merge tree of a terrain.

the TIN definitions.

Given a grid $\mathbb{M}$, we restrict the height function $h : \mathbb{M} \to \mathbb{R}$ to be constant in the face of each cell. It follows that the terrain formed by a grid DEM, denoted $\Sigma = (\mathbb{M}, h)$, does not form a continuous surface unlike that of a TIN DEM. We say that two cells are *adjacent* or *neighbors* if the boundaries of the two cells have non-empty intersection. That is, cells that share a corner point or an edge are adjacent to each other. Let $\mathbb{V}$ denote the centers of the cells of $\mathbb{M}$. We define the *adjacency graph* G_Σ to be the graph with vertices $\mathbb{V}$, where two vertices are connected if the corresponding cells are adjacent.

Critical Cells

We say that a cell u is a *downslope neighbor* of v if u and v are adjacent and $h(u) < h(v)$. Correspondingly, u is an *upslope neighbor* of v if $h(u) > h(v)$. As we did for TIN DEMs, we assume that no two adjacent cells have the same height. A cell v is a *peak* or *maximum* if it has no upslope neighbors. Correspondingly, v is a *sink* or *minimum* if it has no downslope neighbors.

Depressions

Depressions on grid terrain have a similar definition to those of TIN terrains, however, one needs to be careful due to the non-continuity of the surface. The ℓ-sublevel set of h is the set $h_{<\ell} = \big\{v \in \mathbb{V} \mid h(v) < \ell\big\}$. The depressions at height ℓ are defined to be the connected components of $h_{<\ell}$. A cell v delimites a depression β at height $h(v)$ if v is adjacent to a cell in β. A *negative saddle* is a cell that is adjacent to two or more depressions. Letting A denote the area of each cell, the *volume* of a depression β at height ℓ is defined as

$$\mathrm{Vol}(\beta) = A \cdot \sum_{v \in \beta} \big(\ell - h(v)\big) \ .$$

Merge Tree

The *merge tree* T of a grid DEM follows the definition of the merge tree for a TIN DEM. The leaves of T correspond to the sink cells of Σ, and the internal nodes correspond to the negative saddle cells of Σ. Each edge (u, v) of T corresponds to the maximal depression β_u delimited by u that contains v. We assume that T is a binary tree. That is, each negative saddle is a *simple saddle* that delimits at most two depressions.

Converting from grid to triangulated irregular network

A natural question that arises is whether one can convert between grid DEMs and TIN DEMs while preserving the topological properties of the terrain. Observe that the adjacency graph G_Σ of a grid terrain Σ may not be planar

due to the inclusion of all diagonal edges. Hence, we remove edges from the adjacency graph to convert a grid DEM to a TIN DEM. Given the planar embedding of G_Σ, let $e_1 = (u, v)$ and $e_2 = (w, t)$ be two edges that cross. The adjacency graph is mapped to a planar graph by removing either e_1 or e_2 for all such pairs. Furthermore, note that this forms a Delaunay triangulation. Thus, this creates a mapping from grid DEMs to TIN DEMs. We let the *mid point height* $\bar{h}(e)$ of an edge $e = (u, v)$ denote the value $\frac{h(u)+h(v)}{2}$. Let the *lower edge triangulation* denote the embedding where we for each crossing pair of diagonals e_1 and e_2 remove the diagonal with the largest midpoint height. If two diagonals have the same midpoint height, we arbitrarily remove either e_1 or e_2. Arge *et al.* [18] showed that the TIN DEM formed by the lower edge triangulation has the same sinks, negative saddles, and merge tree as the corresponding grid DEM.

2.3 Flow Model Definitions

Having described what a terrain is, we proceed by describing how the flow of water and flooding can be modeled on terrain. When the amount of water on the terrain exceeds what can be absorbed by the soil and sewer systems, water starts accumulating in depressions of the terrain. The fill rate of a terrain depends on several factors, such as the volume of the depression, the rate at which rain falls, and the area from which rain flows into the depression. Whenever a depression becomes full, it will start spilling into the sibling depression, thus, increasing the fill rate of the sibling depression. We refer to such rain events as *flash flood events*. In this section, we formally state how the flow of water is modeled on terrain. For simplicity, we only describe the model for TIN DEMs, however, the model can be adapted to grid DEMs using the definitions stated in Section 2.2.2.

The problem of modeling the flow of water on a terrain has been studied extensively in the GIS community. When modeling water flow, it is important to use models that are simple enough to be computationally efficient but still maintain a sufficient level of detail. Liu *et al.* [75] described the *surface flow model* in which water flows on the surface of the terrain with infinite velocity without being absorbed by the terrain. At each point x on the surface of Σ, water flows according to a *flow direction* of x which is selected as the direction of steepest descent. In the case of flat areas or ties, one needs to carefully handle ties ensuring that the flow directions do not form cycles.

A simpler model is the *edge flow* model in which water is restricted to flow on only on the vertices and edges of the terrain [22, 78]. For very large terrains, this assumption provides a good approximation since the input triangles tend to be small compared to the total area. We proceed by stating definitions and results for the edge flow model.

2.3.1 Flow Graph

In the edge flow model, we assume water flows along the edges of the terrain. To determine the direction in which water flows, we introduce the *flow graph* $\mathsf{F_G}$ which is a directed acyclic graph. The vertices of the $\mathsf{F_G}$ are in one-to-one correspondence to the vertices of the terrain. Each vertex of $\mathsf{F_G}$ has outgoing edges corresponding to the direction in which water flows from the vertex. Water collects in sinks that all have no outgoing edges. Two commonly used variants of this model are the *single-flow direction (SFD) model* and the *multiflow direction (MFD) model*. In the single-flow direction model, each vertex v has at most one outgoing edge to a neighbor u of v. Typically the outgoing edge is selected to be the edge with the *lowest direction*, where u is selected to minimize $h(u)$, or the *steepest direction*, where u is selected to minimize $\big(h(v) - h(u)\big)/|v - u|$.

In the multiflow direction model, each vertex contains edges to all downslope neighbors. For each vertex v we define $\lambda(v, u)$ to be the proportion of the water arriving at v that flows along the edge (v, u) to u. Note that $\sum_u \lambda(v, u) = 1$ for a vertex v. The value of $\lambda(v, u)$ is typically selected based on the heights of v and u [77].

2.3.2 Rain Distribution

When modeling rainfall, we let $\mathcal{R} : \mathbb{V} \to \mathbb{R}_{\geq 0}$ denote a *rain distribution* which is a distribution describing the rate at which rain falls on each vertex of the terrain. That is, for a vertex v, $\mathcal{R}(v)$ units of rain fall on v in one time unit. Since $\mathcal{R}$ is a distribution, it follows $\sum_v \mathcal{R}(v) = 1$ and, for all vertices v, $\mathcal{R}(v) \geq 0$. Let $|\mathcal{R}|$ denote the number of vertices with non-zero rainfall in $\mathcal{R}$. For many of the algorithms presented in the next section, $\mathcal{R}$ is simply the uniform distribution. However, as we will describe below, some results achieve a significant speed-up by preprocessing the terrain to answer queries for varying rain distributions. Furthermore, for queries where $|\mathcal{R}| \ll N$, some algorithms achieve a significant speed-up by visiting only vertices with non-zero rainfall or by preprocessing the terrain to efficiently answer queries for varying rain distributions.

2.3.3 Depression Filling Model

Liu *et al.* [75] presented a *depression filling model* that formalizes the filling and spilling of depressions. In this model, rains falls on the vertices of the terrain Σ according to a rain distribution $\mathcal{R}$, flows on Σ according to the flow directions, accumulating in the maximal depressions of Σ. When the amount of rain accumulated in a maximal depression β is equal to $\mathrm{Vol}(\beta)$, β becomes full and water will start spilling from β. Let v be the vertex delimiting β and let β be the sibling depression of β'. Assume that β' is not already full. In this case, the water that falls into β spills over the saddle v into β', thus, increasing the rate at which β' fills. We refer to such an event as a *spill event*.

In the single-flow direction model, we let the *secondary flow direction* of the negative saddle vertex v be the flow direction of v into β' after β has filled. In the multiflow direction model, we update λ by setting the $\lambda(v,u) = 0$ for all vertices u in β depression. We then reweigh $\lambda(v,u)$ such that $\sum_u \lambda(v,u) = 1$. This process results in a sequence of spill events, where each event corresponds to a depression becoming full and spilling into the neighboring depression.

2.4 Terrain Analysis Algorithms

In this section, we provide a survey of algorithmic results for terrain analysis and the modeling of water flow on a terrain. We focus on the research area of I/O-efficiently modeling flooding from rainfall and the problems directly related to this. However, there is also prior work focusing on modelling floods from sea-level rise [18, 23] and river-rise [8].

2.4.1 TIN DEM Construction

When terrain data is collected, it is typically represented as a *point cloud* such that each point denotes a point on the surface of the terrain. Given a point cloud, we can use it construct the grid DEM and TIN DEM which were described in Section 2.2. A TIN DEM can be constructed by projecting the point cloud onto the plane, computing a Delaunay triangulation of the projected points, and lifting the triangulation back up to $\mathbb{R}^3$. The problem of computing Delaunay triangulations has been studied extensively in computational geometry, and several algorithms have been presented. The algorithms are typically based on the sweep-line or divide-and-conquer paradigms [30]. In the RAM model, Guibas *et al.* [59] presented a randomized incremental algorithm with expected $O(N \log N)$ running time, where N is the number of vertices in the triangulation. An extension of the Delaunay triangulation is the so-called *constrained Delaunay triangulation* in which the triangulation is constrained to contain a given set of edges. This problem is particularly interesting in the context of TIN DEM construction since the constraint edges can be used to represent features on the terrain such as roads and rivers. In the RAM model, constrained Delaunay triangulations can be computed in $O(N \log N)$ time [41].

Results for computing Delaunay triangulations have also been presented in the I/O model. Goodrich *et al.* [58] showed that the Delaunay triangulation can be computed using expected $O(\text{Sort}(N))$ I/Os. Agarwal *et al.* [2] presented an expected $O(\text{Sort}(N))$ I/Os algorithm for computing constrained Delaunay triangulations.

2.4.2 Grid DEM Construction

There are two approaches commonly used to constructing grid DEMs from point clouds. For the first approach, a TIN DEM is computed based on the

point cloud. The TIN DEM is then converted to grid DEM by interpolating the height of each grid cell from the corner points of the triangle in which the center point of the grid cell is contained. Isenburg *et al.* [66] proposed an algorithm that computes the TIN and converts it to a grid DEM by sweeping over the point cloud. The second method for computing grid DEMs is to compute the height of each grid cell in the grid DEM directly from the point cloud by interpolating the heights of the nearest points. The heights are typically interpolated by inverse distance or using a piecewise polynomial function [1]. Agarwal *et al.* [1] presented an I/O-efficient algorithm for computing such interpolations by recursively computing a quadtree on the input points.

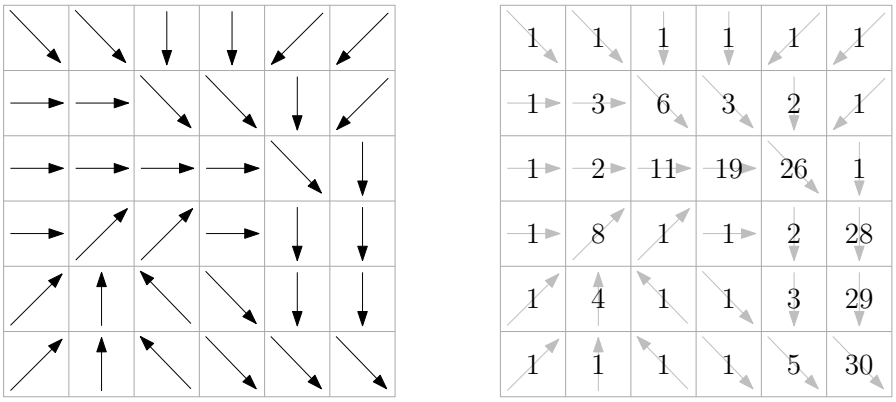
2.4.3 Merge Tree

Carr *et al.* [39] presented an $O(N \log N)$ time algorithm for computing the merge tree of a TIN DEM in the RAM model. Furthermore, using $O(N)$ preprocessing time, the merge tree can be augmented such that for a point $x \in \mathbb{M}$, the volume of the depression β_x delimited by x can be computed using $O(\log N)$ time [39]. Additionally, they showed that each vertex v can be augmented with a pointer to the smallest maximal depression containing v in $O(N \log N)$ time. In the I/O model, Agarwal *et al.* [3] presented an $O(\text{Sort}(N))$ algorithm for computing the merge tree of a TIN DEM. Arge *et al.* [21] extended the algorithm by Carr *et al.* [39] to obtain an algorithm for computing the merge tree and the volumes of all maximal depressions using $O(\text{Sort}(N))$ I/Os. Furthermore, Arge *et al.* [22] extended this to compute the volume of β_v and the smallest maximal depression containing v for all vertices on the terrain. We remark that the above algorithms can be adapted to grid DEMs using the observations in Section 2.2.2.

2.4.4 Flow Routing

In Section 2.3.1, we described how to assign flow directions based on the assumption that no two adjacent vertices and cells have equal height in the input DEM. However, this assumption might not hold on real-world data and, thus, we need to ensure flow directions are chosen such that the flow graph is acyclic. This is addressed in the *flow routing* problem in which we assign flow directions to vertices and cells in flat areas of the terrain. We proceed by describing the problem and algorithms for grid DEMs, however, the algorithms described can be adapted to TIN DEMs [44].

In the flow routing problem, we define a *flat area* to be two or more adjacent cells with equal height. We distinguish between two types of flat areas: *plateaux* and *reservoirs*. A plateau is a flat area for which there is an adjacent cell with a lower height. A reservoir is a flat area for which all adjacent cells have greater height. We remark that reservoirs are also referred to as sinks in literature, however, this is not to be confused with sink vertices. The objective of the flow



(a) Flow directions of all cells (b) Flow accumulation of all cells

Figure 2.6: Computing the flow accumulation of a grid DEM.

routing problem is to compute an acyclic flow graph F_G such that each vertex in a plateau has a path to a vertex with lower height and that each reservoir has exactly one vertex with no outgoing edges. Arge *et al.* [13] presented an algorithm for a variant of the flow routing on grid DEMs where flow directions are computed for a terrain with all reservoirs filled. That is, the reservoirs are raised to the height of the lowest adjacent cell, thus, turning reservoirs into plateaux. Arge *et al.* [13] described how this problem can be solved using a breadth-first-search style algorithm that uses $O(\mathrm{Sort}(N))$ I/Os. Furthermore, their algorithm also computes a *rank* $\mathrm{rank}(v)$ for each cell v such that a cell u is before v in the topological ordering of F_G if and only if $h(u) > h(v)$ or $\mathrm{rank}(u) < \mathrm{rank}(v)$. The algorithm can be adapted to compute flow directions and ranks without first filling the terrain.

2.4.5 Flow Accumulation

The motivation for Arge *et al.* [13] to study flow routing was to further extend their work on the *flow accumulation* problem [86] on grid DEMs under the single-flow direction model. In the flow accumulation problem, we are given a grid DEM, the flow direction for each cell on the terrain, and a rain distribution $\mathcal{R}$. For each cell v, we initially assign $\mathcal{R}(v)$ units of water to v. Water is distributed by pushing it along the flow directions of each cell until all water is at the sinks of the terrain. The *flow accumulation* of a cell v is the accumulated amount of water that is pushed through v during this process. Cells with high flow accumulation correspond to areas with a large flow of water and can be used to approximately identify river networks on the terrain. See Figure 2.6 for an example.

An I/O-efficient algorithm for the computation of flow accumulation was presented by Arge *et al.* [25]. We proceed by describing the algorithm in more detail to demonstrate the application of the *time-forward processing* technique

to terrain processing. First, observe that the flow accumulation $f(v)$ of a cell v is defined recursively as follows:

$$f(v) = \mathcal{R}(v) + \sum_{(u,v)\in \mathsf{F_G}} f(u) \; . \tag{2.1}$$

The algorithm exploits this observation by visiting the cells of the terrain in topological order and *forwarding* values computed at each node using a priority queue. In order to do so, we compute flow directions using the flow routing algorithm by Arge *et al.* [13] and sort the cells lexicographically by $(h(v), \text{rank}(v))$. In other words, we sort the cells of the terrain according to their topological ordering. We scan the sorted list of cells and visit each cell while maintaining an I/O-efficient priority queue containing elements such that the following invariants are satisfied:

1. If there is a cell u that has been visited and a cell v that has not been visited where $(u, v) \in \mathsf{F_G}$ then $(v, f(u))$ is contained in the priority queue.
2. If there is a cell v that has not been visited, then $\big(v, \mathcal{R}(v)\big)$ is contained in the priority queue.

Furthermore, the elements $(v, \cdot)$ in the queue are keyed on $(h(v), \text{rank}(v))$. The algorithm proceeds as follows: Initially, for each cell v, add $\big(v, \mathcal{R}(v)\big)$ to the priority queue. Scan the sorted list of cells and, for each cell v, delete elements $(v, \cdot)$ from the priority queue and use (2.1) to compute the flow accumulation of v. If there is an edge $(v, w) \in \mathsf{F_G}$, insert $(w, f(v))$ into the priority queue. We observe that the invariants are maintained at each step. It follows that the algorithm computes the flow accumulation of each cell. Using an I/O-efficient priority queue [10] and sorting algorithm [6], we observe that the algorithm uses $O(\text{Sort}(N))$ I/Os, where N is the number of cells in the terrain. Note that this algorithm can be adapted to TIN DEMs. Furthermore, the concept of flow accumulation can be adapted to the multiflow direction model by multiplying the flow along each outgoing edge (v, w) by $\lambda(v, w)$.

Haverkort *et al.* [62] presented an algorithm for computing flow accumulation on grid terrains using $O(\text{Scan}(N))$ I/Os under the assumption that the grid is stored in row-by-row order and $M \geq cB^2$, where $c > 0$ is a constant. Instead of relying on an I/O-efficient priority queue, their algorithms partition the grid into memory-sized regions and uses the partitioning to compute the flow accumulation of all cells.

2.4.6 Watershed

Additionally, Arge *et al.* [13] considered the *watershed* problem on a grid based terrain in the single-flow direction model. In the single-flow direction model, we observe that water in a cell v flows to exactly one sink on the terrain. Hence, for each sink u on the terrain, we define the *watershed* of u to be the cells on

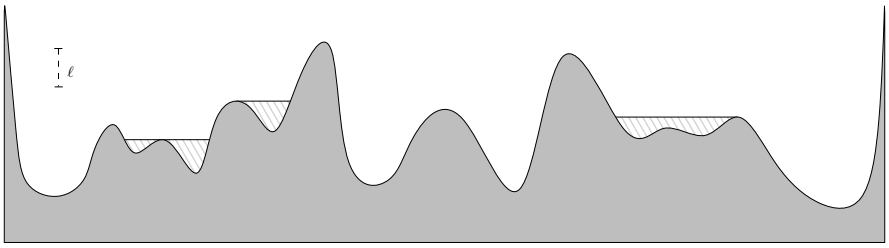


Figure 2.7: A terrain where maximal depressions with height less than ℓ are filled.

the terrain for which there is a path in F_G to u. It follows that a cell belongs to exactly one watershed. Arge *et al.* [13] described an $O(\text{Sort}(N))$ time-forward processing based algorithm for computing all watersheds of a grid terrain.

2.4.7 Partial Flooding

The computation of flow accumulation and watersheds provides only a rough model of terrain flooding. In practice, water does not disappear at sinks but accumulates in depressions and starts spilling whenever depressions are filled. In order to more accurately model flow, one can preprocess the terrain by *partial flooding*. Let the *height* of a depression be the difference in elevation between the sink of the depression and the negative saddle delimiting the depression. When partially flooding a terrain, maximal depressions smaller than a given geometric measure, such as height or volume, are filled. That is, all cells within a depression β_v are raised to the height of the saddle vertex v delimiting β_v. Thus, when computing flow accumulation and watersheds on a partially flooded terrain, water will not be trapped by smaller depressions which results in a more realistic model for extracting river networks. Agarwal *et al.* [3] presented an $O(\text{Sort}(N))$ algorithm for partially flooding a TIN terrain such that depression with height less than ℓ are removed. See Figure 2.7 for illustration. Arge *et al.* [21] further extended this to remove depressions based on the volumes and areas of depressions.

Partially filling a terrain is particularly useful for real-world data which can contain *spurious depressions* that are created as a result of noise in the input data. By partially flooding the terrain, we not only remove spurious depressions but also significantly reduce the size of the merge tree. For example, when spurious depressions smaller than $1\,\text{m}^3$ are filled in the Danish Elevation Model [50], the total number of sinks is approximately 30 million. Whereas the unfilled terrain has 10.5 billion sinks. As shown in the next section, this can lead to an improvement in performance for several terrain flooding

algorithms [17, 22, 77].

2.4.8 Terrain Flooding

The motivation for partially flooding the terrain is to avoid water collecting and being trapped in spurious depressions when computing flow accumulation and watersheds. However, this solution fails to model how water accumulates in large depressions. Whenever a large depression fills, the rain which falls in that depression will start spilling into the sibling depression, thus, increasing the fill rate of the sibling. It follows that the fill rate of a depression depends not only on the rain falling directly in the depression but also on the water spilling into the depression. In this section, we describe various computational problems where the filling of depressions is modelled using the depression filling model by Liu *et al.* [75] (Section 2.3.3).

We first consider the *terrain flood-time* problem. Given a rain distribution $\mathcal{R}$, we say that it rains $\mathcal{R}(v)$ units of water on vertex v per unit of time. When solving the terrain flood-time problem, we compute the flood-time of all vertices v on the terrain. That is, we compute the time at which v is submerged by water. Liu *et al.* [75] presented an internal memory algorithm for solving the terrain flood-time problem on TIN DEMs in the SFD model in time $O(N \log N)$. They state their result given a uniform rain distribution, however, the algorithm can be adapted to arbritary given rain distributions.

Rav *et al.* [89] presented an internal memory algorithm for constructing a linear size data structure that can answer *vertex flood-time* queries on TIN DEMs in the SFD model. Given a rain distribution $\mathcal{R}$ and a vertex v, determine at which time v is flooded. Their data structure can answer such queries in time $O(|\mathcal{R}| + Q \log N)$, where Q is the number of *tributaries* of v. A tributary of v is a depression that spills into a depression containing v when full. We note that Q is at most the height of the merge tree.

Arge *et al.* [22] presented an I/O-efficient algorithm for the terrain flood-time problem on TIN DEMs in the SFD model by extending the algorithm by Liu *et al.* [75]. Their algorithm solves the flood-time problem using $O\big(\operatorname{Sort}(X) \log(X/M) + \operatorname{Sort}(N)\big)$ I/Os, where X is the number of sinks in the terrain and M is the size of the internal memory. Their algorithm is conceptually very complex and is not tested in practice. However, when $X = O(M)$, the algorithm can be simplified greatly and the number of I/Os becomes $O(\operatorname{Sort}(N))$.

Arge *et al.* [17] presented I/O-efficient algorithms for the *terrain flood-event* problem for grid DEMs in the SFD model; Given a rain distribution $\mathcal{R}$, determine which vertices of the terrain are flooded at a fixed time t. Note that solving the terrain flood-time problem also solves the terrain flood-event problem. In other words, the flood-time problem is conceptually harder. Arge *et al.* [17] presented an I/O-efficient algorithm that uses $O(\operatorname{Sort}(N) + \operatorname{Scan}(H \cdot X)$ I/Os, where X is the number of sinks in the terrain and H is the height of

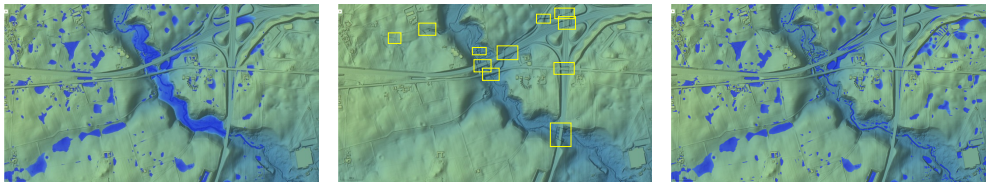
the merge tree. Their result can be adapted to TIN DEMs without increasing the number of I/Os performed. Additionally, they describe how to modify their algorithm to use $O(\text{Sort}(N))$ I/Os provided $H = O(M)$. Furthermore, provided a constant number of rows of the grid DEM fit in memory, they show how to solve the terrain flood-event problem using $O(\text{Scan}(N) + \text{Sort}(X))$ I/Os after using $O(\text{Sort}(N))$ I/Os of preprocessing. When $X = O(M)$, the number of I/Os can be further reduced to $O(\text{Scan}(N))$. That is, most of the work of the algorithm can be handled in a preprocessing step independent of a specific rain distribution.

The solutions for terrain flooding stated above can not trivially be adapted to the MFD model. The various algorithms stated for the SFD model rely on the property that the water spilling from a depression follows a path into exactly one other depression. However, in the MFD model this property does not hold since the water spilling from a saddle spreads over several paths and may spill into more than one depression. Recently, Lowe *et al.* [77] presented internal memory algorithms for TIN DEMs in the MFD model. First, they presented an $O(N \log N)$ algorithm for solving the terrain flood-event problem. Secondly, they presented an $O(N \log N + NX)$-time internal memory algorithm for preprocessing the terrain into a data structure for answering *vertex flood-event* queries: given a rain distribution $\mathcal{R}$, determine whether a vertex v is flooded at a given time t. The data structure can answer queries in $O(|\mathcal{R}|K + K^2)$ time, where K is the number of maximal depressions containing v. Finally, they presented an algorithm for answering vertex flood time queries. Assuming two $K \times K$ matrices can be multiplied in time $O(K^\omega)$ for some constant $\omega \geq 2$, they show how vertex flood time queries can be answered in $O(NK + K^\omega)$ time, where K is the number of maximal depressions containing v.

2.4.9 Hydrological Correction Identification

In order to realistically model water flow on terrain, a series of modifications need to be made on the terrain. We previously discussed how errors in the input data can result in spurious sinks that trap water when computing flow accumulation and watersheds. Another common issue with input data is that bridges and culverts under roads are not realistically modeled by grid and TIN DEMs. Since the data is recorded from a bird's-eye view, bridges and culverts impede the flow of water in the digital elevation model even though they are not true hydrological barriers. Therefore, we consider the problem of identifying *hydrological corrections*, which we loosely define to be a set of modification of the terrain that ensures that the flow of water is not impeded by bridges and culverts on the terrain. The effect of hydrological corrections is illustrated in Figure 2.8. A hydrological correction can be represented by a set of cells that are to be corrected by lowering their elevation to the minimum elevation of the cells adjacent to cells in the set.

Traditionally, hydrological corrections are identified by hand. An example



(a) Flooded terrain without hydrological corrections.

(b) Hydrological corrections identified.

(c) Flooded terrain with hydrological corrections.

Figure 2.8: Illustrating the effect of hydrological corrections on terrain flooding [15]. Water is impeded by bridges when hydrological corrections are not included.

of this is the list of hydrological corrections [57] created by the Danish Agency of Supply and Efficiency. This data set is a detailed and publicly available list of hydrological corrections for the Danish elevation model [50], which contains bridges and culverts. The list is maintained by the municipalities of Denmark and is updated mostly manually using local knowledge and human input. Typically, corrections are identified by manually inspecting aerial photos near intersections between road and river networks. This might result in errors in the data set when the road and river data are not aligned with the elevation model or when a hydrological correction is not near a road or river. Furthermore, the process of creating and maintaining such a data set is quite laborious and requires manual input when the elevation model is updated.

More recently, Carlson *et al.* [38] approached the problem algorithmically using feature engineering and machine learning to automatically detect bridges in a grid elevation model. In order to detect corrections, they manually engineered various features based on the elevation model of a terrain and applied the AdaBoost [55] algorithm to train a classifier on the features. The feature engineering used by Carlson *et al.* [38] is based on various edge detectors computed on the elevation model. Furthermore, Carlson *et al.* [38] computed a partial filling of the terrain to detect smaller depression which might be caused by bridges and culverts in the terrain. This results in a total of 510 features for each cell in the terrain which they use to train an AdaBoost classifier [55] to predict whether a cell is part of a bridge or not. After having trained their classifier on manually labeled data, they describe how to correct bridges by grouping cells identified as bridges and lowering the elevation of cells in each group.

2.5 External Memory Algorithms in Practice

We have so far discussed the design and analysis of I/O-efficient in only a theoretical context. However, since terrain analysis is heavily motivated by

practical applications, it is valuable to discuss the issues encountered when implementing and evaluating terrain algorithms in practice. Typically when implementing internal memory algorithms, the many low-level details have been hidden by the operating system and low-level libraries. Instead of directly accessing the RAM of the computer, the programmer accesses a virtual memory where addresses are mapped to physical memory by the operating system. This way, the operating system can transparently swap elements in and out of memory, and the programmer can use the virtual memory as if it were of infinite size. Furthermore, the different cache layers are transparently managed by the operating system. Whenever an element is accessed, the element will be read from cache and not RAM directly. If an element is not loaded into the cache, the operating system will transparently load the element and subsequent elements into the cache. This helps the programmer in reaping the benefits of the high-speed cache without having to explicitly manage each layer of cache. However, when implementing I/O-efficient algorithms, the internal memory size and block size are often parameters in the algorithms themselves and need to be explicitly provided. Thus, in order to write the implementation, the programmer needs to be aware of these parameters.

One approach to hiding these parameters is the *cache-oblivious model*. In this model, the algorithms are not provided the block size and size of internal memory. This has the benefit that optimal cache-oblivious algorithms are asymptotically efficient for all sizes of cache. That is, a cache-oblivious algorithm that minimizes the number of I/Os will do so for all levels of the memory hierarchy and not just the internal memory. However, cache-oblivious algorithms tend to be slower in practice despite being asymptotically optimal. For example, Brodal *et al.* [36] compared an implementation of the cache-oblivious Lazy Funnelsort to the I/O-efficient multiway merge sort implemented in the TPIE software library [85]. In their experiments, they show that the I/O-efficient multiway merge sort clearly outperforms the cache-oblivious Lazy Funnelsort implementation. The authors suggest that this might be due to optimizations implemented in the TPIE library that are difficult to transfer to the cache-oblivious setting.

To simplify the implementation of I/O-efficient algorithms, the concept of *pipelining* can be used. Pipelining is the concept of composing an I/O-efficient algorithm as an acyclic directed graph where data elements are pushed through the graph. We refer to a node of the graph as a *component*. For example, the time-forward based flow accumulation algorithm (Section 2.4.5) consists of a component that sorts the data followed by a component that streams through the sorted elements while maintaining a priority queue. There are currently two major software libraries for the implementation of I/O-efficient algorithms in this manner: *the Standard Template Library for Extra Large Data Sets (STXXL)* [46] and *the Templated Portable I/O Environment (TPIE)* [19]. In this section, we will briefly discuss the features of the libraries. This is followed by a more detailed discussion in Chapter 3. When implementing

pipelined algorithms, one needs to be careful not to introduce large overhead when streaming data from one component to the next. A naive implementation would let each component read its input data from disk and write its output back to disk. However, by letting components directly pass the data elements to the next component through a method call, we avoid this unnecessary disk operation. Both TPIE and STTXL provide robust and efficient frameworks for the implementation of such components. Furthermore, both frameworks enable modularity and reusability of components without introducing additional run-time overhead.

While most programming languages include robust and comprehensive standard libraries for internal memory algorithms, implementations of I/O-efficient algorithms are typically not included in standard libraries. Besides providing the framework for implementing modular pipeline components, both TPIE and STXXL provide well-engineered and robust implementations of fundamental I/O-efficient algorithms and data structures such as sorting and priority queues. Additionally, both TPIE and STXXL provide an interface for handling *file streams* which support the reading and writing of $O(B)$ elements using $O(1)$ I/Os. By providing such an interface, the programmer avoids manually keeping track of block boundaries and writes. Furthermore, both libraries also support compression of blocks to disk as well as asynchronous file operations that enable overlapping between I/O and computation.

Danner *et al.* [44] presented the *TerraSTREAM* project that computes flow accumulation and watersheds on a given point cloud. The project consists of four main stages: construction of a DEM, partial flooding of the DEM, computing flow accumulation, and construction of a *watershed hierarchy*, which represents how the watersheds of the DEM are nested. The stages are based on the I/O-efficient algorithms described in the previous sections, however, the project represents TIN and grid DEMs as a unified graph. This approach eases the implementation of subsequent stages since only one implementation needs to be maintained. However, as demonstrated by Haverkort *et al.* [62], having a unified representation is not always ideal since the structure of grid DEMs can be used to speed up computation for problems such as flow accumulation.

2.6 Our Contributions

In this thesis, we present several algorithmic results for the computation of flood risk on a terrain. Furthermore, we evaluate the results experimentally and present a software framework that enables modular and maintainable implementation of I/O-efficient algorithms.

First, in Chapter 3, we present an extension to the TPIE software library that provides functionality for the implementation of pipelined I/O-efficient algorithms. The extension consists of a framework that promotes and simplifies the implementation of pipelined algorithms while minimizing the I/O

overhead of the implementation. Although pipelining is also provided by libraries such as STXXL [46], the underlying philosophy of the TPIE pipelining framework is different since TPIE aims to hide the characteristics of the underlying hardware and tedious details of the implementation. Furthermore, TPIE simplifies memory management and progress tracking by automatically managing application-wide memory limits and progress. Although hand-optimizing such details may provide a slightly larger I/O-throughput by tailoring the implementation to specific hardware, these abstractions greatly simplify implementation and improve portability and re-useable of pipelining components. The TPIE pipelining library is used heavily in both scientific and commercial applications [12, 27].

In Chapter 4, we revisit the problem of I/O-efficiently computing multiway separators for planar graphs. We present a simple sampling-based algorithm that divides a planar graph into regions when given the *Koebe-embedding* of the graph. A Koebe-embedding of a planar graph is a set of disks in the plane with disjoint interiors such that each disk corresponds to a vertex in the graph, and two disks are adjacent if and only if the corresponding disks are adjacent. Additionally, we provide guarantees on the number of boundary vertices of the division under certain assumptions on the size of the internal memory. There are currently no known algorithms for computing Koebe-embeddings I/O-efficiently. Therefore, we describe how to generalize our result to Delaunay triangulations. This generalization may result in a high number of boundary vertices in the worst-case. However, we evaluate our algorithm on the Danish elevation model and show that the number of boundary vertices remains small in practice. Furthermore, we adapt the grid-based flow accumulation algorithm by Haverkort *et al.* [62] to a TIN DEM using multiway separators and show that an implementation of the algorithm performs well in practice when compared to the time-forward based algorithm by Arge *et al.* [25].

In Chapter 5, we study a number of problems related to modeling the flow of water on a TIN DEM in the multiflow direction model. Given a rain distribution $\mathcal{R}$ and a TIN DEM Σ, compute how much water is flowing over the vertices of the terrain as a function of time. This problem differs from the flow accumulation problem since we model how water accumulates in depressions. That is, the amount of water flowing over a vertex changes over time as depressions spill. Additionally, our results can be adapted to solve the terrain flood-time and terrain flood-event problems. First, we study the *terrain flow-query* problem in the multiflow direction model, where we a given a rain distribution $\mathcal{R}$ and compute the flow rate over time for all vertices of Σ. We provide efficient algorithms in both the RAM model and the I/O model. Furthermore, when adapted to solve the terrain flood-time problem, our algorithms achieve a better worst-case bound than previous best known results [77]. Additionally, our results are the first I/O-efficient results presented in the multiflow direction model. We also provide internal memory algorithms for the *vertex-flow query* problem in the single-flow direction model. Given a

rain distribution $\mathcal{R}$, a TIN Σ, and a query vertex v, compute how much water flows over v as a function of time. Finally, in reality, the flow of water is not restricted to edges but form a 2D channel of rivers on the terrain. That is, the cross-section of a channel of water is not a point but a polygon. Given a path P in Σ and the rate at which water flows along edges of P, we present a model for determining the geometry of a 2D channel from P using the empirical Manning's equation [81]. Furthermore, we present an efficient memory for computing the channel in internal memory.

Finally, in Chapter 6, we revisit the problem of automatically identifying hydrological corrections for a terrain. The Danish Agency for Supply and Efficiency provides a list of corrections that accompanies the Danish elevation model [51]. However, this list is produced semi-manually using a slow and expensive process. Furthermore, many corrections are missing from the list, and the included corrections are of varying quality. We propose a machine learning-based approach to identifying hydrological corrections on a grid DEM. In order to identify corrections, we train a convolutional neural network using the elevation model and the manually labeled list of corrections produced by the Danish Agency for Supply and Efficiency [51]. Our trained model detects most of the corrections in the manually labeled list and quite a few corrections not included in the original list. Furthermore, we describe how to output the geometry of each correction such that identified corrections can be used to modify the terrain for realistic flow modeling.

Part II

Publications

Chapter 3

External Memory Pipelining Made Easy With TPIE

Abstract

When handling large datasets that exceed the capacity of the main memory, movement of data between main memory and external memory (disk), rather than actual (CPU) computation time, is often the bottleneck in the computation. Since data is moved between disk and main memory in large contiguous blocks, this has led to the development of a large number of I/O-efficient algorithms that minimize the number of such block movements. However, actually implementing these algorithms can be somewhat of a challenge since operating systems do not give complete control over movement of blocks and management of main memory.

TPIE is one of two major libraries that have been developed to support I/O-efficient algorithm implementations. It relies heavily on the fact that most I/O-efficient algorithms are naturally composed of components that stream through one or more lists of data items, while producing one or more such output lists, or components that sort such lists. Thus TPIE provides an interface where list stream processing and sorting can be implemented in a simple and modular way without having to worry about memory management or block movement. However, if care is not taken, such streaming-based implementations can lead to practically inefficient algorithms since lists of data items are typically written to (and read from) disk between components.

In this paper we present a major extension of the TPIE library that includes a pipelining framework that allows for practically efficient streaming-based implementations while minimizing I/O-overhead between streaming components. The framework pipelines streaming components to avoid I/Os between components, that is, it processes several components simultaneously while passing output from one component directly to the input of the next component in main memory. TPIE automatically determines which components to pipeline and performs the required main memory management, and the extension also includes support for parallelization of internal memory computation and progress tracking

across an entire application. Thus TPIE supports efficient streaming-based implementations of I/O-efficient algorithms in a simple, modular and maintainable way. The extended library has already been used to evaluate I/O-efficient algorithms in the research literature, and is heavily used in I/O-efficient commercial terrain processing applications by the Danish startup SCALGO.

3.1 Introduction

When handling large datasets that exceed the capacity of the main memory, movement of data between main memory and external memory (disk), rather than actual (CPU) computation time, is often the bottleneck in the computation. The reason for this is that disk access is orders of magnitude slower than internal memory access. Thus, since data is moved between disk and main memory in large contiguous blocks, it is often more important to design algorithms that minimize block movement than computation time when handling massive data. This has led to the development of a large number of *I/O-efficient algorithms* in the I/O-model by Aggarwal and Vitter [6]. In this model, the computer is equipped with a two-level memory hierarchy consisting of an internal memory capable of holding M data items, and an external memory of conceptually unlimited size. All computation has to happen on data in internal memory, and data is transferred between internal and external memory in blocks of B consecutive data items. Such a transfer is called an *I/O-operation* or *I/O*, and the cost of an algorithm is the number of I/Os it performs. The number of I/Os required to read or write N items from disk is $\text{Scan}(N) = \lceil N/B \rceil$, while the number of I/Os required to sort N items is $\Theta(\text{Sort}(N)) = \Theta((N/B)\log_{M/B}(N/B))$ [6].

While many I/O-efficient algorithms have been developed in the I/O-model of computation, actually implementing these algorithms can be somewhat of a challenge since operating systems do not give complete control over movement of blocks and management of main memory. However, two major libraries TPIE [85] and STXXL [46] have been developed to support I/O-efficient algorithm implementations. It turns out that most I/O-efficient algorithms are naturally composed of components that stream through one or more lists of data items, while producing one or more such output lists, or components that sort such lists. TPIE in particular uses this to provide an interface where list stream processing and sorting can be implemented in a simple and modular way, without having to worry about memory management or block movement. However, if care is not taken, such a streaming-based implementation can lead to practically inefficient algorithms since lists of data items are typically written to (and read from) disk between components. In implementations consisting of many small (but I/O-efficient) components, the I/Os incurred when writing and reading such lists can easily comprise more than half of the total number of I/Os. While this may not be a problem when considering asymptotic theoretical

performance, it is unacceptable in practice when the total execution time is measured in hours or days.

In this paper we present a major extension of the TPIE library that includes a pipelining framework that allows for practically efficient streaming-based implementations while minimizing I/O-overhead between streaming components. The framework pipelines streaming components to avoid I/Os between components, that is, it processes several components simultaneously while passing output from one component directly to the input of the next component in main memory. TPIE automatically determines which components to pipeline and performs the required main memory management, and the extension also includes support for parallelization of internal memory computation and progress tracking across an entire application. Thus TPIE supports efficient streaming-based implementations of I/O-efficient algorithms, and TPIE applications are naturally implemented as reusable components, thereby reducing programming time and code duplication.

3.1.1 Previous Work

As mentioned, two major software libraries support I/O-efficient algorithm implementations for big data analysis, namely TPIE [85] and STXXL [46]. They are both C++ software libraries, and as opposed to many of the frameworks that have emerged for supporting big data analysis in the last decade, such as e.g. MapReduce [45], Spark [101], and Flink [9], they mainly support single-host implementations. One reason for this is that the libraries, in particular TPIE, are designed to support implementations on standard commodity hardware. Another reason is that no efficient distributed algorithms are known for many of the problems for which I/O-efficient algorithms have been studied and implemented; we refer to surveys [11, 99] and descriptions of implementations (e.g. [7, 12, 27, 47, 82]) for references. Thus in this paper we also focus on single-host implementations. However, is should be mentioned that in the context of distributed programming, pipelining has recently been studied with the Thrill framework [35].

Although both are libraries for implementation of I/O-efficient algorithms, the overall philosophies of TPIE and STXXL are somewhat different. The philosophy of TPIE (the Templated Portable I/O Environment) is to provide a high-level interface that allows for easy translation of abstract I/O-efficient algorithm descriptions into code that is portable across computational platforms and not unnecessarily complex. Thus building on the fact that most I/O-efficient algorithms are composed of streaming components, TPIE provides a generic stream interface that hides how blocked I/O is performed and instead provides methods for processing one data item at a time. TPIE also provides internal memory management, where memory allocations are automatically counted towards an application-wide memory limit, and where an application can at any point determine the currently available main memory. Thus applications

do not have to explicitly keep track of available memory, which often simplifies implementations considerably. For example, in the TPIE built-in streaming-based implementation of the I/O-optimal $O(\text{Sort}(N))$ external multi-way merge-sort, the number of sorted streams that can be merged I/O-efficiently (without being swapped out by the operating system) depends on the available main memory, where care has to be taken to ensure that the memory used to hold blocks of items for each used stream is counted towards the amount of available memory; the TPIE memory management allows for determining the number of streams to merge without explicitly keeping track of available memory and memory used for blocked I/O. Overall, TPIE is designed to remove focus from the tedious details of creating I/O-efficient applications and allows for implementations that are efficient on all hardware platforms with minimal configuration.

The philosophy of STXXL (Standard Template library for XXL data sets) on the other hand is to achieve maximum I/O-throughput by reducing I/O-overhead as much as possible, e.g. by exposing the characteristics of the hardware to the application programmer. Thus, to avoid any overhead induced by the operating system, STXXL allows the user to configure separate disks for use with applications outside of the file system of the operating system. In fact, STXXL project programmers recommend that a separate disk is set aside for STXXL applications. STXXL also explicitly supports parallel disks. Like TPIE, STXXL supports streaming-based implementations and includes various basic streaming components such as sorting, but unlike TPIE it actually contains support for pipelining of streaming components. However, STXXL expects the application programmer to explicitly define which components to pipeline and explicitly manage main memory. Thus, the programmer e.g. has to specify how much memory each streaming component in a pipelined application should use. A separate (not officially released) branch of STXXL contains support for utilizing multi-core processors for the internal-memory work of pipelined applications [34]. Overall, STXXL is designed such that an application can be tailored to the available hardware, and with the proper configuration an STXXL application can achieve close to full utilization of the available I/O bandwidth.

3.1.2 Our Results

In this paper we present a major extension of the TPIE library that includes a pipelining framework that allows for practically efficient streaming-based implementations while minimizing I/O-overhead between streaming components. The extension also includes support for progress tracking across an entire application, and for parallelization of internal memory computation.

Like STXXL, the TPIE pipelining framework saves I/Os by passing intermediate results between streaming components directly in main memory. However, TPIE pipelining is the first framework to provide automatic pipeline

and memory management, and thus combining the best of the TPIE and STXXL streaming philosophies. The framework is component-centric in that the memory requirement of each streaming component is specified locally by the component developer. The automatic pipeline and memory management then means that at runtime TPIE will automatically determine which components to pipeline, and distribute memory among multiple components of a large application in a way that automatically uses all the main memory available to the application. Thus, unlike in STXXL, a TPIE programmer e.g. does not have to consider how the memory use of the individual components has to be adjusted when they are combined into an application. While such adjustments along with adjustments of the grouping of components into pipelines can be done manually for small projects, it can be very cumbersome for large-scale professional software projects involving many programmers, where modification of a component to use more memory can very easily lead to memory over-usage problems (if the memory use of other components are not adjusted accordingly). Thus the TPIE component-centric approach simplifies the application development process, promotes modularity and supports maintainability.

Since I/O-efficient applications are typically long-running processes that take hours or days to complete, it is important that an application is able to provide a progress bar that gives a precise estimate of the progress of its execution. To be able to do so in a simple way, the TPIE extension also takes a component-centric approach. Like for memory use, a component developer can in a simple way include support for information about the progress of the component, and TPIE automatically combines information from all components and thus supports a single progress bar that advances from 0% to 100% at a constant pace. Thus, again the use of a component-centric approach promotes modularity.

Especially after minimizing I/O, the use of multi-core parallelization can often help to bring down the running time of massive data applications. Thus the TPIE extension includes easy support for such parallelization by allowing the application programmer to wrap a part of a pipeline in a parallelization directive that will trivially parallelize it across all CPU cores. For instance, when forming sorted runs in multi-way merge sort, the internal memory sorting algorithm in TPIE automatically uses all the available CPU cores.

Overall, the major TPIE library extension presented in this paper supports efficient streaming-based implementations of I/O-efficient algorithm in a simple, modular and maintainable way, and I/O-efficient algorithms can thus be composed and adapted in commercial and research applications while dealing systematically with important aspects, such as memory management and progress tracking, that are not intrinsic to the algorithmically optimal solution. The extended library has already been used to evaluate I/O-efficient algorithms in the research literature (e.g. [12, 27]) and is heavily used in I/O-efficient

commercial terrain processing applications by the Danish startup SCALGO[1]. The extension is integrated into the official TPIE project that is available on GitHub as free and open-source software[2].

The rest of the paper is structured as follows. In Section 3.2, we motivate pipelining with a concrete algorithm based on scanning and sorting. After this motivation, we in Section 3.3 present in full generality how to use the TPIE extension and briefly discuss its implementation.

3.2 An Example Problem

In this section we present an example of a typical sub-problem in an I/O-efficient application. We show that the problem benefits from a pipelined implementation; by implementing every sub-problem in a bigger data processing application (such as the real-world example in Figure 3.1) using pipelining, more than half of the I/Os can be saved.

3.2.1 The Raster Transformation Problem

In geographic information systems (GIS), a terrain is often represented as a raster of heights with each cell indicating the height of the terrain in a certain point. Since the Earth is spherical and a raster is flat, it is not possible to map the entire surface of the Earth continuously to a raster. However, if only a particular region, country or continent needs to be represented, it is possible to project the chosen region to a plane in a way that roughly maintains the geodesic distances and areas. When several rasters must be processed together they must be in the same projection.

We call the problem of transforming a raster from one projection to another the *raster transformation problem*. Essentially, the problem consists of projecting each cell of the raster from the source projection plane to the unit sphere, and from the unit sphere to the target projection plane. These two steps can be represented by a function $f : \mathbb{Z}^2 \to \mathbb{Z}^2$ that maps each cell of the *target* raster projection to the corresponding cell of the *source* raster projection. Thus, in the raster transformation problem we are given an input raster A of size $W \times H$ (that is a W by H matrix of numbers) stored in row-major order, and we want to produce an output raster B of size $W' \times H'$ in row-major order, such that the value of a cell (x, y) in B is copied from the value of a cell $(x', y') = f(x, y)$ in A. Below we for convenience let $N = WH = W'H'$ be the number of cells in both the input and output raster.

The raster transformation problem can easily be solved in optimal $O(N)$ time simply by for each cell (x, y) in B reading the corresponding input value at $f(x, y)$ in A. However, this solution might be very I/O-inefficient. For example,

[1]SCALGO: Scalable Algorithmics. `https://scalgo.com`

[2]TPIE: Templated Portable I/O Environment. `http://madalgo.au.dk/tpie`

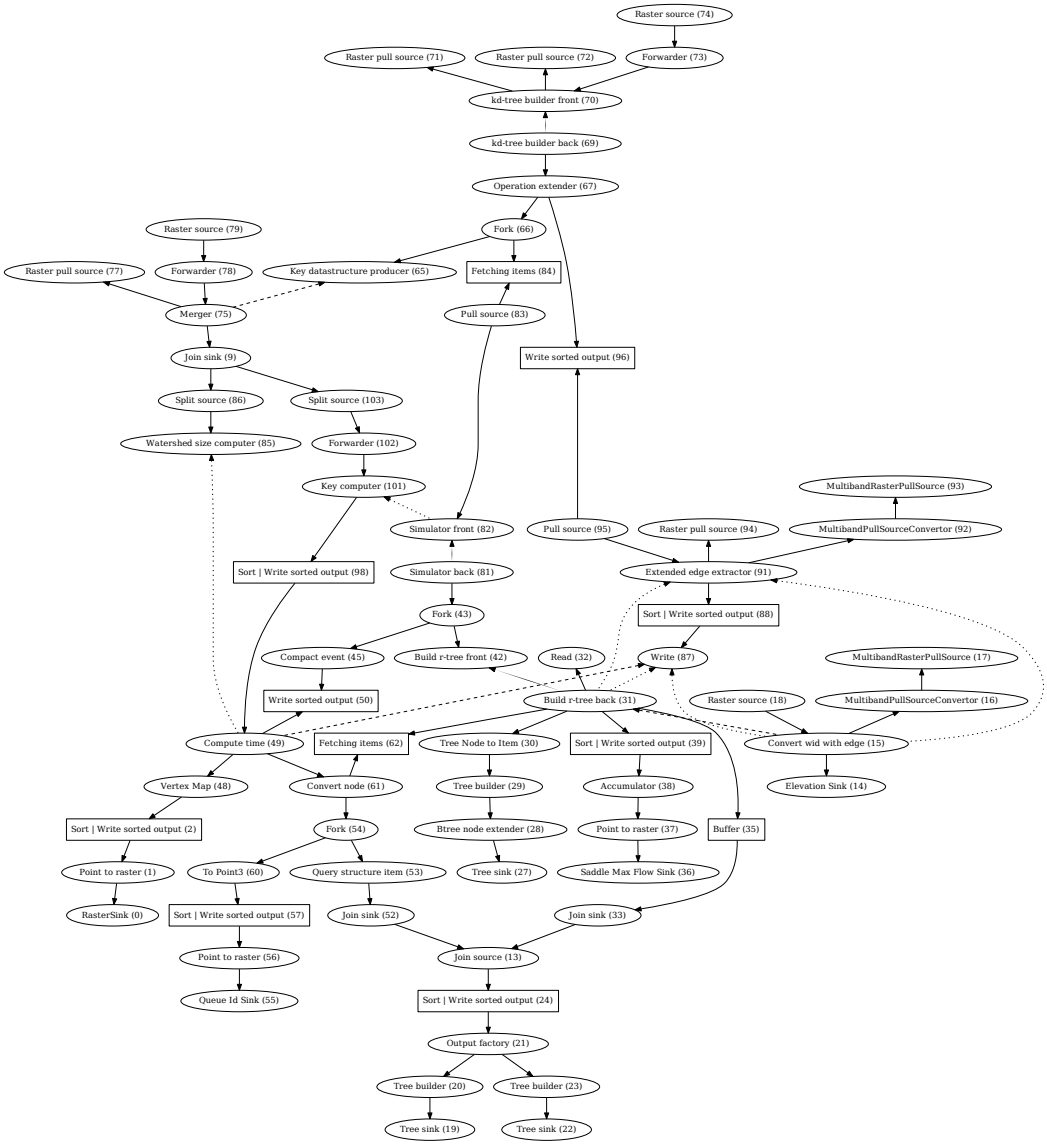


Figure 3.1: Pipelined components in a real TPIE application developed by SCALGO. Dashed lines represent phase ordering dependencies resulting from blocking edges in the flow graph.

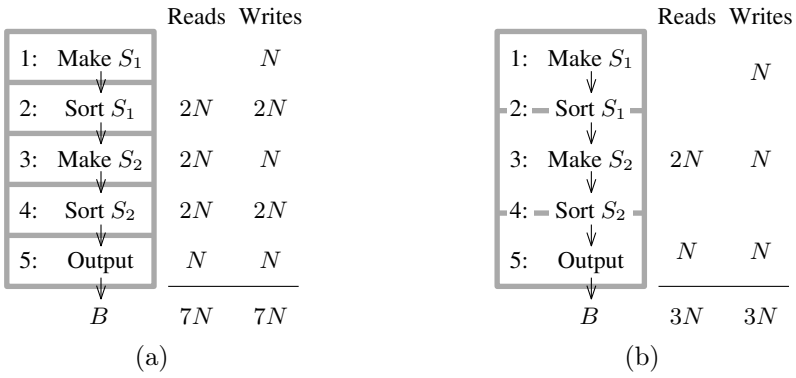


Figure 3.2: Raster transformation algorithm. (a) The algorithm without pipelining, requiring $7N$ reads and writes (assuming use of merge-sort using just one merge step). (b) The algorithm with pipelining, requiring just $3N$ reads and writes.

if f represents matrix transposition where $f(x, y) = (y, x)$, then each access to A requires a new block to be read (assuming $W, H \geq \frac{M}{B}$), and thus the solution performs $\Theta(N)$ I/Os in the worst case. For matrix transposition in particular, only $\Theta(\text{Sort}(N))$ I/Os are required [6]. In fact, in general the raster transformation problem can be solved in $O(\text{Sort}(N))$ I/Os using a simple five step streaming algorithm (refer to Figure 3.2a): First a stream S_1 is constructed containing for each cell (x, y) of B an item consisting of a pair $(f(x, y), (x, y))$. Next S_1 is sorted such that $f(x, y)$ appear in the same row-major order used to store A. In the third step, A and S_1 are then scanned simultaneously to construct a stream S_2 containing an item (x, y, v) for each pair $(f(x, y), (x, y))$ in S_1 where v is the value of A at position $f(x, y)$. Then S_2 is sorted into the row-major order used to store B. In the fifth and final step, S_2 is scanned and for each entry (x, y, v) the value v is output to $B[x, y]$. Since the algorithm performs a constant number of scanning and sorting steps it uses $O(\text{Sort}(N))$ I/Os and can easily be implemented using the streaming support of either TPIE or STXXL. Refer to Figure 3.3 for a TPIE code example.

As discussed in the introduction, streaming-based implementations of even simple I/O-efficient algorithms, as the raster transformation algorithm above, might not be practically efficient because items are written to disk between steps. To illustrate this, we will analyze the exact number of items read and written by the above algorithm. For simplicity, we assume that N elements can be sorted using $2N$ reads and $2N$ writes, which is a practically realistic assumption if external merge-sort is used. Recall that external merge-sort works by first scanning through the N input elements and sorting M elements at a time in internal memory to produce $\frac{N}{M}$ sorted runs. This requires N reads and N writes. Next the sorted runs are merged together $\frac{M}{B}$ at a time (using a block of internal memory for each run) to produce $\frac{N}{M}/\frac{M}{B}$ longer sorted runs,

```
1  struct vec2 { int x, y; };
2  vec2 f(vec2 a) { return {a.y, a.x}; } // Here, f is matrix transposition
3  bool operator<(vec2 a, vec2 b) { return (a.y != b.y) ? (a.y < b.y) : (a.x < b.x); }
4  struct map_point { vec2 from, to; };
5  bool operator<(map_point a, map_point b) { return a.from < b.from; }
6  struct value_point { vec2 point; float value; };
7  bool operator<(value_point a, value_point b) { return a.point < b.point; }
8
9  // Sort the input points into the order in which they appear in the output.
10 tpie::file_stream<map_point> stream1; stream1.open();
11 for (int y = 0; y < outputysize; ++y)
12     for (int x = 0; x < outputxsize; ++x) {
13         map_point p = { f({x, y}), {x, y} };
14         if (0 <= p.from.x && p.from.x < xsize && 0 <= p.from.y && p.from.y < ysize)
15             stream1.write(p);
16     }
17 // Sort the input points in row-major order, so we can scan them simultaneously with A.
18 tpie::sort(stream1);
19 // Scan input raster and input point stream, filling the input points with values.
20 stream1.seek(0); // Seek to beginning of stream
21 tpie::file_stream<value_point> stream2; stream2.open();
22 tpie::array<float> row1(xsize);
23 for (int y = 0; y < ysize; ++y) {
24     input.read_next_row(&row1);
25     while (stream1.can_read() && stream1.peek().from.y == y) {
26         map_point p = stream1.read();
27         stream2.write(value_point{ p.to, row1[p.from.x] });
28     }
29 }
30 stream1.close();
31 // Sort the filled input points into output order.
32 tpie::sort(stream2);
33 // Write the output points to a raster.
34 stream2.seek(0); // Seek to beginning of stream
35 tpie::array<float> row2(outputxsize);
36 for (int y = 0; y < outputysize; ++y) {
37     for (int x = 0; x < outputxsize; ++x) row2[x] = nodata;
38     while(stream2.can_read() && stream2.peek().point.y == y) {
39         value_point p = stream2.read();
40         row2[p.point.x] = p.value;
41     }
42     output.write_next_row(&row2);
43 }
44 stream2.close();
```

Figure 3.3: Raster Transformation Using TPIE Without Pipelining

again using N reads and N writes. The merging process is repeated until a single sorted output is obtained. However, in practice (where N, M and $\frac{M}{B}$ are on the order of 10^{12}, 10^9 and 10^3, respectively) $\frac{N}{M}/\frac{M}{B} < 1$ so only a single merging step is required. Using this, we can easily realize that the above raster transformation algorithm requires $7N$ reads and $7N$ writes without pipelining (refer again to Figure 3.2a): Generating the stream S_1 in the first step requires N writes, and sorting it in the second step requires $2N$ reads and $2N$ writes. Reading A and S_1 simultaneously in the third step to produce S_2 requires $2N$ reads and N writes. Again, sorting S_2 in the fourth step requires $2N$ reads and writes, and finally, reading S_2 and writing B in the fifth step requires N reads and writes. However, by modifying the five steps of the algorithm so that the intermediate result of one step is immediately used by the next step (if possible) without storing the intermediate result on disk, that is, by using pipelining, we can reduce the number of reads and writes to $3N$ each. More precisely, we can save N writes and N reads of S_1 between step one and two by immediately producing the initial sorted runs of step two while performing step one. Similarly, we can save the N writes and N reads of S_1 between step two and three by performing step three (scanning S_1 and A) simultaneously with merging the sorted runs. Note that apart from the write and read between the run formation and merging in step two, we in this way avoid writing S_1 altogether. In a similar way, we can avoid writing S_2 and save N reads and N writes by also producing the initial sorted run of step four while performing step 3, as well as N reads and N writes by performing step five simultaneously with merging of the sorted runs in step 4. Altogether, we save $4N$ reads and $4N$ writes, that is, over half of the I/Os. Although this does not change the asymptotic I/O-complexity of the algorithm, it translates into a running time reduction of 22 hours if we assume an input size N of 1 TB and a disk read/write speed of 100 MB/s.

Note that the pipelining process described above conceptually transforms the five-step algorithm into a three-phase algorithm as indicated in Figure 3.2b, where e.g. the second phase consists of the merging part of the step two sorting, step three, and the run formation part of the step four sorting. One could of course implement the algorithm by implementing these three phases directly, that is, by implementing several special versions of external merge-sort (or rather, special run formation, merging, and merging-run formation implementations). However, this would not only be cumbersome, but also unacceptable from a software engineering point of view. Instead, direct support of pipelining where the five-step algorithm is automatically pipelined would be desirable. However, such a pipelining would require system support for identification of phases and careful memory management. For example, the merge and run formation parts of phase two of the three-phase algorithm would normally both require all of the main memory, so the memory somehow needs to be divided between the two parts. As described below, this is handled somewhat differently in STXXL and the new TPIE extension.

```
1  void transform(raster_input & in, raster_output & out, size_t mem_available) {
2      // In phase 1, the single sorter can use all the available memory.
3      const size_t phase1_sort_memory = mem_available;
4      // In phase 2, the two sorters receive each half the available memory,
5      // excluding the memory used to store a single block from the input.
6      const size_t phase2_sort_memory = (mem_available − in.buffer_size()) / 2;
7      // In phase 3, there is a single sorter and an output buffer.
8      const size_t phase3_sort_memory = mem_available − out.buffer_size();
9      GenerateOutputPoints out_points(out.dimensions());
10     typedef TransformPoints<GenerateOutputPoints> TransformOutputPoints;
11     TransformOutputPoints point_pairs(std::move(out_points), in.dimensions());
12     typedef stxxl::stream::runs_creator<TransformOutputPoints, input_yorder> rc_t;
13     rc_t rc(std::move(point_pairs), input_yorder(), phase1_sort_memory);
14     typedef stxxl::stream::runs_merger<rc_t::sorted_runs_t, input_yorder> rm_t;
15     // The following call to rc.result() executes the first phase.
16     rm_t rm(rc.result(), input_yorder(), phase2_sort_memory);
17     RasterReader in_raster_reader(in);
18     typedef PointFiller<rm_t, RasterReader> FillOutputPoints;
19     FillOutputPoints filler(std::move(rm), std::move(in_raster_reader));
20     typedef stxxl::stream::runs_creator<FillOutputPoints, point::yorder> rc2_t;
21     rc2_t rc2(std::move(filler), point::yorder(), phase2_sort_memory);
22     typedef stxxl::stream::runs_merger<rc2_t::sorted_runs_t, point::yorder> rm2_t;
23     // The following call to rc2.result() executes the second phase.
24     rm2_t rm2(rc2.result(), point::yorder(), phase3_sort_memory);
25     // The following call to write_raster() executes the third phase.
26     write_raster(std::move(rm2), out);
27 }
```

Figure 3.4: Raster transformation using the STXXL streaming layer.

3.2.2 STXXL Implementation

When implementing the raster transformation algorithm with pipelining using the STXXL streaming layer, the five steps of the algorithm are implemented individually as is natural from a software engineering point of view; we call each such individual part of a pipeline a *component*. However, since STXXL does not handle memory management, the implementation that combines the components then has to identify the three phases of the algorithm explicitly and compute how much memory is allocated to each of the components of a phase. Refer to Figure 3.4 for STXXL code that implements this, that is, the main code that implements the five step algorithm in three phases (excluding the code for the individual components). The code illustrates how three phases are explicitly identified and memory allocated. For example, in phase two the memory available for the two sorting components (merging of step two and run formation of step four) is computed by setting aside a buffer of size B of the available main memory for reading the input, and then share the remaining memory between the two sorting components. Concretely, the computation is

performed with the statement: sort_memory = (memory_available − block_size) / 2. While identifying phases and allocating memory in this way is easy in our simple example algorithm, it is more difficult in larger applications such as the example given in Figure 3.1. Especially if more than one programmer is working on the application it is difficult and error-prone to distribute memory correctly.

Apart from the complexity that the need for phase identification and memory allocation adds to pipelined STXXL code, there are also some C++ syntax issues that add to the code complexity. More precisely, the C++ syntax used is quite verbose, since for technical reasons names of the components often need to be repeated. The reason is that STXXL combines pipelining components using a C++ feature known as *template instantiation* that allows for the compiler to inline function calls between different components of the pipeline. For performance reasons, this is necessary when many small components are pipelined. However, the template instantiation syntax is not well-suited for use in large pipelined applications.

As an example, consider the C++ statement typedef TransformPoints<GenerateOutputPoints> TransformOutputPoints; in the STXXL implementation of the raster transformation algorithm. In this statement, the C++ language typedef statement is used to declare TransformOutputPoints to be a *type alias* for the TransformPoints component instantiated with the GenerateOutputPoints component. Such a type alias is needed for each component of the pipeline, and only when all the type aliases have been defined the individual component objects can be declared and constructed. In this way, the pipeline has to be defined both in terms of type aliases nested within each other and as actual component objects combined together.

3.2.3 TPIE Implementation Using Pipelining

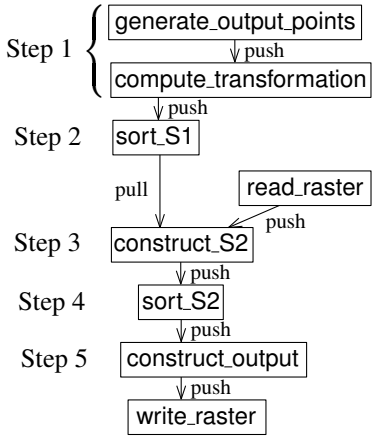
As in the case of pipelined STXXL, in the implementation of the five step raster transformation algorithm using the extended TPIE library with pipelining, the components of the pipeline are implemented individually. However unlike in the STXXL implementation, the combination of the components becomes very simple, since TPIE is component-centric and automatically identifies phases and performs memory management. To illustrate this, a diagram showing the eight components used to implement the five steps is given in Figure 3.6a along with the pipelining code in Figure 3.6b. Note how the code in Figure 3.6b lines 3-12 naturally corresponds to the pipeline in Figure 3.6a. Note also that the reading and writing of rasters are handled by two special components to separate the handling of specific raster formats from the algorithm, and how the two sorting components are implemented using two different built-in TPIE sorting components defined in lines 3 and 4 on Figure 3.6b. The reason two different sorter implementations are used (and that the pipeline is defined in two statements defining p1 and p2, respectively) is that the output from

```
1  template <typename dest_t>
2  struct GenerateOutputPoints : public tpie::pipelining::node {
3      GenerateOutputPoints(dest_t d): dest(std::move(d)) {}
4      virtual void propagate() override {
5          dimensions = fetch<rastersize_t>("outputsize");
6          set_steps(dimensions.width * dimensions.height);
7      }
8      virtual void go() override {
9          for (int y = 0; y < dimensions.height; y++)
10             for (int x = 0; x < dimensions.width; x++)
11                 { step(); dest.push(point(x, y)); }
12     }
13     rastersize_t dimensions; dest_t dest;
14 };
15 typedef tp::pipe_begin<tp::factory<GenerateOutputPoints>>
16     generate_output_points;
```

Figure 3.5: generate_output_points component used in the TPIE raster transformation algorithm.

the component sorting S_1 has to be read by the component constructing S_2 simultaneously with the output from the component reading the input raster A. Thus, the component constructing S_2 has to control when data is received from the sorting component, which is done through *pull-based streaming*. This functionality is implemented with a TPIE so-called *passive sorter* with an input and an output part defined in line 3. On the other hand, the component sorting S_2 is a more traditional pipelined component that uses *push-based streaming*, where input data is received from preceding component (in this case the component constructing S_2) when ready, and output data in turn pushed to the subsequent component. It is defined with an ordinary TPIE sorter in line 4. As an example of a component implementation, the code implementing the component generate_output_points is given in Figure 3.5. The component contains a method propagate() that is called by TPIE when setting up the pipeline, and a method go() that is called by TPIE to execute the actual component. The component also uses push-based streaming, and it pushes each produced element to the next component by calling the push() method of that component. The details of how the push and pull mechanisms work will be discussed in Section 3.3, where the full TPIE pipelining framework and its implementation is described. Below we highlight some of the other framework features that are used in the raster transformation example.

Memory management As mentioned, TPIE automatically manages memory and divides available memory among components in a pipeline. Thus in the pipeline definition in Figure 3.6b there is no code at all dealing with memory distribution. Often many components use only a small amount of static



(a)

```
1   void transform(raster_input & A, raster_output & B,
2                  tpie::progress_indicator_base & pi) {
3       auto sort_S1 = tpie::pipelining::passive_sorter<projected_point>();
4       auto sort_S2 = tpie::pipelining::sort(point::yorder());
5       tpie::pipelining::pipeline p1 = generate_output_points()
6           | tpie::pipelining::parallel(compute_transformation())
7           | sort_S1.input();
8       tpie::pipelining::pipeline p2 = read_raster(A)
9           | construct_S2(sort_S1.output())
10          | sort_S2
11          | construct_output()
12          | write_raster(B);
13      p1.forward("inputsize", A.dimensions());
14      p1.forward("outputsize", B.dimensions());
15      uint64_t n = A.cell_count() + B.cell_count();
16      p1(n, pi, TPIE_FSI); // Execute the pipeline
17  }
```

(b)

Figure 3.6: Raster transformation algorithm. (a) Pipeline illustrated as components. (b) TPIE code implementing the pipeline.

memory, whereas components such as sorting require dynamically allocated memory depending on the amount of available memory. In the latter case the component has to specify its minimum and maximum memory requirements in its implementation. In the example component shown in Figure 3.5 no requirement is specified since only static memory is used.

Metadata Often many components in a pipeline need some sort of metadata about the items that are being streamed between components. In the example, certain components need to use the dimensions of the input and output rasters. While such metadata can of course be passed as parameters to the individual components in the pipeline definition, doing so makes the definition needlessly cluttered. Instead, TPIE provides a general facility for passing metadata between pipeline components. Thus, in Figure 3.6b lines 13-14, the pipeline definition uses forward() to pass the dimensions of the input and output rasters to the components that need them. The individual components can then obtain the metadata using fetch(), such as when the component generate_output_points in Figure 3.5 line 5 retrieves the dimensions of the output raster.

Progress reporting In the example, the TPIE support for progress reporting (e.g. as a progress bar) is also used. As with memory requirements, the code required to supply progress information is not part of the code in Figure 3.6b where the pipeline is defined, but rather part of the implementation of the individual component. Thus, the component in Figure 3.5 provides the needed information by using set_steps() in line 6 in the propagate() method to define how many items it will produce, and then calling a progress stepping function in line 11 in the go() method for each item that it produces. When executing the pipeline in line 16 of Figure 3.6b the argument pi is a reference to a progress indicator object that tells TPIE how to display progress. To provide accurate progress estimations, TPIE actually uses statistical information about progress of previous runs of the code. To store information about runs, the problem's instance size n, computed in line 15, as well as a symbol TPIE_FSI used to identify the application being executed, are also passed to the TPIE framework when executing the pipeline in line 16.

Parallelism The example takes advantage of TPIE support for multi-core CPU parallelism in two ways. First, in the TPIE built-in implementation of multi-way merge-sort, the initial run-formation phase is automatically parallelized. Second, by wrapping the component compute_transformation in the directive tpie::pipelining::parallel(...) in line 6, TPIE automatically distributes this part of the computation among all available CPU cores.

3.3 TPIE Pipelining

In this section we describe the TPIE pipelining framework in more detail. First in Section 3.3.1 we describe how to use the framework, and then in Section 3.3.2 we discuss some aspects of the implementation of the framework.

3.3.1 Pipelining Use

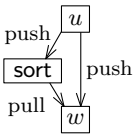
In this section we first describe how a TPIE pipeline consisting of a number of components can be modeled using a so-called flow graph, and how this graph can be used to identify pipeline phases. Then we describe how components are implemented. Finally, we describe how a TPIE pipeline is constructed and executed.

Flow graph and phase identification As described in Section 3.2.3, a TPIE pipeline consists of a number of components that push data to or pull data from other components. We distinguish between two types of components, namely *regular components* that produce output as the input is processed, and *blocking components* that have to process all input before producing any output. Blocking components consist of two sub-components, namely an input and an output sub-component, where the input sub-component processes all input before the output sub-component is invoked to produce the output; the input sub-component might store intermediate results on disk. Merge-sorting is an example of a blocking component; the input sub-component naturally corresponds to the initial run formation step and the output sub-component to the merging step. Blocking components introduce the need for *pipeline phases* that are executed independently, since the input and output sub-components cannot be executed simultaneously. In the raster transformation algorithm in Section 3.2, the three phases are exactly needed due to the two blocking sorting components.

A pipeline can conveniently be represented by a directed acyclic *flow graph*, with *regular nodes* corresponding to regular components and *input nodes* and *output nodes* corresponding to the sub-components of blocking components. Regular nodes are connected with other regular nodes and input and output nodes by edges directed along the streaming direction and labeled as *push edges* or *pull edges* in a natural way; note that a node cannot have both an outgoing push and an outgoing pull edge. Each input node is also connected with a directed *blocking edge* to the corresponding output node. To be able to automatically determine the phases of a pipeline, TPIE requires that the flow graph corresponding to the pipeline has two particular properties: First, if all blocking edges are removed, then no input and output nodes corresponding to the same blocking component should be in the same connected component. Second, if all push and pull edges are contracted, then the graph should be acyclic. The first property means that the connected components directly

identify the pipeline phases that need to be executed independently. When each such connected component is contracted, each directed edge (u, v) in the resulting graph indicates that the phase corresponding to u needs to be executed before the phase corresponding to v. Thus the second property ensures that there exists a valid (topological) order in which to execute the components.

While the second flow graph property above has to be fulfilled for any pipelined program to be valid, the first property only has to be fulfilled if we require that the program is constructed such that all but blocking components can be pipelined, that is, that streaming items are only written to disk by blocking components. For example, consider the small pipeline shown on the right, where component u pushes to both a sorter and to a component w, which in turn pulls output from the sorter. In this case, u, w and the input and output nodes of the sorter are all in the same connected component in the phase graph without blocking edges. However, it is obviously not possible to pipeline all the components in one phase. In particular, it is not possible to pipeline u and w, since the output from the sorter used in w is not available at the same time as the output from u also used in w. To remedy this problem, and make the flow graph fulfill the second property, a simple blocking component that delays the stream of items from u to w, by writing them temporarily to disk, can be inserted between u and w in the pipeline (such that the example has two phases). To support this, TPIE not only contains a built-in sorting blocking component, but also blocking components that delay and reverse a stream. Each of these components come in an *active* and a *passive* variant. In the active variant both the input and output sub-components use push-based streaming, and in the passive variant the input sub-component is push-based and the output sub-component pull-based.



Component implementation The interface of each component in a TPIE pipeline must have certain methods; sub-components are essentially like regular components, so when we refer to components below we mean regular components and sub-components.

Methods push(), pull(), and can_pull() are used to stream data items between components. Consider a component corresponding to a node u in the flow graph that produces data that is processed by a component corresponding to a node v in the graph, that is, where there is an edge (u, v). With a slight abuse of notation, we use u and v to refer to the two components. If the edge is a push edge we say that v is a *destination* of u, and then v must implement a push() method and u must push each item in the stream to v by calling the method v.push(). If on the other hand the edge is a pull edge we say that u is a *source* of v, and then u must implement a pull() method and v must pull each item from u by calling the method u.pull(); u must also implement the method can_pull() to return true if there is more data to pull and false otherwise.

A component u that is neither the destination or the source of any other component must implement a go() method that repeatedly pushes or pulls data until there is no more data to process. This is because data is neither pushed to or pulled from u by other components calling u.push() or u.pull(). Thus the go() method is used to start the execution of a phase.

Each component u must implement (possibly empty) begin() and end() methods that are called before and after the stream processing of the phase containing u, respectively. These methods can for example be used to allocate and deallocate memory used by u, or set up data structures needed by u. Component u is also allowed to push items to its destinations and pull items from its sources in begin() and end(). This is e.g. useful when buffers need to be filled up at the beginning or emptied at the end of a phase.

Each component u must also implement a (possibly empty) propagate() method that is also called before any stream processing in the phase containing u and used to pass metadata between components. Inside the propagate() method u may use the forward function to pass key-value pairs to components v that can be reached from u in the flow graph. It may use the fetch function to retrieve named metadata from other components. Often metadata includes information about input and output data size, and if TPIE should provide progress reporting, at least one component u in each phase should provide progress information by calling the function set_steps(n) inside the propagate() method to indicate the number of items n that it will process, and then call the step() function once for each item that is processed. Often u is a node with no incoming push or pull edges in the flow graph, that is, the node that creates streaming data.

Finally, a component u that requires dynamically allocated memory to perform its stream processing needs to indicate this to TPIE by calling the functions set_minimum_memory(a_u) and set_maximum_memory(b_u) in its class constructor to request between a_u and b_u bytes of memory, where $b_u = \infty$ is used to indicate that the component requests as much memory as possible. After TPIE has distributed memory, u can then obtain information about how much memory it was assigned between a_u and b_u by calling the function get_available_memory() in the begin() method.

Pipeline construction and execution After defining the pipelining components, a pipeline is constructed by stringing together components using the so-called pipe operator as in the expression p = generate_output_points() | compute_transformation() | sort_S1.input().memory(2) where three components generate_output_points(), compute_transformation() and the input sub-component of sort_S1 are pipelined. For each component, as for the sort_S1.input() component in the example, one can set a memory priority using memory() to indicate to TPIE how important it is to allocate memory to the component; by default the priority is one, and a priority of k means that if several components all request

as much memory as possible using set_maximum_memory(∞) then a component with priority k will receive k times the amount of memory as one with priority one.

To execute the pipeline one simply calls the object p. TPIE then builds the flow graph and computes connected components to identify phases, and then contracts the components and topologically sorts the graph to find the order in which to execute the phases. After this TPIE executes each phase in turn. To execute a phase, TPIE first distributes memory to each component in the phase based on the memory requests and priorities, and then it calls the methods propagate(), begin(), go() and end() on the components in a specific order based on the flow graph. First, propagate() is called on the components in the phase in topological order to allow each component to call forward(), fetch() and set_steps(). The topological order is used since a component u has to be able to pass metadata to components reachable from u in the flow graph. Next begin() is called on all components in the order obtained by topologically sorting the flow graph where all push edges have been reversed; this topological order exists as the graph is acyclic, since a node in the flow graph cannot have both an outgoing push and an outgoing pull edge. This particular topological order is used since for a push edge (u, v), u should be able to push to v in u.begin(), so u.begin() should be called after v.begin(); similarly, if (u, v) is a pull edge, then v.begin() must be called after u.begin() is called. After this initialization, the main streaming part of the phase is executed by calling the go() method on the appropriate component. Finally, at the end of the phase end() is called on the components in reverse order of the begin() order, that is, in reverse topological order.

3.3.2 Pipelining Implementation

Above we have already discussed how TPIE identifies and executes phases of a pipeline, and due to space constraints we cannot describe the entire implementation of TPIE pipelining in detail. In this section we therefore briefly discuss a few aspects of the implementation not described above. The interested reader is also referred to the technical documentation of TPIE [97].

Memory management As mentioned, TPIE distributes memory to components in a phase based on the minimum a_u and maximum b_u memory requirements, along with the memory priority c_u of each component u in the phase. Component u is assigned $M_u(\lambda) = \max\{a_u, \min\{b_u, \lambda c_u\}\}$ bytes of memory, for a value of λ such that the total assigned memory $M(\lambda) = \sum_u M_u(\lambda)$ is smaller than the available memory. This way memory is distributed proportionally to the memory priorities unless this gives an amount of memory outside the $[a_u, b_u]$ interval. Since $M(\lambda)$ is a non-decreasing function of λ, TPIE can use binary search to find λ. Refer to Figure 3.7 for an example.

Node v	Minimum a_v	Maximum b_v	Priority c_v	Assigned $M_v(\lambda)$
A	4	12	**5**	10
B	1	7	**3**	6
C	**8**	∞	3	8
D	7	**12**	7	12

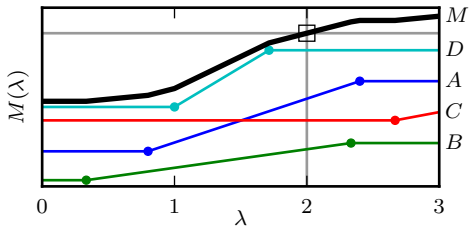


Figure 3.7: Memory assignment for four nodes where $\lambda = 2$, $M(\lambda) = 36$. Nodes A and B are assigned λ times their priority, whereas nodes C and D are assigned their minimum and maximum memory, respectively.

Progress reporting TPIE also supports progress reporting for a pipeline. To ensure that a progress bar shown to the user progresses from 0% to 100% at a constant pace, TPIE maintains an *execution time database* with information about how large a fraction of the execution time was spent in each phase in the execution of the pipeline that processed the largest instance size. In order to distinguish between different pipelines in the execution time database, each pipeline execution carries the preprocessor macro TPIE_FSI as an argument, which is expanded by the compiler into a string that uniquely identifies the location in the code where the pipeline is defined. In this way, TPIE can store information about multiple pipelines in the execution time database.

Automatic parallelization Automatic multi-core CPU parallelism is supported in TPIE by applying the tpie::pipelining::parallel(...) directive to a push-based pipeline component. In this case, the processing of the component is distributed among the available CPU cores by instantiating a copy of the component for each core and passing items to these components in a round-robin fashion as they arrive. To amortize the overhead of thread synchronization, items are passed to components in batches of 2048 items at a time.

Function inlining To minimize the computational overhead of the many push() and pull() function calls required when executing a pipeline, TPIE is designed to allow the compiler to inline the processing of several consecutive pipelining components into one function. As in STXXL, this is achieved using *template instantiations* in C++. Thus, if u pushes to v in the pipeline, then when u is compiled the type of v is known to the compiler so the implementation

of v.push() can be inlined into u. However unlike STXXL, TPIE is designed to hide the resulting complex type definitions from the pipeline definition, and instead of using recursive template instantiations the pipe operator can be used to define a TPIE pipeline. This is very convenient when building large pipelined applications.

Chapter 4

Practical I/O-Efficient Multiway Separators

Abstract

We revisit the fundamental problem of I/O-efficiently computing r-way separators on planar graphs. An r-way separator divides a planar graph with N vertices into $O(r)$ regions of size $O(N/r)$ and $O(\sqrt{Nr})$ boundary vertices in total, where boundary vertices are vertices that are adjacent to more than one region. Such separators are used in I/O-efficient solutions to many fundamental problems on planar graphs such as breadth-first search, finding single-source shortest paths, topological sorting, and finding strongly connected components. Our main result is an I/O-efficient sampling-based algorithm that, given a Koebe-embedding of a graph with N vertices and a parameter r, computes an r-way separator for the graph under certain assumptions on the size of internal memory. Computing a Koebe-embedding of a planar graph is difficult in practice and no known I/O-efficient algorithm currently exists. Therefore, we show how our algorithm can be generalized and applied directly to Delaunay triangulations without relying on a Koebe-embedding. This adaptation can produce many boundary vertices in the worst-case, however, to our knowledge our result is the first to be implemented in practice due to the many non-trivial and complex techniques used in previous results. Furthermore, we show that our algorithm performs well on real-world data and that the number of boundary vertices is small in practice.

Motivated by applications in geometric information systems, we show how our algorithm for Delaunay triangulations can be applied to compute the flow accumulation over a terrain, which models how much water flows over the vertices of a terrain. When given an r-way separator, our implementation of the algorithm outperforms traditional sweep-line-based algorithms on the publicly available digital elevation model of Denmark.

4.1 Introduction

In this paper, we revisit the fundamental problem of computing r-way separators by presenting I/O-efficient algorithms and demonstrating how our results can be applied to I/O-efficiently compute flow accumulation on a terrain. We implement and evaluate our algorithms on real-world terrain data.

The r-way separator is a generalization of the *planar separator theorem* by Lipton *et al.* [74]. The planar separator theorem states that a planar graph with N vertices can be partitioned into two unconnected sets each of size at most $(2/3)N$ by removing $O(\sqrt{N})$ vertices from the graph. Lipton and Tarjan [74] showed that such a partitioning can be computed in linear time in classical models of computation. Frederickson *et al.* [53] described how the planar separator theorem can be generalized to the concept of an *r-way separator*: Given a parameter r, an r-way separator is a division of the vertices of the graph into $O(r)$ non-disjoint *regions* such that each vertex of the graph is contained in at least one region. A region contains two types of vertices: boundary vertices and interior vertices. An *interior vertex* is contained in exactly one region and is adjacent only to vertices in that region. A *boundary vertex* is shared among at least two regions and is adjacent to vertices in multiple regions. Each region contains $O(N/r)$ vertices in total of which $O(\sqrt{N/r})$ are boundary vertices. It follows that the total number of boundary vertices is $O(\sqrt{Nr})$.

The concept of r-way separators is particularly interesting when handling planar graphs that exceed the capacity of the main memory since the computation of such separators can be used to divide the graph into memory-sized regions. In this situation, data is written and read in large blocks to disk, so it is important to design algorithms that minimize the movement of such blocks. This has led to the development of the so-called *I/O model* by Aggarwal and Vitter [6]. In this model, the computer is equipped with a two-level memory hierarchy consisting of an *internal memory* capable of holding M data items and an *external memory* of unlimited size. All computation has to happen on data in internal memory. Data is transferred between internal and external memory in blocks of B consecutive data items. Such a transfer is referred to as an *I/O-operation* or an *I/O*. The cost of an algorithm is the number of I/Os it performs. The number of I/Os required to read N consecutive items from disk is $\mathrm{Scan}(N) = O(N/B)$ and the number of I/Os required to sort N items is $\mathrm{Sort}(N) = \Theta\big((N/B)\log_{M/B}(N/B)\big)$ [6]. For all realistic values of N, M and B we have $\mathrm{Scan}(N) < \mathrm{Sort}(N) \ll N$. Maheshwari *et al.* [80] showed that an r-way separator can be computed in $O(\mathrm{Sort}(N))$ I/Os. This algorithm results in solutions to fundamental graph problems, such as breadth-first search, finding single-source shortest paths, topological sorting, and finding strongly connected components, that uses $O(\mathrm{Sort}(N))$ I/Os [26, 29]. Later, Arge *et al.* [28] presented an I/O-efficient r-way separator algorithm that uses $O(\mathrm{Sort}(N))$ I/Os and $O(N \log N)$ internal memory computation time. Furthermore, they showed that this result can be used to derive algorithms for finding single-source short-

est paths, topological sorting, and finding strongly connected components using $O(\mathrm{Sort}(N))$ I/Os and $O(N \log N)$ internal memory computation time. This improves upon the result by Maheshwari *et al.* by upper bounding the internal memory computation time used.

To our knowledge, no algorithms for I/O-efficiently computing multiway separators have been implemented in practice yet due to the many non-trivial and complex techniques used to derive them. Therefore, we consider the problem of computing r-way planar separators when given a *Koebe-embedding* of the graph. A *Koebe-embedding* of a planar graph is a set of disks in the plane with disjoint interiors where the center of each disk corresponds to a vertex in the graph and two disks are tangent if and only if the corresponding vertices in the graph are adjacent. Miller *et al.* [84] showed that a Koebe-embedding can be used to partition the corresponding graph into two unconnected parts each of size at most $(3/4)N$ by removing $O(\sqrt{N})$ vertices from the graph. In this paper, we present a simple I/O-efficient algorithm that computes an r-way separator for a planar graph when given a *Koebe-embedding* of the graph and having certain assumptions on the size of internal memory.

To our knowledge, the computation of Koebe-embeddings is not trivial and no I/O-efficient algorithms have been presented. Bannister *et al.* [32] showed that computing exact Koebe-embedding requires computing the roots of polynomials of unbounded degree. Thus, the focus of the current state-of-the-art algorithms is to numerically approximate the Koebe-embedding. Orick *et al.* [87] presented an algorithm that approximates a Koebe-embedding by alternating between adjusting radii and positions of vertices. Empirical results show that the algorithm runs in approximately linear time, however, no theoretical worst-case bounds are given. Recently, Dong *et al.* [48] presented an algorithm based on convex optimization that computes an approximate Koebe-embedding in near-linear worst-case time. To our knowledge, these algorithms do not trivially extend to the I/O model.

Motivated by applications in geometric information systems, we show that our algorithm can be adapted to Delaunay triangulation without having to first compute a Koebe-embedding. Delaunay triangulations can be computed using $O(\mathrm{Sort}(N))$ I/Os [2] and are widely used to convert terrain point clouds into so-called triangulated irregular networks which represent a terrain as a triangulated surface. A *triangulated irregular network* (*TIN*) is computed by projecting the terrain point cloud in $\mathbb{R}^3$ onto the xy-plane, computing the Delaunay triangulation of the projected points, and lifting the Delaunay triangulation back to $\mathbb{R}^3$. This adaptation can result in $\Omega(N)$ boundary vertices in the worst case [83]. However, we test our algorithm on the publicly available digital elevation model of Denmark [50] and show that the algorithm results in a small number of boundary vertices in practice.

Finally, we describe how r-way separators can be used to compute the *flow accumulation* over a terrain, which models the flow of water over a terrain represented as a TIN. We consider a variant of the flow accumulation problem,

where we are given a rain distribution function $\mathcal{R}$ that fits in internal memory and assigns $\mathcal{R}(v) \geq 0$ units of water to each vertex v. The water in each vertex v is then distributed by pushing water to a neighboring vertex according to a given flow direction of v. The flow accumulation of a vertex v is the total amount of water that flows through v. This problem is traditionally solved in the I/O-model using $O(\text{Sort}(N))$ I/Os by a sweep-line algorithm where the flow is propagated using a priority queue during a downward sweep of the terrain [62]. In this paper, we adapt the grid terrain algorithm by Haverkort *et al.* [62] to speed up the computation of flow accumulation over a TIN when given an r-way separator of the terrain. Furthermore, we show that this algorithm performs well in practice and outperforms the traditional sweep-line algorithm on the digital elevation model of Denmark when given an r-way separator of the terrain.

4.2 Preliminaries

In this section, we state several preliminary definitions and introduce a more general definition of the r-way separator.

4.2.1 k-ply Neighborhood Systems

We begin by presenting a more formal definition of a Koebe-embedding and then introduce the more general k-ply neighborhood system. This generalization will be used later when applying our result to Delaunay triangulations. The definitions in this section follow Miller *et al.* [84]. Let a *disk packing* be a set of disks $\{B_1, \ldots, B_N\}$ in the plane that have disjoint interiors. Koebe [68] showed that every planar graph can be embedded as a disk packing such that the center of each disk corresponds to a vertex in the planar graph and two disks are tangent if and only if there is an edge connecting the two corresponding vertices in the graph. We refer to this as a *Koebe-embedding* of the planar graph. Note that a partitioning of a Koebe-embedding into disjoint subsets implies a partitioning of the vertices of the graph. Miller *et al.* [84] used this idea to describe how a Koebe-embedding of a planar graph can be used to compute a planar separator. In order to describe this result, we first introduce the more general k-ply neighborhood system:

Definition 1 (k-ply neighborhood system)**.** A k-ply neighborhood system in d dimensions is a set $\Gamma = \{B_1, \ldots, B_n\}$ of closed balls in $\mathbb{R}^d$ such that no point in $\mathbb{R}^d$ is in the interior of more than k of the balls.

In the following sections, we introduce the notion of a separator for k-ply neighborhood systems in $\mathbb{R}^d$ for general k. Observe that a disk packing is a 1-ply neighborhood system and, thus, this separator will also be applicable to Koebe-embeddings.

We now state the planar separator result by Miller *et al.* [84]. A d-dimensional sphere h partitions a k-neighborhood system Γ in $\mathbb{R}^d$ into three subsets: the set $\Gamma(h_>)$ of all balls of Γ contained in the exterior of h, the set $\Gamma(h_<)$ of all balls of Γ contained in the interior of h, and the set $\Gamma(h_=)$ of all balls of Γ that intersect the boundary of h. Correspondingly, we define the subsets $\Gamma(h_\leq) = \Gamma(h_<) \cup \Gamma(h_=)$ and $\Gamma(h_\geq) = \Gamma(h_>) \cup \Gamma(h_=)$.

Theorem 4.1 (Sphere Separator [84])**.** Suppose Γ is a k-ply neighborhood system in $\mathbb{R}^d$ with size $|\Gamma|$. Then there exists a sphere h in $\mathbb{R}^d$ such that

$$|\Gamma(h_<)|, |\Gamma(h_>)| \leq \frac{d+1}{d+2} \cdot |\Gamma| \,,$$
$$|\Gamma(h_=)| = O\big(k^{1/d} \cdot |\Gamma|^{1-1/d}\big) \,.$$

Additionally, Miller *et al.* [84] presented a sampling-based algorithm for approximately computing sphere separators. We state their result with two additional properties that follow from their original proof; first, we state the result when applied to a subset $\Upsilon \subseteq \Gamma$. Note that Υ does not have to be a proper subset. Secondly, we state the number of I/Os used.

Theorem 4.2 (Randomized Separator Algorithm [84])**.** Suppose Γ is a k-ply neighborhood system in $\mathbb{R}^d$, $\Upsilon \subseteq \Gamma$ is a subset of Γ. Then for any constant $\varepsilon > 0$ we can compute a sphere h such that with probability at least $1/2$

$$|\Upsilon(h_<)|, |\Upsilon(h_>)| \leq \left(\frac{d+1}{d+2} + \varepsilon\right) |\Upsilon| \,,$$
$$|\Gamma(h_=)| = O\big(k^{1/d} \cdot |\Gamma|^{1-1/d}\big) \,.$$

Furthermore, the algorithm uses $O(\mathrm{Scan}(|\Upsilon| \cdot d) + c_2)$ I/Os, where c_2 is a constant depending only on ε and d.

We remark that the randomization in the algorithm is over random numbers chosen by the algorithm independent of the input. Therefore, the algorithm can be used to find a sphere satisfying the inequalities by applying the algorithm an expected constant number of times [84]. When describing our algorithm, we will use Theorem 4.2 as a black box and refer to the resulting sphere as a *sphere separator*.

4.2.2 Multiway Separator

We now present a generalization of the sphere separator result by Miller *et al.* [84]. Given a k-ply neighborhood system Γ in $\mathbb{R}^d$ and a parameter $r \leq |\Gamma|/k$, an *r-way division* of Γ is a division of Γ into $O(r)$ non-disjoint *regions* such that each ball in Γ is contained in at least one region. A region contains two types of balls: boundary balls and interior balls. An *interior ball* is contained in exactly one region and has non-empty intersection only with balls contained

in the same region. A *boundary ball* is shared among at least two regions and has non-empty intersection with balls in multiple regions. Each region contains $O(|\Gamma|/r)$ balls in total which are stored consecutively on disk. An *r-way separator* is an r-way division where each region contains $O\big(\sqrt{k|\Gamma|/r}\big)$ boundary balls. It follows that the total number of boundary balls of an r-way separator is $O\big(\sqrt{k|\Gamma|r}\big)$. We use the term *multiway separator* and *multiway division* whenever r is clear from the context.

4.2.3 Range Spaces, VC dimensions, and Samples

The main result of this paper is obtained by computing a multiway separator on a sample of a given k-ply neighborhood system. In order to prove correctness of our algorithm, we show that the result generalizes to the entire neighborhood system with at least constant probability. This proof relies on the concepts of Vapnik–Chervonenkis dimension (VC dimension) [60] and relative ε-approximations [61]. Here, we will provide a quick summary of various definitions and theorems. For a more in-depth introduction to VC-dimension, we refer to Har-Peled *et al.* [60].

Definition 2 (Range Space)**.** A *range space* is a pair $(\mathcal{X}, \mathcal{H})$, where $\mathcal{X}$ is the ground set (finite or infinite) and $\mathcal{H}$ is a (finite or infinite) family of subsets of $\mathcal{X}$. The elements of $\mathcal{H}$ are referred to as *classifiers*.

Definition 3 (VC Dimension)**.** Let $S = (\mathcal{X}, \mathcal{H})$ be a range space. Given $Y \subseteq \mathcal{X}$, let the *intersection* of Y and $\mathcal{H}$ be defined as

$$Y \cap \mathcal{H} = \{h \cap Y \mid h \in \mathcal{H}\} \ .$$

If $Y \cap \mathcal{H}$ contains all subsets of Y, then we say that Y is *shattered* by $\mathcal{H}$. The *VC Dimension* of S, denoted by $\dim_{\mathrm{VC}}(S)$, is the maximum cardinality of a shattered subset of $\mathcal{X}$:

$$\dim_{\mathrm{VC}}(S) = \max\Big\{|Y| \;\Big|\; Y \subseteq \mathcal{X} \wedge |Y \cap \mathcal{H}| = 2^{|Y|}\Big\} \ .$$

If there are arbitrarily large shattered subsets, then $\dim_{\mathrm{VC}}(S) = \infty$.

Lemma 4.1 (VC Dimension of Halfspaces [60, Chapter 5])**.** Let $S = (\mathcal{X}, \mathcal{H}_{\mathrm{halfspace}})$ be the range space where $\mathcal{X} = \mathbb{R}^d$ and $\mathcal{H}_{\mathrm{halfspace}}$ is the set of halfspaces in $\mathbb{R}^d$. Then S has VC dimension $d + 1$.

Lemma 4.2 (Mixing of Range Spaces [60, Chapter 5])**.** Let $S_1 = (\mathcal{X}, \mathcal{H}_1), \ldots, S_k = (\mathcal{X}, \mathcal{H}_k)$ be k range spaces which share the same ground set $\mathcal{X}$ and all have VC dimension at most ξ. Consider the sets of classifiers $\mathcal{H}_\cap$ and $\mathcal{H}_\cup$, where

$$\mathcal{H}_\cap = \{h_1 \cap \cdots \cap h_k \mid h_1 \in \mathcal{H}_1, \ldots, h_k \in \mathcal{H}_k\} \ ,$$
$$\mathcal{H}_\cup = \{h_1 \cup \cdots \cup h_k \mid h_1 \in \mathcal{H}_1, \ldots, h_k \in \mathcal{H}_k\} \ .$$

Then the range spaces $S_\cap = (\mathcal{X}, \mathcal{H}_\cap)$ and $S_\cup = (\mathcal{X}, \mathcal{H}_\cup)$ have VC dimension $O(\xi k \log k)$.

Definition 4 (Measure)**.** Let $S = (\mathcal{X}, \mathcal{H})$ be a range space, and let $X \subseteq \mathcal{X}$ be a finite subset of $\mathcal{X}$. The *measure* of a classifier $h \in \mathcal{H}$ in X is the quantity

$$\bar{X}(h) = \frac{|h \cap X|}{|X|} .$$

Definition 5 (Relative Approximation)**.** Let $S = (\mathcal{X}, \mathcal{H})$ be a range space, and let $X \subseteq \mathcal{X}$ be a finite subset of $\mathcal{X}$. For given parameters $0 < p, \varepsilon < 1$, a subset $Y \subseteq X$ is a *relative* (p, ε)*-approximation* for (X, S) if, for each $h \in \mathcal{H}$, we have

$$\begin{aligned} (1-\varepsilon)\bar{X}(h) \leq \bar{Y}(h) \leq (1+\varepsilon)\bar{X}(h) \quad &\text{if } \bar{X}(h) \geq p . \\ \bar{X}(h) - \varepsilon p \leq \bar{Y}(h) \leq \bar{X}(h) + \varepsilon p \quad &\text{if } \bar{X}(h) < p . \end{aligned}$$

Lemma 4.3 (Relative Approximation Sampling [61])**.** Let $S = (\mathcal{X}, \mathcal{H})$ be a range space with VC dimension ξ, and let $X \subseteq \mathcal{X}$ be a finite subset of $\mathcal{X}$. Given parameters $0 < p, \varepsilon, q < 1$, a random sample $Y \subseteq X$ of size at least

$$\frac{c}{\varepsilon^2 p}\left(\xi \log \frac{1}{p} + \log \frac{1}{q}\right) ,$$

for an appropriate constant c, is a relative (p, ε)-approximation for (X, S) with probability at least $1 - q$.

4.3 Multiway Separator Algorithm for k-ply Neighborhood Systems

In this section, we state our main result for I/O-efficiently computing an r-way separator of a k-ply neighborhood system Γ. The algorithm can be applied to k-ply neighborhood systems in $\mathbb{R}^d$ for any dimensions $d > 2$, however, we prove correctness only for $d = 2$.

We begin by presenting an algorithm that computes an r-way division of Γ under the assumption that $k \leq \log \frac{M}{B} \log\log \frac{M}{B}$. Given a k-ply neighborhood system Γ in the plane and a parameter r, we let $\hat{r} = \min(r, \lfloor M/B \rfloor)$ and compute an r-way division by recursively computing $\hat{r}$-way divisions until Γ is divided into regions of size $O(|\Gamma|/r)$. In order to compute an $\hat{r}$-way division on Γ, we sample a subset $\Upsilon \subseteq \Gamma$ of sufficiently large size. By recursively computing sphere separators using Theorem 4.2, we can compute an $\hat{r}$-way separator for Υ. Let H denote the sphere separators that are computed during the recursion. We refer to H as a *separator tree*. We prove that with at least constant probability we obtain an r-way division by recursively applying the sphere separators of H on Γ. It follows that we obtain an $\hat{r}$-way division for Γ by repeating this sampling-based algorithm an expected constant number of times. This result provides guarantees on the number of boundary balls in the sample Υ, however, we do not prove bounds for the total number of boundary balls in

the r-way division of Γ. We expect the number of boundary balls to be small and confirm so by experimental evaluation in later sections. Additionally, by increasing the sample size and slightly modifying the algorithm, one can remove the assumption on k and prove that the result is an r-way separator for Γ. This results in an I/O-efficient algorithm for computing r-way separators when $\log^3 \frac{M}{B} \log\log \frac{M}{B} \log \frac{|\Gamma|}{k} = O(\sqrt{Mk})$. In other words, we provide guarantees on the number of boundary balls by assuming M is sufficiently large.

The rest of this section is structured as follows: in Section 4.3.1, given $\hat{r} \leq M/B$, we describe how to sample Υ, recursively apply Theorem 4.2, and prove that the result can be used to divide Γ into regions of size $O(|\Gamma|/\hat{r})$ with at least constant probability. In Section 4.3.2, we bound the number of regions to $O(\hat{r})$ and the total number of boundary balls in Υ to $O(\sqrt{k|\Gamma|\hat{r}})$. In Section 4.3.3, we bound the number of boundary balls in each region of Υ to $O(\sqrt{k|\Gamma|/\hat{r}})$. Finally, in Section 4.3.4, we bound the expected number of I/Os used and state the final algorithm.

4.3.1 Recursively Computing Separators

In this subsection, we describe how to sample Υ, recursively apply Theorem 4.2, and prove that the result can be used to divide Γ into regions of size $O(|\Gamma|/\hat{r})$ with at least constant probability, where $\hat{r} \leq M/B$ and assuming $k \leq |\Upsilon|/\hat{r}$.

First, sample $\Upsilon \subseteq \Gamma$ of size at least $c_0 \cdot \hat{r} \log^2 \hat{r} \log\log \hat{r}$, where $c_0 > 0$ is a constant we choose later. Letting $l = O(\log \hat{r})$, we recursively compute sphere separators on Υ for at most l levels; let $\Upsilon_i \subseteq \Upsilon$ denote the balls of Υ that occur in a node i of the recursion. In the root of the recursion, we let $\Upsilon_i = \Upsilon$. At each node of the recursion, we compute a sphere separator h such that $\Upsilon_i(h_\leq)$ and $\Upsilon_i(h_\geq)$ are smaller than Υ_i by at least a constant factor. For now, assume that such a sphere separator h is obtained. We then recurse on the two subproblems $\Upsilon_i(h_\leq)$ and $\Upsilon_i(h_\geq)$. The recursion is continued until the problem size is at most $c \cdot (|\Upsilon|/\hat{r})$, where $c > 0$ is a sufficiently large constant. The separator tree H is then formed from the sphere separators by letting the nodes in H correspond to the recursively computed sphere separators.

We proceed by describing how to compute a sphere separator h in a node i of the recursion. Using Theorem 4.2 and setting $\varepsilon = \frac{1}{12}$, we compute a sphere separator h that with probability at least $1/2$ satisfies

$$|\Upsilon_i(h_<)|, |\Upsilon_i(h_>)| \leq \frac{10}{12} \cdot |\Upsilon_i| \,, \tag{4.1}$$

$$|\Upsilon_i(h_=)| \leq c_1 \sqrt{k|\Upsilon_i|} \,, \tag{4.2}$$

where $c_1 > 0$ is a constant. Note that this uses $O(\text{Scan}(|\Upsilon_i|))$ I/Os. We apply Theorem 4.2 an expected constant number of times until a separator h that satisfies (4.2) and (4.1) is obtained. Since we divide Υ into regions of size at most $c \cdot |\Upsilon|/\hat{r}$, it follows that $|\Upsilon|/\hat{r} \leq |\Upsilon_i|/c$. Using the assumption that

$k \leq |\Upsilon|/\hat{r}$ and that (4.2) holds, we upper bound $|\Upsilon_i(h_=)|$ as follows:

$$|\Upsilon_i(h_=)| \leq c_1\sqrt{k|\Upsilon_i|} \leq c_1\sqrt{\frac{|\Upsilon|}{\hat{r}}|\Upsilon_i|} \leq c_1\sqrt{\frac{|\Upsilon_i|^2}{c}} \leq \frac{c_1}{\sqrt{c}}|\Upsilon_i| \ . \tag{4.3}$$

Thus, for $\sqrt{c} \geq 12 \cdot c_1$, it follows from (4.3) that $|\Upsilon_i(h_=)| \leq \frac{1}{12}|\Upsilon_i|$. Combining this with (4.1), we obtain a separator h that satisfies

$$|\Upsilon_i(h_\geq)|, |\Upsilon_i(h_\leq)| \leq \frac{11}{12}|\Upsilon_i| \ . \tag{4.4}$$

Thus, the problem size becomes smaller by a constant factor and we can recursively compute separators h until Υ is divided into regions of size at most $c \cdot |\Upsilon|/\hat{r}$. This requires at most $l = O(\log r)$ levels of recursion.

We proceed by showing that Γ is divided into regions of size $O(|\Gamma|/\hat{r})$ when Γ is divided recursively using the sphere separators of H. We proceed introducing the following two lemmas:

Lemma 4.4. Let $\mathcal{C}$ be the set of all circles in the plane and let $\mathcal{D}$ be the set of all disks in the plane. Let $\mathcal{H}_\leq$ be the set of classifiers defined as $\mathcal{H}_\leq = \{\mathcal{D}(h_\leq) \mid h \in \mathcal{C}\}$. Correspondingly, we define $\mathcal{H}_\geq$. The range spaces $(\mathcal{D}, \mathcal{H}_\leq)$ and $(\mathcal{D}, \mathcal{H}_\geq)$ have constant VC dimension.

The proof of Lemma 4.4 is included in Appendix 4.6.

Lemma 4.5. Let $\mathcal{H}_l$ be the set of classifiers defined as

$$\mathcal{H}_l = \left\{h_1 \cap \cdots \cap h_l \mid h_1, \ldots, h_l \in (\mathcal{H}_\leq \cup \mathcal{H}_\geq)\right\} \ .$$

The range space $(\mathcal{D}, \mathcal{H}_l)$ has VC dimension $O(l \log l)$.

Proof. Observe that any finite subset $Y \subset \mathcal{D}$ shattered in the range space $(\mathcal{D}, \mathcal{H}_\leq \cup \mathcal{H}_\geq)$ can also be shattered in the range space $(\mathcal{D}, \mathcal{H}_\cup)$, where $\mathcal{H}_\cup = \{h_1 \cup h_2 \mid h_1 \in \mathcal{H}_\leq, h_2 \in \mathcal{H}_\geq\}$. Thus the VC-dimension of $(\mathcal{D}, \mathcal{H}_\leq \cup \mathcal{H}_\leq)$ is upper bounded by the VC-dimension of $(\mathcal{D}, \mathcal{H}_\cup)$. The proof now follows from Lemma 4.4 and Lemma 4.2. □

Observe that the separator tree H defines a set of regions such that each region is defined by the intersection of at most l classifiers in the set $(\mathcal{H}_\leq \cup \mathcal{H}_\geq)$ corresponding to the sphere separators in a path from the root to a leaf in H. Thus, a region can be defined by a classifier in $\mathcal{H}_l$. Furthermore, each region contains at most $c \cdot (|\Upsilon|/\hat{r})$ balls of Υ. It now follows from Lemma 4.5 and Lemma 4.3, that by sampling Υ with size at least $c_0 \cdot \hat{r} \log^2 \hat{r} \log\log \hat{r}$, Υ is a relative $(1/\hat{r}, \varepsilon)$-approximation of Γ in the range space $(\mathcal{D}, \mathcal{H}_l)$ with at least constant probability. The constant $c_0 > 0$ is chosen according to Lemma 4.5 and Lemma 4.3. Thus, with at least constant probability, the regions of Γ contains at most $O\big(\frac{|\Gamma|}{|\Upsilon|}c \cdot (|\Upsilon|/\hat{r})\big) = O(|\Gamma|/\hat{r})$ balls.

4.3.2 Bounding the Total Number of Boundary Balls

In the previous subsection, we bounded the size of each region. However, the number of regions may be large, since boundary balls occur in multiple regions. Recall that in a node i of the recursion we obtain a sphere separator h such that the number of intersected balls is $|\Upsilon_i(h_=)| \leq c_1\sqrt{k|\Upsilon_i|}$. We proceed to upper bound the total number of boundary balls by bounding the total number of intersected balls during the recursion. We show how to bound the total number of intersections in Υ by $O\big(\sqrt{k|\Upsilon|\hat{r}}\big)$.

In Section 4.3.1, we argued that the recursion on Υ produces regions of size at most $a = c\frac{|\Upsilon|}{\hat{r}}$, where $c > 0$ is a constant. Furthermore, it follows from (4.4) that the size of the smallest region is at least $(1/12) \cdot a$. Similar to Frederickson [53], we let $b(|\Upsilon|)$ denote the number of intersections of balls in Υ during the recursion. At each node i of the recursion, Υ_i is divided into two regions $\Upsilon_i(h_\leq)$ and $\Upsilon_i(h_\geq)$ by the sphere separator h. Recall that h is selected such that $\Upsilon_i(h_\leq) \leq \beta|\Upsilon_i|$, where $(1/12) \leq \beta \leq (11/12)$. Furthermore, it follows from (4.2) that $\Upsilon_i(h_\geq) \leq (1-\beta) \cdot |\Upsilon_i| + c_1\sqrt{k|\Upsilon_i|}$. We upper bound $b(|\Upsilon_i|)$ as follows:

$$b(|\Upsilon_i|) \leq \begin{cases} b\left(\beta|\Upsilon_i|\right) + b\left((1-\beta)\cdot|\Upsilon_i| + c_1\sqrt{k|\Upsilon_i|}\right) + c_1\sqrt{k|\Upsilon_i|} & \text{if } |\Upsilon_i| > a \\ 0 & \text{if } \frac{1}{12}\cdot a \leq |\Upsilon_i| \leq a \end{cases}$$

It can be shown by induction in the size of Υ_i that $b(|\Upsilon_i|) = O\big(\sqrt{k|\Upsilon_i|\hat{r}}\big)$. The proof of this is included in Appendix 4.7. Using the assumption $k \leq |\Upsilon|/\hat{r}$, it follows that the number of regions in Υ and Γ is $O\Big(\frac{|\Upsilon|+b(|\Upsilon|)}{(1/12)\cdot a}\Big) = O\Big(\frac{|\Upsilon|}{a}\Big) = O(\hat{r})$. Thus, the separator tree H can be used to divide Γ into $O(\hat{r})$ regions of size $O(|\Gamma|/\hat{r})$ by recursively applying the sphere separators of H to Γ

4.3.3 Reducing the Number of Boundary Balls in a Region

In order to obtain an r-way separator for Υ from an r-way division of Υ, we reduce the number of boundary balls in each region. Let b_i be the number of boundary balls in a region R_i of Υ. Letting $c_2 > 0$ be a constant, we describe how to ensure $b_i \leq c_2 \cdot \sqrt{k|\Upsilon|/\hat{r}}$ for each region R_i. We do this by adapting the technique described by Arge *et al.* [28, Section 3.3]. It follows from the previous subsection that $\sum b_i = O\big(\sqrt{k|\Upsilon|\hat{r}}\big)$. Let g denote the total number of regions which contain more than $c_2\sqrt{k|\Upsilon|/\hat{r}}$ boundary balls. For each region R_i where $b_i > c_2\sqrt{k|\Upsilon|/\hat{r}}$, we recursively apply Theorem 4.2 on the boundary balls of R_i. In other words, we recursively compute sphere separators h that divide R_i into regions $R_i(h_\leq)$ and $R_i(h_\geq)$ with at most $(10/12)\cdot b_i + c_3\sqrt{k|R_i|}$ boundary balls each, where $c_3 > 0$ is a constant. Recall that $|R_i| \leq c\frac{|\Upsilon|}{\hat{r}}$, where $c > 0$ is a constant. It follows that for sufficiently large c_2, the region is divided into two regions such that the number of boundary balls in each is $(11/12)\cdot b_i$. We recurse until the number of boundary balls is at most $c_2\sqrt{k|\Upsilon|/\hat{r}}$. Observe,

that the total number of sphere separators required to divide all g regions is

$$O\left(\sum \frac{b_i}{c_2\sqrt{k|\Upsilon|/\hat{r}}}\right) = O\left(\frac{\sqrt{k|\Upsilon|\hat{r}}}{c_2\sqrt{k|\Upsilon|/\hat{r}}}\right) = O(\hat{r}) \ .$$

Thus, the total number of regions will be increased by only $O(\hat{r})$. Furthermore, since each sphere separator adds $O(\sqrt{k|\Upsilon|/\hat{r}})$ new boundary balls, the total number of boundary balls remains $O(\sqrt{k|\Upsilon|\hat{r}})$. Each separator can be computed using expected $O(\text{Scan}(|\Upsilon|/\hat{r}))$ I/Os and, thus, we can reduce the number of boundary balls in each region by using an additional $O(\text{Scan}(|\Upsilon|))$ I/Os. Thus, the algorithm results in an $\hat{r}$-way separator for Υ.

4.3.4 Bounding the Total I/O-Complexity

We now bound the total I/O-complexity of the algorithm described. Let H be the separator tree computed during the $O(\log \hat{r})$ levels of the recursion. Recall, at each node i of the recursion we compute a sphere separator h using Theorem 4.2 using expected $\text{Scan}(|\Upsilon_i|)$ I/Os. Thus, the expected number of I/Os used in a level of recursion is $O\big(\text{Scan}(|\Upsilon|+b(|\Upsilon|))\big)$. Using the upper bound on $b(|\Gamma|)$ and the assumption $k \leq |\Upsilon|/\hat{r}$, we conclude that the total number of I/Os used during the $O(\log \hat{r})$ levels of recursion is $O\big(\text{Scan}(|\Upsilon|)\log \hat{r}\big)$. The results can now be stated in the following lemma:

Lemma 4.6. Given a k-ply neighborhood system Γ in the plane, a parameter $\hat{r} \leq M/B$, and a random sample Υ such that $|\Upsilon| \geq c_0 \cdot \hat{r}\log^2 \hat{r}\log\log \hat{r}$ and $k \leq |\Upsilon|/\hat{r}$, where c_0 is a constant. Then an $\hat{r}$-way separator H for Υ can be computed using expected $O\big(\text{Scan}(|\Upsilon|)\log \hat{r}\big)$ I/Os. Furthermore, with at least constant probability H produces an $\hat{r}$-way division of Γ when applied to Γ.

In order to compute the $\hat{r}$-way division of Γ, we apply Lemma 4.6 to obtain a separator tree H such that the sphere separators of H can be used to divide Γ into $O(\hat{r})$ regions of size $O(|\Gamma|/\hat{r})$ with at least constant probability. Note that since $\hat{r} = \lfloor M/B \rfloor$, the number of leaves in H is $O(M/B)$. We partition H into a constant number of subtrees such that each subtree $\hat{H}$ fits in memory along with one block per leaf of $\hat{H}$. We now divide Γ by recursing over the subtrees using $O(\text{Scan}(|\Gamma|))$ I/Os in total. We repeat the above an expected constant number of times until an $\hat{r}$-way division of Γ is obtained.

We proceed by bounding the number of I/Os used by Lemma 4.6 to $O(\text{Scan}(\Gamma))$. We use a sample of size $c_0\frac{M}{B}\log^2\frac{M}{B}\log\log\frac{M}{B}$ and assume $|\Gamma| > \frac{M}{B}\log^3\frac{M}{B}\log\log\frac{M}{B}$. Under this assumption, the expected number of I/Os is bounded as follows:

$$\begin{aligned}\text{Scan}(|\Upsilon|)\log\hat{r} &= O\left(\frac{\hat{r}\log^3\hat{r}\log\log\hat{r}}{B}\right) = O\left(\frac{M}{B^2}\log^3\frac{M}{B}\log\log\frac{M}{B}\right)\\ &= O\left(\frac{|\Gamma|}{B}\right) = O\big(\text{Scan}(|\Gamma|)\big) \ .\end{aligned}$$

Next, consider the case when $|\Gamma| \leq \frac{M}{B} \log^3 \frac{M}{B} \log\log \frac{M}{B}$. We observe that it is sufficient to divide Γ into regions of size $O(M)$, since regions of size $O(M)$ can be divided further by directly applying Theorem 4.2 on the regions. That is, each region of size $O(M)$ fits in memory after a constant number of applications of Theorem 4.2, so directly applying Theorem 4.2 uses $O(\text{Scan}(|\Gamma|))$ additional I/Os. Thus, using Lemma 4.6, we compute a separator tree H such that H can be used to divide Γ into $O(\bar{r})$ regions of size $O(M)$ with at least constant probability, where

$$\bar{r} = \frac{1}{B} \log^3 \frac{M}{B} \log\log \frac{M}{B} .$$

It follows that a sample $\bar{\Upsilon}$ of size $O(\text{polylog}(M/B)) = O(M)$ is sufficient. Similar to before, we repeat the sampling and separator computation until an $\bar{r}$-way division of Γ is found. This uses expected $O(\text{Scan}(|\Gamma|))$ I/Os since the problem size fits in internal memory after a constant number of levels of recursion.

Lemma 4.7. Given a k-ply neighborhood system Γ in the plane and a parameter $\hat{r} \leq M/B$ such that $k \leq \log^2 \frac{M}{B} \log\log \frac{M}{B}$, an $\hat{r}$-way division of Γ can be computed using expected $O\big(\text{Scan}(|\Gamma|)\big)$ I/Os.

We now present the final algorithm for computing an r-way division for Γ and general r. This algorithm follows immediately from Lemma 4.7. Let $\hat{r} = \min(r, \lfloor M/B \rfloor)$ and recursively apply Lemma 4.7 for $O(\log_{M/B}(r))$ levels of recursion. This uses a total of $O\big(\text{Scan}(|\Gamma|) \log_{M/B}(r)\big)$ I/Os.

Theorem 4.3. Given a k-ply neighborhood system Γ in the plane, an r-way division of Γ can be computed using expected $O\big(\text{Scan}(|\Gamma|) \log_{M/B}(r)\big)$ I/Os, assuming $k \leq \log^2 \frac{M}{B} \log\log \frac{M}{B}$.

Furthermore, this implies an algorithm for Koebe-embeddings, since Koebe-embeddings are 1-ply neighborhood systems.

Theorem 4.4. Given a planar graph $\mathcal{G}$ and a Koebe-embedding Γ of $\mathcal{G}$, an r-way division of $\mathcal{G}$ can be computed using expected $O\big(\text{Scan}(|\Gamma|) \log_{M/B}(r)\big)$ I/Os.

Note that the above results in an r-way division, but does not provide any bounds on the number of boundary balls when dividing Γ. In Appendix 4.9, we argue that the number of boundary balls in Γ can be bounded by increasing the sample size. The result is stated in the theorem below.

Theorem 4.5. Given a k-ply neighborhood system Γ in the plane, an r-way separator of Γ can be computed using expected $O\big(\text{Scan}(|\Gamma|) \log_{M/B}(r)\big)$ I/Os, assuming $k \leq |\Gamma|/r$ and

$$\log^3 \frac{M}{B} \log\log \frac{M}{B} \log \frac{|\Gamma|}{k} = O\Big(\sqrt{Mk}\Big) .$$

4.4 Applications to Delaunay Triangulations and Terrain

Motivated by applications in geographic information systems, we turn our attention to Delaunay triangulations. Delaunay triangulations are often used in practice to compute TINs from point clouds and can be computed I/O-efficiently using $O(\text{Sort}(N))$ I/Os [2]. In the previous section, we described how an r-way separator for a planar graph $\mathcal{G}$ can be computed I/O-efficiently, when given a Koebe-embedding of $\mathcal{G}$ and having certain assumptions on the size of the internal memory. However, to our knowledge, there are no known I/O-efficient algorithms for the computation of Koebe-embeddings. Therefore, we describe how to adapt our algorithm to compute separators for triangulations without computing a Koebe-embedding. This adaptation can also be applied to any triangulation in the plane and, thus, any planar graph since a planar graph $\mathcal{G}$ can be triangulated by trivially adding edges until every face of $\mathcal{G}$ is a triangle.

Observe that the circumcircles of a triangulation $\mathcal{G}$ in the plane form a k-ply neighborhood system Γ in the plane. However, k is $\Omega(|\mathcal{G}|)$ in the worst-case since many circumcircles may overlap in one point. Miller *et al.* [83] showed that the circumcircles of a Delaunay triangulation form a k-ply neighborhood with at most constant k when the largest ratio of the circum-radius to the length of the smallest edge over all triangles is at most constant. Therefore, we describe how to adapt our algorithm to circumcircles of a Delaunay triangulation. We compute an r-way division H for Γ and use H to divide $\mathcal{G}$ by mapping each region R_i of H to a region $\hat{R}_i$ as follows: Let the boundary vertices of $\hat{R}_i$ be the vertices that are contained in a triangle whose circumcircle is a boundary ball of R_i. The internal vertices of $\hat{R}_i$ are the vertices which are contained only in triangles whose circumcircles are internal balls of region R_i.

We see that the number of vertices in $\hat{R}_i$ is $O(|R_i|)$, where $|R_i|$ is the number of circumcircles of R_i. Furthermore, the number of boundary vertices in $\hat{R}_i$ regions is at most a constant factor larger than the number of boundary circumcircles of R_i.

It follows that we can also divide a TIN into regions by applying the above construction to the Delaunay triangulation used to construct the TIN. This approach allows for division-based algorithms on TINs. Haverkort *et al.* [62] showed that a separator of a grid graph can be used to I/O-efficiently compute the flow accumulation of a terrain. Their results can be applied to TINs and r-way divisions:

Lemma 4.8 (Division-Based Flow Accumulation [62])**.** Let Σ be a TIN and let $\mathbb{V}$ denote the vertices of the TIN. Given a rain distribution $\mathcal{R} : \mathbb{V} \to \mathbb{R}^+$ and an r-way division of Σ such that each region fits in internal memory along with

the rain distribution. Letting b_i denote the number of boundary vertices in a region R_i of the r-way division, the flow accumulation over Σ can be computed using $O(\text{Scan}(|\mathbb{V}|) + \text{Sort}(\sum b_i))$ I/Os

The flow accumulation over a TIN can also be computed using an algorithm that performs a top-down sweep of the terrain [62]. The result is stated in the following lemma:

Lemma 4.9 (Sweep-Based Flow Accumulation [62])**.** Let Σ be a TIN and let $\mathbb{V}$ denote the vertices of the TIN. Given a rain distribution $\mathcal{R} : \mathbb{V} \to \mathbb{R}^+$ that fits in internal memory, the flow accumulation of Σ can be computed using $O(\text{Sort}(|\mathbb{V}|))$ I/Os.

4.5 Experiments

In this section, we present the experiments we have conducted on real terrain data to demonstrate the efficiency of our algorithms. We have implemented and tested the r-way division algorithm for Delaunay triangulations as described in Section 4.4 and Theorem 4.3. Furthermore, we have implemented and tested the two flow accumulation algorithms stated in Lemma 4.8 and Lemma 4.9. The algorithms have been implemented in C++ and make heavy use of the TPIE library [19] which provides implementations of fundamental I/O-efficient algorithms such as sorting and priority queues. The experiments were run on an Intel i7-3770 CPU with 32 GB of RAM and 28 TB of storage running Linux. Each program was assigned 25 GB of memory during testing.

For our tests, we used the Danish Elevation Model published by the Danish Agency for Data Supply and Efficiency [50]. The model consists of a highly detailed point cloud collected using LiDAR technology. Each point in the model is labeled as ground, vegetation, a building rooftop, and others. There is an average of 4.5 points per square meter, however, this varies for each area of the terrain since non-reflective surfaces, such as certain types of vegetation, can result in large holes in the point cloud. In this paper, we focus on modeling the flow of water and, thus, we filtered out points not labeled as ground or building points. We constructed a TIN from the resulting point cloud using a Delaunay triangulation.

We have implemented Theorem 4.2 to compute sphere separators as described by Miller *et al.* [84] and Clarkson *et al.* [43]. Whenever we sample sphere separators on input $\mathcal{G}$, we discard a separator h if it does not satisfy that $|\mathcal{G}(h_{\leq})| \geq 1/4 \cdot |\mathcal{G}|$ and $|\mathcal{G}(h_{\geq})| \geq 1/4 \cdot |\mathcal{G}|$, where $|\mathcal{G}|$ is the number of triangles in $\mathcal{G}$, $|\mathcal{G}(h_{\leq})|$ is the number of triangles inside or intersecting h, and $|\mathcal{G}(h_{\geq})|$ is the number of triangles outside or intersecting h. We note that this differs from previous sections, where we computed the separator based on the circumcircles. However, since the number of intersected circumcircles upper bounds the number of intersected triangles, the previous proofs still apply to

this adaptation. Furthermore, this simplifies the algorithm by avoiding the computation of circumcircles. In order to examine the size of $|\mathcal{G}(h_=)|$, we sampled a large number of sphere separators on various areas of the terrain and computed the number of intersected triangles. The results are shown in Appendix 4.8 and demonstrate that $|\mathcal{G}(h_=)|$ is small in practice.

Next, we assessed our r-way division algorithm (Theorem 4.3) and the flow accumulation algorithms described in Lemma 4.8 and Lemma 4.9. We evaluated our implementation on a TIN representing a $3500\,\mathrm{km}^2$ area around Herning, Denmark. The TIN is represented as a list of triangles, such that each vertex of a triangle is annotated with its coordinates, an index, and the flow direction of the vertex. The flow directions have been computed by selecting the neighboring vertex with minimum height. This resulted in a TIN with 23.3 billion points and a total size of 6.2 TB. In order to measure the I/O throughput of the system, we implemented a simple program that reads the entire TIN and observed that the program took 2 hours and 56 minutes to run on the TIN.

We computed a multiway division on the input such that our implementation of division-based flow accumulation (Lemma 4.8) can fit each region in memory. The division was saved to disk by representing each region as a file containing a list of triangles. Each triangle is annotated with the same information as the input as well as a boolean variable indicating whether the triangle is a boundary triangle or not. This resulted in a division with 131 regions and only 9 394 816 boundary vertices in total. Furthermore, $\sum b_i = 17\,257\,717$, where b_i is the number of boundary vertices in region R_i. Thus, we see that the number of boundary vertices is very small in practice despite having no theoretical bounds for this algorithm. The computation of this multiway division took 19 hours and 23 minutes. We implemented a program that reads the output division and writes a dummy division with the same number of regions and the same number of triangles in each region. This program took 3 hours and 16 minutes to read the division and 8 hours and 12 minutes to write the dummy division. In other words, we see that a large proportion of the running time is due to computation in internal memory.

Having computed the division, we applied it to compute the flow accumulation over the TIN using the division-based algorithm (Lemma 4.8). In our implementation, we used the uniform rain distribution that distributes one unit of water on each vertex. In this setup, our implementation of division-based flow accumulation took 13 hours and 59 minutes when given the division as input. We compared this to an implementation of Lemma 4.9 which took 20 hours and 18 minutes when running on the same input TIN. Thus, we see a significant improvement in running time when the terrain has been preprocessed by computing the multiway division. In other words, our approach improves performance when computing flow accumulation for different rain distributions on the same TIN. However, the division-based approach is slower when computing the flow accumulation for only a single rain distribution, and

the time spent preprocessing is included.

4.6 Appendix: Proof of Constant VC dimension

In this section, we present the proof of Lemma 4.4.

Proof. We begin by proving that the VC dimension of $(\mathcal{D}, \mathcal{H}_{\geq})$ is constant. Let $h_{\geq} \in \mathcal{H}_{\geq}$ denote a classifier corresponding to the separator sphere h with center (x_h, y_h) and radius r_h. Let $d \in \mathcal{D}$ be a disk in the plane with center (x_d, y_d) and radius r_d. We can express $d \in \mathcal{D}(h_{\geq})$ as follows:

$$\begin{aligned} d \in \mathcal{D}(h_{\geq}) &\Leftrightarrow \sqrt{(x_h - x_d)^2 + (y_h - y_d)^2} \geq r_h - r_d \\ &\Leftrightarrow r_d \geq r_h \vee (x_h - x_d)^2 + (y_h - y_d)^2 \geq (r_h - r_d)^2 \ . \end{aligned}$$

We now map d to $\mathbb{R}^7$ by mapping the monomials $x_d^i y_d^j r_d^k$ to variables z_i as follows:

$$\langle z_1, z_2, z_3, z_4, z_5, z_6, z_7 \rangle = \langle x_d, y_d, x_d y_d, x_d^2, y_d^2, r_d, r_d^2 \rangle \ .$$

Assume there is a finite subset $Y \subset \mathcal{D}$ that is shattered by $\mathcal{H}_{\geq}$. It follows that Y maps to a subset $\hat{Y} \subset \mathbb{R}^7$ where $|Y| = |\hat{Y}|$ and $\hat{Y}$ is shattered by $\mathcal{H}_{\cup} = \{h_1 \cup h_2 \mid h_1, h_2 \in \mathcal{H}_{halfspace}\}$, where $\mathcal{H}_{halfspace}$ is the set of all halfspaces in $\mathbb{R}^7$. Thus, the VC dimension of $(\mathcal{D}, \mathcal{H}_{\geq})$ is at most the VC dimension of $(\mathbb{R}^7, \mathcal{H}_{\cup})$. It follows from Lemma 4.1 and Lemma 4.2 that the VC dimension of $(\mathcal{D}, \mathcal{H}_{\geq})$ is constant. The VC dimension of $(\mathcal{D}, \mathcal{H}_{\leq})$ can be bounded correspondingly. □

4.7 Appendix: Bound on the Total Number of Boundary Balls

In this section, we provide an upper bound for the function $b(|\Upsilon_i|)$ introduced in Section 4.3.2. Recall the definition of $b(|\Upsilon_i|)$:

$$b(|\Upsilon_i|) \leq \begin{cases} b\left(\beta|\Upsilon_i|\right) + b\left((1-\beta)\cdot|\Upsilon_i| + c_1\sqrt{k|\Upsilon_i|}\right) + c_1\sqrt{k|\Upsilon_i|} & \text{if } |\Upsilon_i| > a \\ 0 & \text{if } \frac{1}{12}\cdot a \leq |\Upsilon_i| \leq a \end{cases}$$

We proceed to show that $b(|\Upsilon_i|) \leq \frac{c_4|\Upsilon_i|\sqrt{k}}{\sqrt{a}} - c_5\sqrt{k|\Upsilon_i|}$, where $c_5 = 5\cdot(c_1 + 1/100)$ and $c_4 = c_5 \cdot \sqrt{12}$. We prove this by induction in the size of Υ_i. Letting $|\Upsilon_i| \geq \frac{1}{12}\cdot a$, it follows that $\frac{c_4|\Upsilon_i|\sqrt{k}}{\sqrt{a}} - c_5\sqrt{k|\Upsilon_i|} \geq 0$ since $c_5 \leq c_4\sqrt{1/12}$. This proves the base case. We proceed with proving the induction step using the

induction hypothesis:

$$\begin{aligned}
b(|\Upsilon_i|) &\leq b\Big(\beta|\Upsilon_i|\Big) + b\Big((1-\beta)\cdot|\Upsilon_i| + c_1\sqrt{k|\Upsilon_i|}\Big) + c_1\sqrt{k|\Upsilon_i|} \\
&\leq \frac{c_4\left(|\Upsilon_i| + c_1\sqrt{k|\Upsilon_i|}\right)\sqrt{k}}{\sqrt{a}} - c_5\sqrt{|\Upsilon_i|k\beta} - c_5\sqrt{|\Upsilon_i|k(1-\beta)} + c_1\sqrt{k|\Upsilon_i|} \\
&\leq \frac{c_4|\Upsilon_i|\sqrt{k}}{\sqrt{a}} - \sqrt{k|\Upsilon_i|}\left(c_5\sqrt{\beta} + c_5\sqrt{1-\beta} - c_1 - \frac{c_4c_1\sqrt{k}}{\sqrt{a}}\right) \\
&\leq \frac{c_4|\Upsilon_i|\sqrt{k}}{\sqrt{a}} - c_5\sqrt{k|\Upsilon_i|}\left(\sqrt{\beta} + \sqrt{1-\beta} - \frac{c_1}{c_5} - \frac{c_4c_1\sqrt{k}}{c_5\sqrt{a}}\right) .
\end{aligned}$$

Recall the assumption that $k \leq |\Upsilon|/\hat{r}$ and that $a = c\frac{|\Upsilon|}{\hat{r}}$. It follows that $k/a \leq 1/c$. Furthermore, since $(1/12) \leq \beta \leq (11/12)$ it follows that $\sqrt{\beta} + \sqrt{1-\beta} \geq \frac{6}{5}$. We now rewrite as follows:

$$\begin{aligned}
b(|\Upsilon_i|) &\leq \frac{c_4|\Upsilon_i|\sqrt{k}}{\sqrt{a}} - c_5\sqrt{k|\Upsilon_i|}\left(\frac{6}{5} - \frac{c_1}{c_5} - \frac{c_4c_1\sqrt{k/a}}{c_5}\right) \\
&\leq \frac{c_4|\Upsilon_i|\sqrt{k}}{\sqrt{a}} - c_5\sqrt{k|\Upsilon_i|}\left(\frac{6}{5} - \frac{c_1 + c_4c_1\sqrt{1/c}}{c_5}\right) .
\end{aligned}$$

Recall that the recursion produces regions of size at most $a = c\frac{|\Upsilon|}{\hat{r}}$, where $c > 0$ is a constant. By inserting $c_5 = 5\cdot(c_1 + 1/100)$ and setting c sufficiently large such that $c_4c_1\sqrt{1/c} \leq 1/100$, we obtain

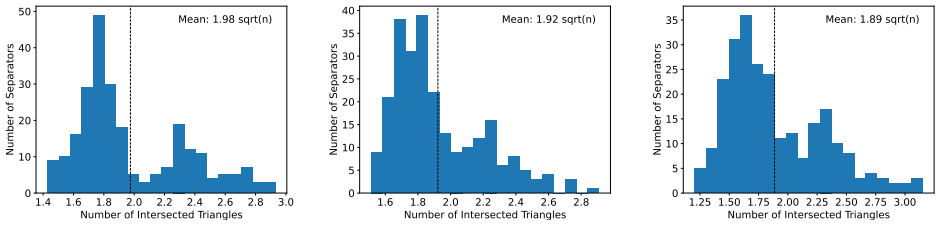
$$\begin{aligned}
b(|\Upsilon_i|) &\leq \frac{c_4|\Upsilon_i|\sqrt{k}}{\sqrt{a}} - c_5\sqrt{k|\Upsilon_i|}\left(\frac{6}{5} - \frac{c_1 + \frac{1}{100}}{5\cdot(c_1 + \frac{1}{100})}\right) \\
&= \frac{c_4|\Upsilon_i|\sqrt{k}}{\sqrt{a}} - c_5\sqrt{k|\Upsilon_i|} .
\end{aligned}$$

This completes the proof by induction. Thus, the total number of intersections when recursing Υ is at most $b(|\Upsilon_i|) \leq \frac{c_4|\Upsilon_i|\sqrt{k}}{\sqrt{a}} = O\big(\sqrt{k|\Upsilon_i|\hat{r}}\big)$.

4.8 Appendix: Experimental Evaluation of Separator Size

In this section, we examine the number of intersections when computing sphere separators on a terrain represented as a TIN. As previously described in Section 4.5, we have implemented Theorem 4.2 as described by Miller *et al.* [84] and Clarkson *et al.* [43]. We apply the implementation to various areas of a TIN constructed from the Danish Elevation model including only ground and building points. Let $\mathcal{G}$ denote the input TIN and let $|\mathcal{G}|$ denote the number of

triangles in $\mathcal{G}$. For each area we sample 250 sphere separators such that each sphere separator h satisfies $|\mathcal{G}(h_{\leq})| \geq 1/4 \cdot |\mathcal{G}|$ and $|\mathcal{G}(h_{\geq})| \geq 1/4 \cdot |\mathcal{G}|$, where $|\mathcal{G}|$ is the number of triangles in $\mathcal{G}$, $|\mathcal{G}(h_{\leq})|$ is the number of triangles inside or intersecting h, and $|\mathcal{G}(h_{\geq})|$ is the number of triangles outside or intersecting h.



(a) Copenhagen (City) (b) Fjeldstervang (Fields) (c) Gammel Rye (Forest)

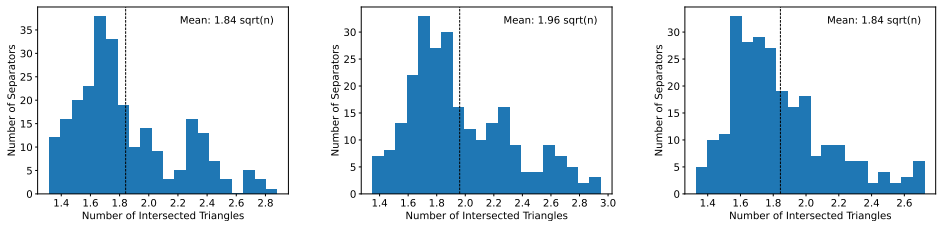
Figure 4.1: Evaluating the number of intersected triangles on various types of terrain. Each area is 10 km by 10 km

	Copenhagen (city)		Fjeldstervang (fields)		Gammel Rye (forest)	
Total points	1 737 352 542		1 016 877 728		1 380 507 284	
Ground	1 003 915 809	(57.78%)	726 748 491	(71.47 %)	514 504 787	(37.27 %)
Building	193 699 578	(11.15%)	6 462 467	(0.64 %)	8 164 875	(0.59 %)
Low veg.	17 072 550	(0.98%)	28 417 823	(2.79 %)	23 066 640	(1.67 %)
Medium veg.	93 116 564	(5.35%)	41 787 026	(4.11 %)	66 254 550	(4.80 %)
High veg.	379 489 187	(21.84%)	210 080 703	(20.66 %)	735 216 819	(53.26 %)
Water	3 619 895	(0.21%)	1 746 674	(0.17 %)	29 697 070	(2.15 %)

Table 4.1: Description of point cloud labels for tiles of size 10 km by 10 km. The most frequent point labels have been included.

We first examine how the separator size changes when the type of terrain changes. For this, we examine three different 10 km by 10 km tiles of the terrain. The first tile is an area from the city of Copenhagen which has a high frequency of building points. The second tile is an area centered on the town of Gammel Rye which has a high frequency of points labeled as high vegetation. Finally, the third tile is centered on the town of Fjeldstervang which has a high frequency of ground points describing agricultural fields. We have described the distribution of point labels in Table 4.1. For each tile, we have plotted a histogram describing the number of intersections of sphere separators. This is shown in Figure 4.1. We compute the mean number of intersections for each area and normalize the result by dividing by the square root of the number of points. We observe that the mean number of intersections is between $1.89\sqrt{n}$ and $1.98\sqrt{n}$ for the tiles tested. Thus, in the examined areas, the number of intersections is on average $c\sqrt{n}$ for a small constant c.

Next, we examine how the number of intersections changes as the total



(a) A 100 km^2 tile. (b) A 400 km^2 tile. (c) A 900 km^2 tile.

Figure 4.2: Evaluating the number of intersected triangles when increasing total area.

	100 km^2 Tile		400 km^2 Tile		900 km^2 Tile	
Total points / 100 km^2	11 841 409		10 088 500		10 120 894	
Ground / 100 km^2	6 733 499	(56.86%)	6 713 922	(66.55%)	6 681 457	(66.02%)
Building / 100 km^2	114 970	(0.97%)	202 454	(2.01%)	124 327	(1.23%)
Low veg. / 100 km^2	293 570	(2.48%)	188 550	(1.87%)	204 808	(2.02%)
Medium veg. / 100 km^2	547 759	(4.63%)	428 343	(4.25%)	418 139	(4.13%)
High veg. / 100 km^2	4 103 293	(34.65%)	2 493 932	(24.72%)	2 645 132	(26.14%)
Water / 100 km^2	19 731	(0.17%)	30 524	(0.30%)	25 666	(0.25%)

Table 4.2: Description of point cloud labels for increasing tile sizes. The most frequent point labels have been included.

area increases. For this, we have conducted an experiment where we create 3 tiles of increasing sizes near the city of Herning. When creating the tiles, we attempted to keep the number of ground and building points per 100 km^2 constant. However, it is very difficult to keep them constant in practice due to the nature of the data. Therefore, we expect that this will affect the number of intersections to some extend. The distribution of point labels is described in Table 4.2. The number of intersections for each tile is shown in Figure 4.2. As expected, we see a varying number of intersections between tiles. However, we note that the normalized number of intersections remains small for all the tested tile sizes.

4.9 Appendix: Algorithm with Larger Sample

In this section, we modify the algorithm described in Section 4.3 by increasing the size of the sample Υ and choosing sphere separators h in a slightly different way. By doing this, we can relax the assumption on k to $k \leq |\Gamma|/\hat{r}$ and provide bounds on the number of boundary balls in Γ.

Define the set of classifiers $\mathcal{H}_=$ as $\mathcal{H}_= = \{\mathcal{D}(h_=) \mid h \in \mathcal{C}\}$. It follows from Lemma 4.2 and Lemma 4.4 that the range space $(\mathcal{D}, \mathcal{H}_=)$ have constant VC dimension. Furthermore, we modify the definition of $\mathcal{H}_l$ to include $\mathcal{H}_=$ as

follows:

$$\mathcal{H}_l = \left\{ h_1 \cap \cdots \cap h_l \mid h_1, \ldots, h_l \in (\mathcal{H}_{\leq} \cup \mathcal{H}_{\geq} \cup \mathcal{H}_{=}) \right\} .$$

Using the same argument as the proof of Lemma 4.5, we see that the VC-dimension of $(\mathcal{D}, \mathcal{H}_l)$ is $O(l \log l)$.

Recall that the algorithm described in Section 4.3.1 forms a separator tree H by recursively computing sphere separators h on Υ. The proof of the algorithm relied on the assumption that Υ is sampled such that Υ is a $(1/\hat{r}, \varepsilon)$-relative approximation of Γ, where $\varepsilon > 0$ is a constant. This allowed us to argue about the size of the regions of Γ within an error of $O(|\Gamma|/\hat{r})$. However, we now wish to argue that the number of boundary balls in a region is $O(\sqrt{k|\Gamma|/\hat{r}})$, which can be much smaller than $|\Gamma|/\hat{r}$. Therefore, we let $p = \left(\sqrt{k|\Gamma|/\hat{r}}\right)/|\Gamma| = \sqrt{k/(|\Gamma|\hat{r})}$. Assume that Υ is a (p, ε)-relative approximation of Γ in the range space $(\mathcal{D}, \mathcal{H}_l)$. Furthermore, we relax the assumption on k to $k \leq |\Gamma|/\hat{r}$.

We modify the recursion of Section 4.3.1 as follows: Recall that we divide Υ into regions of size at most $c \cdot |\Upsilon|/\hat{r}$. Let $\Upsilon_i \subseteq \Upsilon$ and $\Gamma_i \subseteq \Gamma$ denote the balls in a node i of the recursion. We use Theorem 4.2 to obtain a sphere separator h that with probability at least $1/2$ satisfies

$$|\Upsilon_i(h_<)|, |\Upsilon_i(h_>)| \leq \frac{10}{12} \cdot |\Upsilon_i| , \tag{4.5}$$

$$|\Gamma_i(h_=)| \leq c_1 \sqrt{k|\Gamma_i|} , \tag{4.6}$$

where $c_1 > 0$ is a constant. Note that this uses expected $\mathrm{Scan}(|\Upsilon_i|)$ I/Os. We use (4.6) and the assumption that Υ is a (p, ε)-relative approximation to obtain the following:

$$\begin{aligned} |\Upsilon_i(h_=)| &\leq (1+\varepsilon)\frac{|\Upsilon|}{|\Gamma|} \cdot c_1 \sqrt{k|\Gamma_i|} \\ &\leq (1+\varepsilon)\frac{|\Upsilon|}{|\Gamma|} \cdot c_1 \sqrt{(1+\varepsilon)\frac{|\Gamma|}{|\Upsilon|} \cdot k|\Upsilon_i|} \\ &= c_2 \sqrt{\frac{|\Upsilon|}{|\Gamma|} \cdot k|\Upsilon_i|} , \end{aligned} \tag{4.7}$$

where $c_2 > 0$ is a constant. Similar to before, we sample an expected constant number of separators h, until (4.5) and (4.7) are satisfied. Note that the assumption $k \leq |\Gamma|/\hat{r}$ implies $k\frac{|\Upsilon|}{|\Gamma|} \leq \frac{|\Upsilon|}{\hat{r}}$. Recalling that $|\Upsilon_i| \geq a = c\frac{|\Upsilon|}{\hat{r}}$, we rewrite as follows:

$$|\Upsilon_i(h_=)| \leq c_2 \sqrt{\frac{|\Upsilon|}{\hat{r}}|\Upsilon_i|} \leq c_2 \sqrt{\frac{|\Upsilon_i|^2}{c}} \leq \frac{c_2}{\sqrt{c}}|\Upsilon_i| . \tag{4.8}$$

Thus, for sufficiently large $c > 0$, (4.5) and (4.8) implies $|\Upsilon_i(h_{\leq})| \leq \frac{11}{12}|\Upsilon_i|$ and $|\Upsilon_i(h_{\geq})| \leq \frac{11}{12}|\Upsilon_i|$. In other words, the problem size becomes smaller by a

constant factor and we can recursively compute separators h until Υ is divided into regions of size at most $c \cdot |\Upsilon|/\hat{r}$.

Recall that in Section 4.3.1, we argued that $|\Upsilon_i(h_=)| = O\big(\sqrt{k|\Upsilon_i|}\big)$. It follows that the above modification results in the number of intersections of Υ_i being smaller by a factor $\Theta\big(\sqrt{|\Upsilon|/|\Gamma|}\big)$. By inserting this into the proofs described in Section 4.3.2 and Section 4.3.3, we obtain that Υ is divided into $O(\hat{r})$ regions of size $O(|\Upsilon|/\hat{r})$ such that each region has $O\left(\sqrt{\frac{|\Upsilon|}{|\Gamma|} \cdot k|\Upsilon|/\hat{r}}\right)$ boundary balls. Furthermore, since Υ is a (p,ε)-relative approximation of Γ, we see that Γ is divided into $O(\hat{r})$ regions of size $O(|\Gamma|/\hat{r})$ such that each region has $O\big(\sqrt{k|\Gamma|/\hat{r}}\big)$ boundary balls. It follows that the total number of boundary balls $O\big(\sqrt{k|\Gamma|\hat{r}}\big)$.

In the description above, we assumed that that Υ is a (p,ε)-relative approximation of Γ. However, if this is not the case, it might not be possible to obtain a classifier satisfying (4.7). To ensure that the algorithm terminates in this case, we let $m = O(\log \hat{r})$ and sample at most m sphere separators in each node of the recursion. We terminate if no separator satisfying (4.5) and (4.7) is found. Recall that in the case where Υ is (p,ε)-relative approximation, a sampled sphere separator h satisfies (4.5) and (4.7) with probability at least $1/2$. Thus, for a fixed node we fail to obtain a separator h with probability 2^{-m} since the randomness in each sphere separator is independent. Since the number of nodes in the separator tree is $O(\hat{r})$, at least one node in the tree will fail with probability at most $O(\hat{r}) \cdot 2^{-m}$. Thus, all nodes succeed with at least constant probability by setting $m = c_4 \cdot \log \hat{r}$ for sufficiently large $c_4 > 0$.

We have now shown that, when given a (p,ε)-relative approximation of Γ, we can compute a separator tree H that results in an $\hat{r}$-way separator for Γ when applied to Υ. Given such a separator tree H, we can compute an $\hat{r}$-way separator of Γ by recursively dividing the balls of Γ using H as described in Section 4.3.4. Note that this uses $O(\mathrm{Scan}(|\Gamma|))$ I/Os. We now bound the expected total number of I/Os used when computing H. Following the same line of the argumentation as in Section 4.3, the expected number of I/Os used in a level of the recursion is $O(\mathrm{Scan}(|\Upsilon|)\log \hat{r})$. Note the additional log-factor that results from the case where Υ is not a (p,ε)-relative approximation of Γ. Thus, the expected number of I/Os used in all $O(\log \hat{r})$ levels of the recursion is $O(\mathrm{Scan}(|\Upsilon|)\log^2 \hat{r})$. It follows from Lemma 4.3, that we obtain a (p,ε)-relative approximation Υ of Γ with at least constant probability by sampling Υ of size

$$|\Upsilon| = O\left(\sqrt{\frac{|\Gamma|\hat{r}}{k}} \log \hat{r} \log\log \hat{r} \log \sqrt{\frac{|\Gamma|\hat{r}}{k}}\right) = O\left(\sqrt{\frac{|\Gamma|\hat{r}}{k}} \log \hat{r} \log\log \hat{r} \log \frac{|\Gamma|}{k}\right)$$

Observe that this is significantly larger than the sample size described in Section 4.3.

Note that we can assume $\hat{r} \leq |\Gamma|/M$, since this will be sufficient to divide Γ into regions of size $O(M)$. This follows from the observation that regions of size $O(M)$ fit in memory after a constant number of applications of Theorem 4.2.

Thus, they can be further divided using $O(\text{Scan}(|\Gamma|))$ I/Os in total. We now obtain the following:

$$\begin{aligned}\text{Scan}(|\Upsilon|)\log^2\hat{r} &= O\left(\frac{\sqrt{|\Gamma|}}{B}\sqrt{\frac{\hat{r}}{k}}\log^3\hat{r}\log\log\hat{r}\log\frac{|\Gamma|}{k}\right)\\ &= O\left(\frac{\sqrt{|\Gamma|}}{B}\sqrt{\frac{|\Gamma|}{Mk}}\log^3\hat{r}\log\log\hat{r}\log\frac{|\Gamma|}{k}\right)\\ &= O\left(\frac{|\Gamma|}{B}\cdot\frac{\log^3\hat{r}\log\log\hat{r}\log\frac{|\Gamma|}{k}}{\sqrt{Mk}}\right).\end{aligned}$$

Recall that we wish to bound the expected total number of I/Os to $O(\text{Scan}(|\Gamma|))$ and that $\hat{r} \leq M/B$. We now obtain the following lemma:

Lemma 4.10. Given a k-ply neighborhood system Γ in the plane and a parameter $\hat{r} \leq M/B$, an $\hat{r}$-way separator of Γ can be computed using expected $O\big(\text{Scan}(|\Gamma|)\big)$ I/Os, assuming $k \leq |\Gamma|/\hat{r}$ and

$$\log^3\hat{r}\log\log\hat{r}\log\frac{|\Gamma|}{k} = O\Big(\sqrt{Mk}\Big).$$

Given a parameter $r > 0$, it follows that we can compute an r-way separator by recursively applying Lemma 4.10 for $O(\log_{M/B}(r))$ levels. This proves Theorem 4.5.

Chapter 5

1D and 2D Flow Routing on a Terrain

Abstract

An important problem in terrain analysis is modeling how water flows across a terrain creating floods by forming channels and filling depressions. In this paper we study a number of *flow-query* related problems: given a terrain Σ, represented as a triangulated xy-monotone surface with n vertices, and a rain distribution R which may vary over time, determine how much water is flowing over a given vertex or edge as a function of time. We develop internal-memory as well as I/O-efficient algorithms for flow queries. This paper contains four main algorithmic results:

(i) An internal-memory algorithm for answering *terrain-flow* queries: preprocess Σ into a linear-size data structure so that given a rain distribution R, the flow-rate functions of all vertices and edges of Σ can be reported quickly.

(ii) I/O-efficient algorithms for answering terrain-flow queries.

(iii) An internal-memory algorithm for answering *vertex-flow* queries: preprocess Σ into a linear-size data structure so that given a rain distribution R, the flow-rate function of a vertex under the single-flow direction (SFD) model can be computed quickly.

(iv) An efficient algorithm that given a path P in Σ and flow rate along P, computes the two-dimensional channel along which water flows.

Additionally we implement a version of the terrain-flow query and 2D channel algorithms, and examine a number of queries on real terrains.

5.1 Introduction

Tribute to Lars Arge: The last three authors of the paper dedicate this paper to the memory of our friend and colleague Lars Arge who passed away on December 23, 2020. A giant in the field of applied algorithms and data structures, he worked extensively on external-memory algorithms for terrain

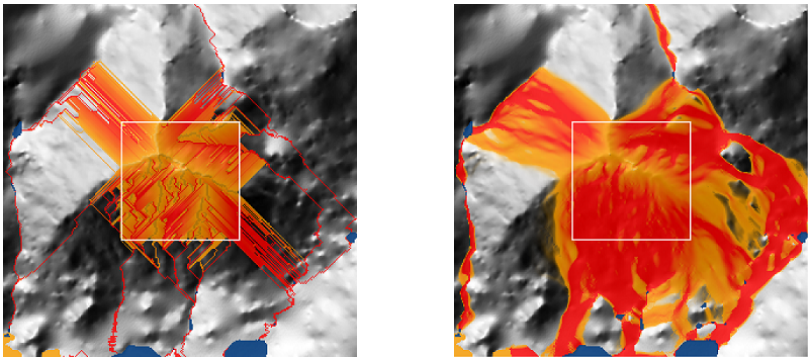
modeling and analysis and made several seminal contributions to this area. This paper is on a topic, namely hydrological analysis on terrains, that was dear to Lars and that was deeply influenced by his work.

An important problem in terrain analysis is modeling how water flows across a terrain and creates floods by forming channels and filling up depressions. The rate at which a depression fills up during a rainfall depends not only on the shape of the depression and the size of its watershed (i.e., the area of the terrain that contributes water to the depression) but also on other depressions that are filling up. Water falling on the watershed of a filled depression flows to a neighboring depression, effectively making the watershed of the latter larger and filling it up faster. Modeling how depressions fill and how water spills into other depressions during a flash flood event is therefore an important computational problem.

Besides determining which areas of a terrain become flooded and when they become flooded, determining the 2D channels (rivers) along which water flows across the terrain is also important. The flow queries we study can also be used to answer related flood-risk queries. We assume we are given a terrain Σ, represented as a triangulated xy-monotone surface with n vertices. As in earlier papers, [22, 77, 89] we assume that water flows along the edges of Σ. Two models of water flow along edges have been proposed: (i) a simple and more widely used model called the single flow-direction (SFD) model in which water from a vertex flows along *one* of its downward edges, and (ii) a more accurate but more complex model called the multiflow-direction (MFD) model in which water at a vertex splits and flows along all of its downward edges. See Figure 5.1. We consider both of these models and study the following three problems in this paper:

- *Terrain-flow query*: given a rain distribution (possibly varying with time), compute as a function of time the flow rate (of water) for all vertices and edges of Σ.
- *Vertex-flow query*: given a rain distribution and a query vertex q of Σ, compute the flow rate of q under the single flow-direction (SFD) model.
- *2D flow network*: Given a 1D flow network, represented as a set of edges along with their flow values, compute 2D channels along which water flows.

Finally, as high-resolution terrain data sets are becoming easily available, their size easily exceeds the size of main memory of a standard computer, so movement of data between main memory and external memory (such as disk) becomes the bottleneck in computations. We use the *I/O-model* with one disk by Aggarwal and Vitter [6]: the computer is equipped with a two-level memory hierarchy consisting of an *internal memory*, which is capable of holding m data items, and an *external memory* of unlimited size. All computation



(a) SFD (b) MFD

Figure 5.1: Rain falls in the marked square according to (a) single-flow direction (SFD) and (b) multiflow direction (MFD) models.

happens on data in internal memory. Data is transferred between internal and external memory in blocks of b consecutive data items. Such a transfer is referred to as an *I/O-operation* or an *I/O*. The cost of an algorithm is the number of I/Os it performs. The number of I/Os required to read n items from disk is $\mathrm{Scan}(n) = O(n/b)$. The number of I/Os required to sort n items is $\mathrm{Sort}(n) = \Theta\big((n/b)\log_{m/b}(n/b)\big)$ [6]. For all realistic values of n, m, and b we have $\mathrm{Scan}(n) < \mathrm{Sort}(n) \ll n$.

Related work. Due to its importance, the problem of modeling how water flows on a terrain has been studied extensively, and many approaches have been taken to address this problem. One approach focuses on accurately simulating fluid dynamics using non-linear partial differential equations such as the Navier-Stokes equations. These equations have no closed-form solutions and are usually solved using numerical methods. They often account for additional factors, such as the effects of different terrain types and drainage networks. While these models tend to be more accurate, they are computationally expensive and do not scale to large terrains. Bates and De Roo [33] developed one such model for simulating flooding on digital elevation models (DEMs) using two flow models for different regions of the terrain: the first handles flow within rivers and the second models flow of water as it spreads over floodplains. While there has been some research into refining the representation of channels, such as Wood *et al.* [100], often the channel geometry is assumed to be a simple model (e.g. rectangular or trapezoidal.)

Water-flow modeling on a terrain also has been studied in the GIS community. These approaches use simpler models, which are computationally efficient and suitable for large datasets. Although some early work, e.g. [75], allowed water to flow in the interior of faces, recent work assumes that water flows along the edges of the terrain. This assumption is a good approximation for

high-resolution data sets, where the size of input triangles is small. The SFD and MFD models described above are used to model the water flow locally at a vertex, see e.g. [22, 77, 78, 89].

Arge et al. [24] described an I/O-efficient algorithm for the *flow-accumulation* problem in the SFD model, which asks how much water flows over each point in a terrain assuming the terrain has only one sink at infinity and water falls uniformly on the terrain. Their algorithm performs a total of $O(\mathrm{Sort}(n))$ I/Os. The flow-accumulation model only provides a rough solution to flow modeling, since it assumes that either the terrain does not have any local minima or that they have been filled in advance.

In order to accurately model flow, it is necessary to compute times at which depressions fill and simulate how water spills from one depression into others. Arge et al. [22] proposed the first I/O-efficient algorithm that computes the fill times of all maximal depressions in $O(\mathrm{Sort}(\rho)\log(\rho/m) + \mathrm{Sort}(n))$ I/Os, where ρ is the number of depressions in the terrain and m is the size of the internal memory. If $\rho = O(m)$, the algorithm can be simplified and requires only $O(\mathrm{Sort}(n))$ I/Os.

Arge et al. [17] described an I/O-efficient algorithm that can answer *terrain-flood* queries: Given a rain distribution R, determine which points on the terrain become flooded if a total volume ψ of rain falls. The algorithm answers queries using $O(\mathrm{Sort}(n) + \mathrm{Scan}(\chi\rho))$ I/Os, where χ is the height of the merge tree of the terrain. Assuming $\chi < m$, the number of I/Os can be further bounded to $O(\mathrm{Sort}(n))$. Furthermore, assuming $\rho < m$, the algorithm can be modified to answer a query in $O(\mathrm{Scan}(n))$ I/Os after $O(\mathrm{Sort}(n))$ preprocessing.

Lowe and Agarwal [77] presented efficient algorithms for several flood queries on a terrain under the multiflow-direction (MFD) model. They presented an $O(n \log n)$-time algorithm to answer terrain-flood queries. They also showed that a terrain Σ can be preprocessed in $O(n \log n + n\rho)$-time into a data structure that can answer *point-flood* queries: given a rain distribution R, a volume of rain ψ, and a point $q \in \Sigma$, determine whether q will be flooded. The query time is $O(|R|b_q + b_q^2)$ time, where b_q is the number of maximal depressions that contain the query point q; $b_q = \Omega(n)$ in the worst case. Finally, they presented an algorithm to determine when a query point q gets flooded. To our knowledge, no I/O-efficient algorithms are known for these flooding queries under the MFD model.

Our results. Unlike [77], which focused on water accumulation or flooding, this paper focuses on computing how water flows along each edge of a terrain as a function of time. It contains four main algorithmic results.

(i) We present (in Section 5.3) an $O(n \log n)$-time algorithm for preprocessing Σ into a linear-size data structure for answering terrain-flow queries: for a rain distribution R, it can compute the flow rate of all vertices in $O(|\phi| \log |\phi|)$ time, where $|\phi|$ is the total complexity of nonzero flow-rate functions. In the worst case $|\phi| = \Theta(n(\chi + k))$, where χ, as above, is the height of the merge

tree of Σ, and k is the number of times the rain distribution changes. However $|\phi|$ is much smaller in practice. An immediate corollary of our result is that a flood-time query (i.e. given R and a point $q \in \Sigma$, when does q become flooded) can be answered in the same time, which is a significant improvement over the result in [77].

(ii) We present (in Section 5.4) two I/O-efficient algorithms for terrain-flow queries. We first preprocess Σ using $O(\text{Sort}(n))$ I/Os and $O(n \log n)$ internal computation time. The first algorithm assumes $\rho(\chi + k) = O(m)$, where χ, k and ρ are as above, and answers a terrain-flow query in $O(\text{Scan}(n) + \text{Sort}(|\phi|))$ I/Os and $O((\chi + k)(n + \rho \log \rho) + |\phi| \log |\phi|)$ internal computation time. The second algorithm assumes $\rho = O(m)$ and answers a terrain-flow query in $O(\text{Sort}((\chi+k)n \log n))$ I/Os and $O((\chi+k)n \log n \log(kn))$ internal computation time. We additionally note that since the terrain-flow query is more general than the terrain-flood query, these algorithms also yield I/O-efficient algorithms for the terrain-flood and flood-time queries under the MFD model studied in [77].

(iii) Under the SFD model, we can build, in $O(n \log n)$ time, a linear-size data structure that given a vertex q and rain distribution R computes ϕ_q in $O(|R| + |\mathcal{B}|k \log n)$ time, where $|R|$ is the complexity of the rain distribution, $|\mathcal{B}|$ is the number of tributaries of q in which rain is falling, and k is the number of times the rain distribution changes. (See Section 5.5.)

(iv) We present an algorithm that given a 1D flow, represented as a path along the edges of Σ and the flow values on these edges, computes, in $O(|\mathcal{C}| \log |\mathcal{C}|)$ time, a 2D channel $\mathcal{C}$ along which water flows, where $|\mathcal{C}|$ is the number of "wetted" faces at least partially covered by the water in the channel (Section 5.6.) We do so using Manning's equation [81], a widely used empirical formula relating flow-rate of water in an open channel to the geometry of the channel. When computing the 2D flow network we assume a real RAM model of computation in which the zeros of a polynomial can be computed in $O(1)$ time; see [5]. We note that previous work computing the 2D channel assumes the cross section of each channel to have a simple geometry (e.g. rectangular or trapezoidal) [33]. In contrast, we do not make any such assumption.

Finally, we have implemented our terrain-flow-query and 2D channel algorithms, and we report a number of empirical results on real terrains in Section 5.7 to demonstrate the efficacy and efficiency of our algorithms.

We note that this paper is an expanded version of a paper first appearing in the ACM SIGSPATIAL conference [79]. Here we give a new, simpler internal memory algorithm for terrain-flow queries, which is also faster in some cases. We also expand upon the 2D channel algorithm and give methods for relaxing certain assumptions made in [79]. Finally, in this version we have implemented the terrain-flow and 2D channel algorithms and included a number of experimental results.

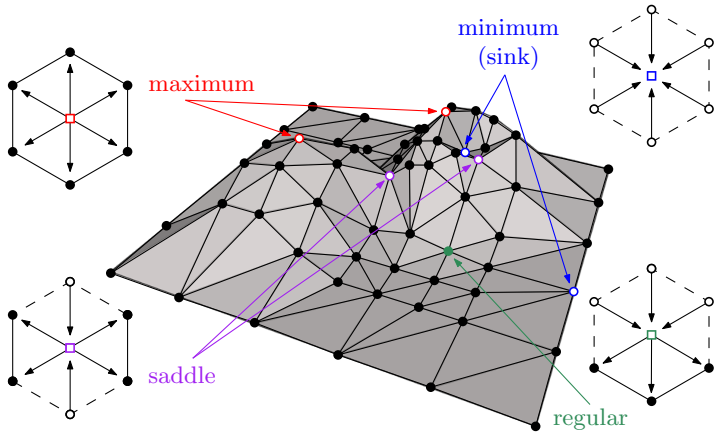


Figure 5.2: An example terrain with its critical vertices marked with colored hollow circles, and regular vertices marked with filled circles.

5.2 Preliminaries & Models

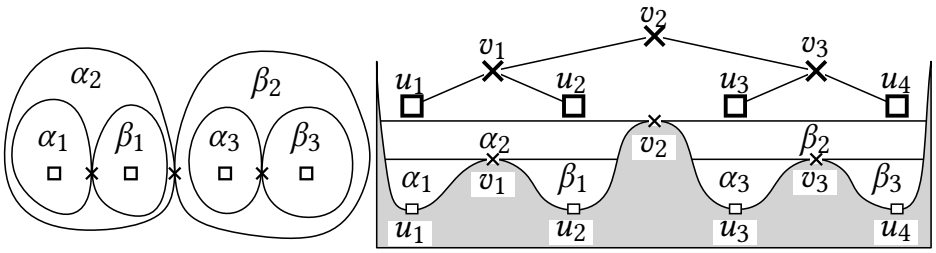
In this section, we give a number of preliminary definitions and describe the flooding model. Most of the text here closely follows [77, 89].

5.2.1 Geometric preliminaries

Terrains. Let $\mathbb{M}$ be a triangulation of $\mathbb{R}^2$, and let $\mathbb{V}$ be the set of vertices of $\mathbb{M}$; set $n = |\mathbb{V}|$. We assume that $\mathbb{V}$ contains a vertex v_∞ at infinity and that each edge (u, v_∞) is a ray emanating from u; the triangles in $\mathbb{M}$ incident to v_∞ are unbounded. Let $h : \mathbb{M} \to \mathbb{R}$ be a height function. We assume that the restriction of h to each triangle of $\mathbb{M}$ is a linear map, that h approaches $+\infty$ at v_∞, and that the heights of all vertices are distinct. Given $\mathbb{M}$ and h, the graph of h, called a *terrain* and denoted by $\Sigma = (\mathbb{M}, h)$, is an xy-monotone triangulated surface whose triangulation is induced by $\mathbb{M}$.

Critical vertices. There is a natural cyclic order on the neighbor vertices of a vertex v of $\mathbb{M}$, and each such vertex is either an *upslope* or *downslope* neighbor. If v has no downslope (resp. upslope) neighbor, then v is a *minimum* (resp. *maximum*). We also refer to a minimum as a *sink*. If v has four neighbors w_1, w_2, w_3, w_4 in clockwise order such that $\max(h(w_1), h(w_3)) < h(v) < \min(h(w_2), h(w_4))$ then v is a *saddle* vertex. See Figure 5.2.

Level sets, contours, depressions. Given $\ell \in \mathbb{R}$, the *ℓ-sublevel set of h* is the set $h_{<\ell} = \{x \in \mathbb{R}^2 \mid h(x) < \ell\}$, and the *$\ell$-level set of h* is the set $h_{=\ell} = \{x \in \mathbb{R}^2 \mid h(x) = \ell\}$. Each connected component of $h_{<\ell}$ is called a *depression*, and each connected component of $h_{=\ell}$ is called a *contour*. Note that the boundary of a depression is not necessarily simply connected, as a saddle may cause a hole to appear in a depression.



(a) (b)

Figure 5.3: An example terrain with saddle vertices v_1-v_3. Each saddle v_i delimits two maximal depressions α_i and β_i. (a) Terrain seen from above. Sinks are marked with a square and saddles are marked with a cross. (b) Terrain seen from the side.

For a point $x \in \mathbb{M}$, a depression β_x of $h_{<\ell}$ is said to be *delimited by the point* x if x lies on the boundary of β, which implies that $h(x) = \ell$. A depression β is *maximal* if every depression containing β contains (strictly) more sinks than β. A maximal depression that contains exactly one sink is called an *elementary depression*. Each maximal depression is delimited by a saddle, and a saddle that delimits more than one maximal depression is called a *negative saddle*. For a maximal depression β, let $\mathrm{Sd}(\beta)$ denote the saddle delimiting β, and let $\mathrm{Sk}(\beta)$ denote the set of sinks in β. The *volume* of a depression β of $h_{<\ell}$ is

$$\mathrm{Vol}(\beta) = \int_{\beta} (\ell - h(v))dv. \tag{5.1}$$

Merge tree. The maximal depressions of a terrain form a hierarchy that is easily represented using a rooted tree called the *merge tree* [39, 69] and denoted by T. Suppose we sweep a horizontal plane from $-\infty$ to ∞. As we vary ℓ, the depressions in $h_{<\ell}$ vary continuously, but their structure changes only at sinks and negative saddles. If we increase ℓ, then a new depression appears at a sink, and two depressions merge at a negative saddle. The merge tree T of Σ is a tree that tracks these changes. Its leaves are the sinks of Σ, and its internal nodes are the negative saddles of Σ. The edges of T are in one-to-one correspondence with the maximal depressions of Σ, that is, we associate each edge $e = (u, v)$ with the maximal depression β_e delimited by u and containing v. We regard each edge $e = (u, v)$ of T with a 1D line segment $(h(u), h(v))$ then the point ξ at height ℓ corresponds to a depression D_ξ of $h_{<\ell}$ that contains the maximal depression delimited by v, and all points of ∂D_ξ contract to ξ. Hence, each point of $\mathbb{M}$ can be mapped to a point of T. See Figure 5.3 for an example. We assume that T has an edge from the root of T extending to $+\infty$, corresponding to the depression that extends to ∞. For simplicity, we assume

that T is binary, that is, each negative saddle delimits exactly two depressions. Non-simple saddles can be unfolded into a number of simple saddles [49].

Let u be a negative saddle, let (u, v_1) and (u, v_2) be two down edges in T from u, and let (w, u) be the up edge from u. We call the depression associated with (u, v_2) (resp. with (w, u)) as the *sibling* (resp. *parent*) (depression) of that associated with (u, v_1).

T can be computed in $O(n \log n)$ time [39], and it can be preprocessed in $O(n)$ additional time so that for a point $x \in \mathbb{R}^2$, $\mathrm{Vol}(\beta_x)$, the volume of the depression delimited by x can be computed in $O(\log n)$ time [39]. In the I/O model, T and the volumes of all maximal depressions can be computed using $O(\mathrm{Sort}(n))$ I/Os [21]. This algorithm can be extended to compute $\mathrm{Vol}(\beta_x)$ and the smallest maximal depression containing x for all vertices $x \in \Sigma$ [22].

A refinement of T in which we map each vertex v of $\mathbb{M}$ to an edge of T and store the sequence of vertices mapped to each edge in increasing order of their heights is called the *extended merge tree*. It can be computed in the same time as T. We will be mostly working with the extended merge tree. With a slight abuse of notation, we use T to denote both merge and extended merge trees of Σ.

5.2.2 Flooding model

We now describe the flooding model, which is the same as in [77], and define flow-rate functions.

Flow graph and flow functions. We transform $\mathbb{M}$ into a directed acyclic graph $\mathcal{M}$, referred to as the *flow graph*, by directing each edge (u, v) of $\mathbb{M}$ from u to v if $h(u) > h(v)$, and from v to u otherwise, i.e., each edge is directed in the downward direction. We say a vertex v is *upstream* (resp. *downstream*) of w if there is a path of edges with non-zero flow from v to w (resp. from w to v.) For each (directed) edge (u, v), we define the *local flow* $\lambda(u, v, t)$ to be the portion of water arriving at u that flows along the edge (u, v) to v at time t. By definition, for any $u \in \mathbb{V}$, $\sum_{(u,v)\in\mathbb{M}} \lambda(u, v, t) = 1$.

The value of $\lambda(u, v, t)$ is, in general, based on relative heights of the downslope neighbors of u. If u is not a negative saddle vertex, then $\lambda(u, v, t)$ remains the same for all t, so we will often drop t and write $\lambda(u, v)$ to denote $\lambda(u, v, t)$. If u is a negative saddle, then $\lambda(u, v)$ changes when one of the depressions delimited by u fills up as no water flows from u to that depression; see below for further discussion.

Rain distribution. Let $R(v, t) : \mathbb{V} \times \mathbb{R} \to \mathbb{R}_{\geq 0}$ denote a *rain distribution*, that is, for each vertex $v \in \mathbb{V}$, $R(v, t)$ indicates the rate at which rain is falling on v at time t. We assume that for each v, $R(v, \cdot)$ is a piecewise-constant function of time, with the function changing at discrete time values $\{t_0 = 0, t_1, \ldots, t_k\}$, and for all v and $t \geq t_k$, $R(v, t) = 0$. For a depression β,

we define $R(\beta, t) = \sum_{v \in \beta} R(v, t)$. For $i \leq k$, let $|R_i|$ denote the number of vertices for which $R(v, t_i) \geq 0$, and let $|R| = \sum_{i=1}^{k} |R_i|$. In practice $|R_i| \ll n$.

Fill and spill rates. For a maximal depression β, we define the *fill rate* $F_\beta : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ as the rate at which water is arriving in the depression β as a function of time. That is, the rate at which rain is falling directly in β plus the rate at which other depressions are spilling water into β. Similarly, we define the *spill rate* $S_\beta : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ as the rate (as a function of time) at which water spills from β through the saddle that delimits β. If the rain rate is piecewise constant, then so are fill and spill rates.

Flow rate. Next, we define *flow rates* ϕ_e and ϕ_v for edges e and vertices v of $\mathbb{M}$, which is the amount of water flowing through e and v, respectively, at time t. For an edge $(u, v) \in \mathbb{M}$, $\phi_{(u,v)}(t)$ is the fraction of water from u that passes along (u, v) as a function of time. That is,

$$\phi_{(u,v)}(t) = \lambda(u, v, t)\phi_u(t). \tag{5.2}$$

The flow rate ϕ_v of a non-saddle vertex v is the sum of the flow rates along incoming edges to v plus the rain on v. That is,

$$\phi_v(t) = R(v, t) + \sum_{(u,v) \in \mathbb{M}} \phi_{(u,v)}(t). \tag{5.3}$$

Letting τ_v be the time at which a vertex v becomes flooded, $\phi_v(t)$ and $\phi_{(u,v)}(t)$ for any $v \in \mathbb{M}$ are undefined for $t \geq \tau_v$. That is to say, when a vertex is flooded, the flow-rate function is undefined.

Let v be a negative saddle delimiting two depressions α and β. Until one of α or β is filled, ϕ_v is defined using (5.3). Without loss of generality, assume that depression α fills first, say at time τ_α, and water starts spilling from α to β through v. The spill rate S_α specifies the rate at which water spills from α to β. It is tempting to simply add S_α to (5.3) to define the flow rate of v for $t > \tau_\alpha$, but it double counts the amount of water that was flowing from v to depression α. For $t < \tau_\alpha$, ϕ_v is defined as in (5.3). For $t \geq \tau_\alpha$, ϕ_v is defined as,

$$\phi_v(t) = \left(R(v, t) + \sum_{(u,v) \in \mathbb{M}} \phi_{(u,v)}(t) \right) \sum_{w \in \beta} \lambda(v, w, 0) + S_\alpha(t). \tag{5.4}$$

Finally for $t \geq \tau_\alpha$ and for any $w \in \beta$,

$$\lambda(v, w, t) = \frac{\lambda(v, w, 0)}{\sum_{z \in \beta} \lambda(v, z, 0)}. \tag{5.5}$$

The model can be extended in a straightforward manner if v delimits more than two depressions. Since we assume the rain distribution to be piecewise constant, flow rates are also piecewise linear.

We conclude this resection by remarking that we denote a piecewise-constant function f as a sequence $(\delta_1, t_1), (\delta_2, t_2), \cdots$, where $0 < t_1 < t_2 < \cdots$ with the interpretation that $f(0) = 0$ and $f(t) = f(t_i) + \delta_i$ for $t \in (t_i, t_{i+1}]$.

5.3 Terrain-flow Query

In this section, we describe an output-sensitive internal-memory algorithm that, given a terrain Σ and rain distribution $R = (\mathbb{M}, h)$, computes the flow-rate functions $\phi_v(t)$ and $\phi_e(t)$ for all vertices and edges $v, e \in \mathbb{M}$. Note that $\phi_{(u,v)}$ for an edge is completely determined by ϕ_u using (5.2), so we focus on computing the flow-rate functions of vertices. The flow rates of edges can be computed at the end using (5.2).

The overall algorithm is a sweep-line algorithm, each of whose step performs another sweep. At the top level, it sweeps along the time axis. At time t, it has computed the flow rates of all vertices and edges for the time interval $[0, t]$. The flow rates change at discrete time instances, which we refer to as *events*. (There will be additional events, see below.) The algorithm stops at each event and sweeps the terrain Σ in the $(-z)$-direction to update the flow rates of relevant vertices and edges. We first describe the top-level sweep, then describe how each event is processed, and finally analyze the running time of the algorithm.

5.3.1 The top-level sweep

We call a vertex v *dry* if its flow rate is 0 for all t, and *wet* otherwise. We call v flooded if the depression delimited by v is under water. Initially no vertex is flooded, but as rain water accumulates, some vertices start getting flooded. Both dry and wet vertices may get flooded. The algorithm maintains the set $\mathcal{W}$ of wet vertices and their flow rates. Recall that once a vertex is flooded, its flow rate is undefined.

We call a maximal depression β *active* if its fill rate is positive, all of its children depressions are flooded, and either β or one of its sibling depressions is not flooded. The algorithm maintains the set $\mathcal{D}$ of active depressions. Recall each maximal depression is associated with an edge e of T. We store a bit b_e with each edge e of T and set it to 1 if the corresponding depression β_e is active and 0 otherwise. We also store a bit b_v with each node v of T, which is set to 1 if v is flooded. For each $\beta \in \mathcal{D}$, it maintains:

1. the current fill and spill rates of β, denoted by F_β and S_β, respectively,
2. the set $\mathcal{W}_\beta$ of wet vertices in β,
3. the water level in β; this information is maintained in a lazy manner as described below.

The flow rates of vertices and the set of active depressions change when the rain distribution changes or a depression becomes full. We refer to the former as a *rain event* and the latter as a *depression event*. More precisely,

- *Rain event.* A rain event at time t_ξ is defined by a triple (t_ξ, X, δ), where $X \subset \mathbb{V}$ is the set of vertices at which rain fall changes at time t_ξ, and $\delta : X \to \mathbb{R}$ describes the change in the rain at the vertex $x \in X$.

- *Depression event.* A depression event at time t_ξ is specified by a pair (t_ξ, β) where β is a maximal depression that gets filled at time t_ξ.

The algorithm processes these events in order, updates the flow rates, the set of active depressions, and the set of wet vertices at each event; and detects and updates future events. The algorithm maintains the following two data structures to perform the sweep efficiently:

- *Time-event priority queue* $\mathcal{Q}_T$: It stores the events detected so far. The time of an event is used as its key. The data structure supports the following four operations:
  - DeleteMin(): Delete and return the event with the smallest key.
  - Insert(ξ)/Delete(ξ): insert or delete an event ξ.
  - Decrease key(ξ, t): Decrease the event time of an event ξ from its current value t_ξ to t; it assumes that $t_\xi \geq t$.
- *Union-find data structure* $\mathcal{K}$: It maintains the set of wet vertices in each active depression and supports the following three operations:
  - Create(x, β): add a new vertex x to an active depression β.
  - Union($\beta_x, \beta_y, \beta_z$): merge two active depressions β_x and β_y and refer to the new depression as β_z (which may be the same as β_x or β_y.)
  - Find(x): return the active depression that contains the wet vertex x; if x does not lie in any active depression then return the maximal depression that contains x.

The algorithm maintains the invariant *that each event ξ with event-time t_ξ is stored in the priority queue $\mathcal{Q}_T$ immediately before t_ξ with the correct key t_ξ.* That is, if a depression β becomes full at time τ_β, we will have the event (τ_β, β) in the event queue by the time we have processed the last event with time before τ_β

Initially, $\phi_v(0) = 0$ for all $v \in \mathbb{V}$, $\mathcal{D} = \emptyset$, and $\mathcal{W} = \emptyset$. We initialize $\mathcal{Q}_T$ with all rain events, which are given as part of the input. At each step, the algorithm extracts the next event from $\mathcal{Q}_T$ using the DeleteMin procedure. It processes each event as described below:

Handling a depression event. Suppose an active depression β has become full at time τ_β. Let u be the saddle delimiting β. If all of the sibling depressions of β are full, β and all of its sibling depressions become inactive and the parent depression β' of β becomes active. We reset the bits of β and its sibling bits to 0, and the bit of β' to 1. We set the bit of u to 1 to indicate u is now flooded. Next, we merge the sets $\mathcal{W}_\xi$ where ξ is β or its sibling depression into a single set and name it $\mathcal{W}_{\beta'}$ (using the union-find data structure $\mathcal{K}$.) We also add the wet vertices that lie in β' but not in any of its children depressions (i.e.

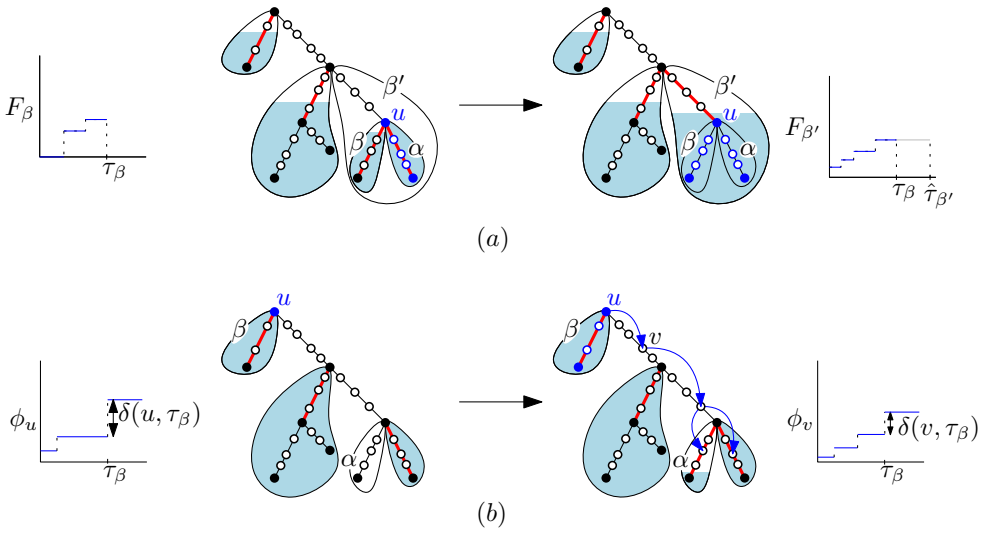


Figure 5.4: The two types of depression events when a depression β, delimited by u, becomes full at time τ_β, with α (resp. β') being the sibling (resp. parent) depression of β. Shown immediately before and after processing; edges corresponding to active depressions are marked bold in red and the water level in active depressions shaded blue. (a) If α is already full, α and β become inactive and β' becomes active, with $F_{\beta'} = F_\beta$ at τ_β. (b) If α is not yet full, we increase the flow rate at the saddle u accordingly; β remains active. Flow-rates of some downstream vertices of u also increase and at time τ_β, and some empty depressions may receive water, becoming active at τ_β (e.g. α).

the vertices that map to the edge of T corresponding to β') to $\mathcal{W}_{\beta'}$. We set $F_{\beta'} = F_\beta$ and $S_{\beta'} = 0$. Based on the current fill rate $F_{\beta'}$ we estimate the time $\hat{\tau}_{\beta'}$ when β' will become full and add the depression event $(\hat{\tau}_{\beta'}, \beta')$ to $\mathcal{Q}_T$.

Next, consider the case when one of the sibling depressions of β is not full. In this case the water from β spills to its sibling depressions through the saddle u. We set $S_\beta = F_\beta$. We increase the flow rate ϕ_u at time t_e as described in (5.4) and update the local-flow rate of downslope neighbors of u. Let δ_u be the increase in the flow rate at u. Since this affects the flow rate at vertices reachable from u in the flow graph, we update their flow rates by calling the second-level sweep algorithm described in the next subsection, which sweeps Σ downward starting from u.

Handling a rain event. Let X be the set of vertices where rainfall has changed at time t_0, and let $\delta : X \to \mathbb{R}$ be the change in the rate. We use the second-level sweep algorithm to update the flow functions and the set of active depressions.

5.3.2 Second-level sweep

We are given a set $X \subset \mathbb{V}$ of vertices and the change in rain on X by $\delta : X \to \mathbb{R}$ at time t_{curr}. The goal is to update the flow rate at all the affected vertices. The algorithm sweeps in the $(-z)$-direction and while at $z = z_0$ maintains a set of pairs (x, δ_x), where $x \in \mathbb{V}$ and $\delta_x \in \mathbb{R}$, such that x is not flooded, $h(x) < z_0$, and the change in flow rate at x through direct rainfall or through the incoming edges whose upslope neighbors have height at least z_0 is δ_x. These pairs are stored in a vertex-event priority queue $\mathcal{Q}_V$, with heights of the vertices as the key. It supports the following operations:

- DeleteMax(): Delete and return the pair with the maximum key value.
- Insert(x)/Delete(x): Insert or delete a pair (x, δ_x).
- UpdateFlow(x, γ): If (x, δ_x) is already stored in $\mathcal{Q}_V$, update it to $(x, \delta_x + \gamma)$. If x is not in $\mathcal{Q}_V$ then insert (x, γ) into $\mathcal{Q}_V$.

This algorithm relies on the following two procedures:

- IsFlooded(x): Given a vertex x, determine whether x is flooded: using the extended merge tree T, we retrieve the edge $e = (u, v) \in \mathsf{T}$, with $h(u) > h(v)$, that contains x. if $b_u = 1$ then x is flooded, and if $b_v = 0$ then x is not flooded, so assume that $b_u = 0$ and $b_v = 1$, in which case the depression β_e is active. Let (h_e, τ_e) be the previous recording of water level at e. Using the current fill rate F_{β_e} and scanning the vertices of β_e starting from height h_e, we compute the new water level h_e at time t_e and update the recording of water level at e. If $h_e \geq h(x)$, then x is flooded otherwise it is not flooded.

- FloodedVertex(x, δ): Given a vertex $x \in \mathbb{M}$ that is already flooded, process the δ change in water reaching x as follows. using Find(x), it computes the active depression β that contains x. It sets the fill rate $F_\beta = F_\beta + \delta$. If β is not flooded (i.e. $b_\beta = 0$) then we recompute the time τ_β when β will become full, based on the new fill rate— this requires computing the current water level in β (as in the above procedure) and computing the volume of β that is not full (using the information stored in extended merge tree). If $\delta > 0$, then the value of the fill time of β has decreased and we use the DecreaseKey(β, τ_β) to update the estimated time of the depression event for β in the priority queue $\mathcal{Q}_T$. If $\delta < 0$, we delete β from $\mathcal{Q}_T$ and insert it with the new key τ_β. In this case, the procedure does not return any vertex.

  If β is already flooded then we update its spill rate $S_\beta = S_\beta + \delta$. Let u be the saddle delimiting β. We call the procedure UpdateFlow(u, δ) and add δ to the flow rate of u.

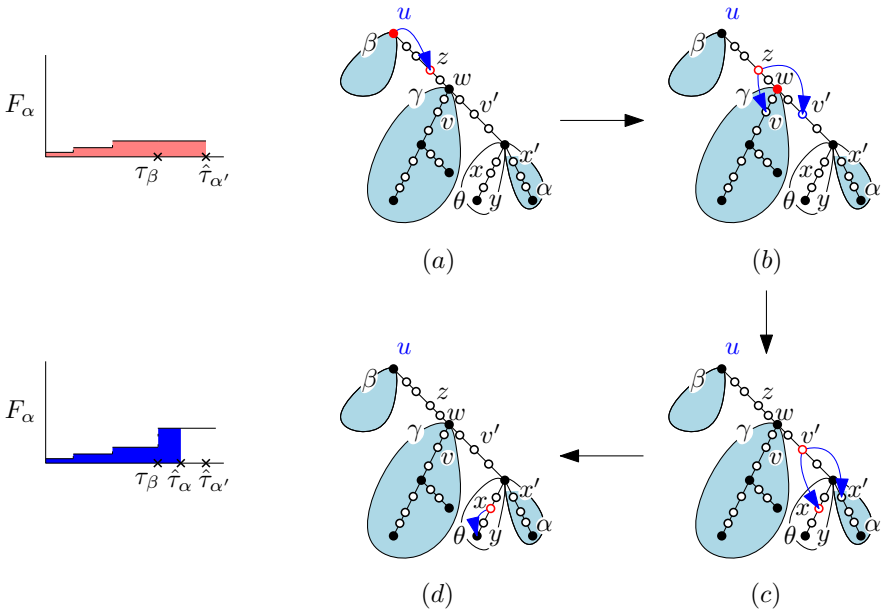


Figure 5.5: Several steps of a second-level sweep. T is shown with the water level of each depression at τ_β marked in blue. To the left we show F_α immediately before and after the second level sweep is performed. As we proceed, the vertices in $\mathcal{Q}_V$ at each step are marked in red. (a) First, β becomes full and spills from the saddle u; the downslope neighbor z is not flooded, its flow is updated and is added it to the priority queue. (b) Updating downslope neighbor of z: a downslope neighbor, v, is flooded. so F_γ (the fill rate of γ) is updated accordingly, and saddle w is added to $\mathcal{Q}_V$; downslopoe neighbor v' is updated and added to $\mathcal{Q}_V$. (c) As we continue the sweep, we find x' is flooded and contained in the active depression α. As α is not yet full, when we update the fill rate F_α, we also compute the new estimated time $\hat{\tau}_\alpha$ when α becomes full, and update the corresponding depression event in $\mathcal{Q}_T$. (d) When we look at the downslope neighbor of x, we find y is a minimum and not yet flooded. We label y as flooded, the depression θ containing it becomes active, and we add a corresponding event to $\mathcal{Q}_T$.

With these two procedures at hand, we are now ready to describe the second-level sweep. We initialize $\mathcal{Q}_V$ as follows. Given $X, \delta \to \mathbb{R}$, for each $x \in X$, we first call IsFlooded(x) to determine whether x is flooded. If the answer is no, we insert $(x, \delta(x))$ into $\mathcal{Q}_V$. Otherwise we call FloodedVertex($x, \delta(x)$).

We repeat the following steps until $\mathcal{Q}_V$ is empty (see Figure 5.5):

1. Retrieve the highest vertex from $\mathcal{Q}_V$ using DeleteMax. Let (v, δ_v) be the pair.

2. We add the pair $(t_{\text{curr}}, \delta_v)$ to ϕ_v.

3. For each downslope neighbor u of v with $\lambda(u, v, t_{curr}) > 0$, we do the following:

   a) We call the procedure IsFlooded(v).

   b) If v is not flooded, we call UpdateFlow($v, \lambda(u, v, t_{curr}), \delta_v$) and add the pair to $\mathcal{Q}_V$ (e.g. Figure 5.5(b) and (c).)

      i. If v is a minimum, we label v flooded by setting $b_v = 1$ in T and add the depression containing v to $\mathcal{D}$ (e.g. vertex y in Figure 5.5(d).)

   c) If u is flooded, we call the procedure FloodedVertex($v, \lambda(u, v, t_{curr}), \delta_v$) (e.g. vertices v and x in Figure 5.5(c) and (d).)

This completes the description of the second-level sweep. We make a few remarks:

Remarks. (i) During a single run of the second-level sweep, the fill time of a depression may be updated multiple times because of different flooded vertices. Instead of updating $\mathcal{Q}_T$ each time, we can maintain a buffer that stores all active depressions whose spill times are being updated. Every time FloodedVertex procedure updates the fill time of a depression β, we update it in the buffer. When the second-level sweep terminates for the current event, we scan the buffer and update $\mathcal{Q}_T$. Although this modification does not improve the worst-case running time of the algorithm, it improves the running time in practice when $\mathcal{Q}_T$ is large.

(ii) We have assumed that no two vertices have the same height but this assumption does not hold for many real data sets, especially for urban areas. If the input has a flat region we handle it as follows: we treat the plateau as a single node in the merge tree. Water which reaches the plateau will flow to all downslope neighbors of any vertex in the plateau.

(iii) We assume that each maximal depression is delimited by a unique saddle, but some data sets have degeneracies and a maximal depression may be determined by multiple saddles. In this case, the water spills through all these saddles if the neighboring depressions are not full. A user can specify weight to each such saddle to describe what proportion of water spills through each of them. For simplicity, we set the weight of all such saddles to be the same.

(iv) The real data sets contain many tiny depressions, which lead to flow rates being "noisy"— the flow rate has many breakpoints with very small increase in the flow rate. We can reduce the noise by hydrologically conditioning the terrain in the preprocessing step, e.g. pre-flooding tiny depressions; see Section 5.7 for a discussion of hydrologically conditioning methods.

5.3.3 Implementation of data structures

Priority queues. The time-event priority queue is implemented as a Fibonacci heap [54]. It performs DecreaseKey and Insert operations in constant time, and DeleteMin operations in $O(\log n)$ time where n is the number of elements in the heap.

There are two possible implementations for the vertex-event priority queue $\mathcal{Q}_V$. The first is a Fibonacci heap. The second implementation uses the fact that the heights of the vertices are known ahead of time. We can associate each vertex with an integer value in the range $[1, n]$ corresponding to its position in the sorted list of all vertices. Thorup [96] proposed a priority queue to store a set of integers in the range $[1, n]$, which requires $O(\log \log n)$ time for each DeleteMin operation, along with constant time Insert and DecreaseKey operations. If there are $|V_\tau|$ vertices with non-zero change in flow-rate at time τ, each DeleteMin operation while using the Fibonacci heap takes $O(\log |V_\tau|)$ time. While $|V_\tau|$ is not known a priori, we can use the Fibonacci heap until $|V_\tau| = \log \log n$, and at that point build the integer priority queue and use it for the rest of the sweep.

Union-find data structure. We now describe how to build and maintain the data structure $\mathcal{K}$, so that if a vertex is flooded, we can determine the largest maximal depression which is flooded and contains it (i.e. the active depression.) Each maximal depression will store an associated set of wet vertices $\mathcal{W}_\beta$. When we call Create(x, β), we add x to the set $\mathcal{W}_\beta$.

To support Union and Find, we use a union-find data structure on the maximal depressions. Each maximal depression starts as its own set. When a depression β becomes active, we take the union of the set corresponding to β and the sets of its children. Finally update the representative of this union to the saddle delimiting β. To find the maximal flooded depression containing a flooded query vertex v, we first find the maximal depression β corresponding to the edge of T that contains v. Then Find(β) returns the desired depression. While the general case of union-find takes super-linear time, Gabow and Tarjan [56] show that if the sets correspond to nodes in a tree and we restrict the operations to unions of nodes with their parents in the tree, union find operations can be performed in amortized constant time. We can see that the operations we perform fall under this restriction by associating each maximal depression β with the highest vertex in the merge tree contained in β (i.e. if β is delimited by a saddle v, it is represented by the child of of v in the merge tree which is contained in β.) We only take the union of β with the sets corresponding to its children, so it satisfies the restriction.

5.3.4 Analysis of the algorithm

The correctness of the algorithm is straightforward. It is easy to verify that the initial estimate of an event time in $\mathcal{Q}_T$ is correct after the predecessor event of

the event in question has been processed. Similarly we can show that when a pair (v, δ_v) is removed from the priority queue $\mathcal{Q}_V$, δ_v is the correct value of the change in ϕ_v at that time.

Next, we analyze the running time of the algorithm. First, we examine the time spent performing operations operations on $\mathcal{Q}_T$ and handling time-events outside of the second-level sweep. If $|\xi|$ time-events are processed, we spend $O(|\xi| \log |\xi|)$ total time performing DeleteMin operations to remove the time-events as we process them. We must update an event's time whenever the fill-rate of an active depression changes. There are two possible cases where this will occur.

The first case is when we are processing a depression event. The fill-rates of depressions must be increasing in this case, so each update can be performed with a DecreaseKey operation, which takes constant time. We update the fill-time of a depression whenever the flow rate of an edge (u, v) changes where v is flooded (i.e. water is flowing into a partially flooded active depression), and the new fill-time can be computed in constant time. All together, the total time spent updating $\mathcal{Q}_T$ when handling depression events is $O(|\phi|)$.

The second case is when we are processing rain events, where the fill-time of depressions may increase. If so, we need to first delete the event and then reinsert it with the new time, which takes $O(\log |\xi|)$-time. In the worse case, we may need to update $O(|\xi|)$ events at each of the k rain events, so the total time spent processing them will be $O(k|\xi| \log |\xi|)$.

In either case, when processing a time event, we must also update the set of active depressions. If all sibling depressions are already flooded, we also merge the sets $\mathcal{W}_\xi$ of wet vertices and update the fill-time of the newly active parent depression. The bit representing if a depression is active will only change twice, when it first becomes active and when it becomes inactive as its parent becomes the active depression. Therefore updating the active status of each depression can be done in amortized constant time. As described, merging the sets using the data structure $\mathcal{K}$ is also amortized constant.

Now we will examine the time spent performing the second-level sweeps. For each change in a vertex flow-rate function, we call DeleteMax and UpdateFlow once. Letting $|V_{\max}| = \max_\tau V_\tau$, these operations will take a total of $O(|\phi| \log \min(\log n, |V_{\max}|))$ time. We can perform each IsFlooded operation in constant time, and it will be called once for each corresponding change in edge flow-rate functions. Therefore the total time spent performing IsFlooded operations will be $O(|\phi|)$. Finally, each FloodedVertex operation, with the exception of the time spent updating $\mathcal{Q}_T$ which we accounted for above, can performed in constant time, utilizing the constant time Find operation on the data structure $\mathcal{K}$.

Therefore the total time spent performing the second-level sweeps, not including the time spent updating $\mathcal{Q}_T$, is $O(|\phi| \log \min(\log n, |V_{\max}|)$. As argued above the time spent in updating $\mathcal{Q}_T$ is $O(|\phi| + k|\xi| \log |\xi|)$. Putting this all together, we have the following:

Theorem 5.1. Given a triangulation $\mathbb{M}$ of $\mathbb{R}^2$ with n vertices and a height function $h : \mathbb{M} \to \mathbb{R}$ that is linear on each face of $\mathbb{M}$, a data structure of size $O(n)$ can be constructed in $O(n \log n)$ time so that for a (time varying) rain distribution R, a terrain-flow query can be answered in $O(|\phi|(\log(\min(\log n, |V_{\max}|)) + k|\xi| \log |\xi|)$ time, where $|\phi|$ is the total complexity of all non-zero flow-rate functions, $|V_{\max}|$ is the maximum number of flow-rate functions changing at any given time, $|\xi|$ is the number of events, and k is the number of times the rain distribution changes.

Remark. We note that whenever the fill time of a depression increases, it also corresponds to a change in flow-rate of an edge. Therefore we can also say the total time spent processing updates to $\mathcal{Q}_T$ of this type is $O(|\phi| \log |\xi|)$. Noting that $|\xi| = O(|\phi|)$ and $|V_{\max}| = O(|\phi|)$, we have the simpler, if less precise bound on running-time of $O(|\phi| \log |\phi|)$.

5.4 I/O-Efficient Algorithms

In this section, we describe two I/O-efficient algorithms that given a terrain $\Sigma = (\mathbb{M}, h)$ and a rain distribution R determine the flow-rate $\phi_{(u,v)}(t)$ for all edges $(u, v) \in \mathbb{M}$.

In the preprocessing step of both algorithms, we compute the merge tree T of Σ and label each node in T according to its in-order traversal as described in [77]. Furthermore, we compute $\mathrm{Vol}(\beta_v)$ for each vertex $v \in \Sigma$ and augment each edge $(u, v) \in \mathbb{M}$ with the index of the smallest maximal depression containing v. This can be computed in $O(\mathrm{Sort}(n))$ I/Os using the algorithm described by Arge et al. [21].

The internal memory terrain-flow algorithm described in the previous section does not trivially extend to the I/O-model since each time event is handled by sweeping only the relevant vertices of $\mathbb{M}$. It is difficult to implement this sweep I/O-efficiently because we do not have random access to disk in the I/O-model. Instead of extending the previously described algorithm, the algorithms perform a single upward sweep of the merge tree T followed by a downward sweep of $\mathbb{M}$. By sorting the vertices of $\mathbb{M}$ in an I/O-efficient priority queue, we can trivially sweep the terrain by scanning through the list of vertices in their descending height order. We use the I/O-efficient priority queue by Brodal [37] which performs n insertations and deletions using $O(\mathrm{Sort}(n))$ I/Os and $O(n \log n)$ internal memory computation time.

In the first algorithm, we assume that $(\chi+k)\rho = O(m)$, where χ is the height of the merge tree T, k is the number of times at which the rain distribution changes, and ρ is the number of sinks in $\mathbb{M}$. Under this assumption, we can explicitly store the fill-rates of all maximal depressions in internal memory. For the second algorithm, we relax the assumption to $\rho = O(m)$ at the cost of a greater number of I/Os. For both algorithms, we store the merge tree T in internal memory.

5.4.1 Computing flow-rates I/O-efficiently

We now describe our first I/O-efficient algorithm. Given a rain distribution R, the algorithm begins by performing an upward sweep of the merge tree T. During the sweep it computes $R(\alpha, \cdot)$ for each depression α delimited by a vertex of T as follows: First assign $R(v, \cdot)$ for each vertex with nonzero rainfall to the smallest maximal depression containing v and then perform an upward sweep on T, maintaining the sum of rainfall functions at each vertex of T. This upward sweep can be trivially implemented in $O(\mathrm{Scan}(n) + \mathrm{Sort}(|R|))$ I/Os and $O(n + |R| \log |R|)$ internal computation time by storing T and the sums of rainfall functions in memory.

Next, we perform a downward sweep. To describe the sweep, we introduce the following notation: For a height ℓ and a maximal depression α, let $\boldsymbol{E}_\ell(\alpha)$ denote the set of edges $(u, v) \in \mathbb{M}$ where $h(v) < \ell \leq h(u)$ and $v \in \alpha$. Let $\mathbb{V}_\alpha$ be the subset of vertices for which α is the smallest maximal depression containing them. We define the subset $\hat{\boldsymbol{E}}_\ell(\alpha) = \{(u, v) \in \boldsymbol{E}_\ell(\alpha) \mid v \in \mathbb{V}_\alpha\}$.

We process vertices of $\mathbb{M}$ in descending height order. When the sweep plane is at height ℓ, we maintain the following information:

1. for each depression α in the sublevel set $h_{<\ell}$, maintain the fill-rate F_α,
2. for each maximal depression β lying below the sweep plane (i.e. the height of the saddle delimiting β is at most ℓ), maintain $\hat{\Phi}_\beta = \sum_{(u,v) \in \hat{\boldsymbol{E}}_\ell(\beta)} \phi_{(u,v)}$, and
3. for each edge (u, v) crossing the sweep plane, we maintain the flow-rate $\phi_{(u,v)}$ in an I/O-efficient priority queue $\mathcal{Q}$ keyed on the height of vertex v.

Note that F_α depends only on the sinks contained in α, so it changes only at negative saddles. Furthermore, we initialize $\mathcal{Q}$ by inserting the functions $R(v, \cdot)$ for each vertex v with $R(v, \cdot) > 0$. Whenever we process a vertex v, we remove functions $\phi_{(u,v)}$ from $\mathcal{Q}$, compute ϕ_v, and for all outgoing edges $(v, w) \in \mathbb{M}$ we propagate the flow-rate $\phi_{(v,w)}$ along the edge (v, w) and insert $\phi_{(v,w)}$ into $\mathcal{Q}$. The fill-rates F_α, which are maintained for each depression α in the sublevel set h_ℓ, and the functions $\hat{\Phi}_\beta$, which are maintained for each depression β lying below the sweep plane, can be stored in memory since we assume $\rho(\chi + k) = O(m)$. We now describe in detail how to process each vertex v as we encounter it in our sweep.

Non-negative-saddle vertex. If v is a non-negative-saddle vertex lying in a depression α_i, we first compute ϕ_v using (5.3); note that $\phi_{(u,v)}$ for all incoming edges to v has been computed and can be removed from the front of the priority queue $\mathcal{Q}$. Additionally, using $F_{\alpha_i}(t)$ along with $\mathrm{Vol}(\beta_v)$, we determine if or when v becomes flooded and update ϕ_v accordingly. Let α be the smallest maximal depression containing v, i.e., $v \subset \mathbb{V}_\alpha$. We set $\hat{\Phi}_\alpha = \hat{\Phi}_\alpha - \sum_{(u,v)} \phi_{(u,v)}$. Next,

for each edge $(v, w) \in \mathbb{M}$ where $\lambda(v, w, t) > 0$ we compute the flow-rate $\phi_{(v,w)}$ using (5.2) and push the result onto the priority queue $\mathcal{Q}$. Let β be the smallest maximal depression containing w, i.e., $w \in \mathbb{V}_\beta$. We set $\hat{\Phi}_\beta = \hat{\Phi}_\beta + \phi_{(v,w)}$. Note, that each edge flow-rate is added to only one function and the complexity of each function is $O(k + \chi)$, where k is the number of times the rain distribution changes and χ is the height of the merge tree of the terrain. Furthermore, for each incoming edge to v with non-zero flow, we perform one deletion from the priority queue $\mathcal{Q}$. Correspondingly, for each outgoing edge with non-zero flow, we perform one insertion into $\mathcal{Q}$. Thus, letting $|\phi|$ be the total complexity of the flow-rate functions, we spend $O(n(\chi+k)+|\phi| \log |\phi|)$ internal computation time and $O(\text{Scan}(n) + \text{Sort}(|\phi|))$ I/Os processing the non-negative-saddle vertices.

Negative-saddle vertex. Let v be a negative saddle delimiting two depressions α and β. As discussed in Section 5.2.2, computing ϕ_v is more involved—it depends on which of α or β fills first and when. We first compute the *pseudo-flow-rate function*, ϕ'_v, using (5.3) and note that $\phi'_v(t) = \phi_v(t)$ for $t \leq \min(\tau_\alpha, \tau_\beta)$, i.e., until one of α or β becomes full. We also compute $\phi'_{(v,w)}$ for all edges $(v, w) \in \mathbb{M}$ using (5.2). Let ξ be the maximal depression such that $w \in \mathbb{V}_\xi$. Then we update $\hat{\Phi}_\xi$ as above. Next, we compute the pseudo-fill-rates $F'_\alpha(t)$ and $F'_\beta(t)$ as follows: $F'_\alpha = R(\alpha, \cdot) + \sum_{\alpha_i \subseteq \alpha} \hat{\Phi}_{\alpha_i}$ and $F'_\beta = R(\beta, \cdot) + \sum_{\beta_i \subseteq \beta} \hat{\Phi}_{\beta_i}$, where the sum is taken over all maximal depressions α_i (resp. β_i) that lie in α (resp. β), i.e., the edge corresponding to α_i (resp. β_i) lie in the subtree below the edge corresponding to α (resp. β). See Figure 5.6 for an example. Note that $F'_\alpha(t) = F_\alpha(t)$ and $F'_\beta(t) = F_\beta(t)$ for $t \leq \min\{\tau_a, \tau_b\}$. Next, using F'_α, F'_β and $\text{Vol}(\alpha), \text{Vol}(\beta)$, we compute which of α and β fills first. If neither of them becomes full, $\phi'_v, \phi'_{(v,w)}, F'_\alpha$ and F'_β are correct flow and fill-rate functions for all values of t and we are done. So assume that one of them, say, α, becomes full first. We truncate these functions at τ_α. $S_\alpha(t) = F'_\alpha(t)$ for $t > \tau_\alpha$. We now compute $\phi_v(t)$ for $t > \tau_\alpha$, using (5.4), and $\phi_{(v,w)}(t)$, for $t > \tau_\alpha$, using (5.2), (5.4) and (5.5). Note that $\phi_{(v,w)}$, for $w \in \alpha$, is not defined for $t > \tau_\alpha$.

Finally, we propagate the flow-rates on the outgoing edges by pushing $\phi_{(v,w)} = \lambda(v, w, \cdot) \cdot \phi_v$ to $\mathcal{Q}$ for all vertices w, where $\lambda(v, w, \cdot) > 0$. Furthermore, for each outgoing edge (v, w) we update $\hat{\Phi}_\xi$ for the maximal depression ξ with $w \in \mathbb{V}_\xi$.

Again, each non-zero edge flow-rate is forwarded only once using $\mathcal{Q}$. Thus, we spend $O(|\phi| \log |\phi|)$ internal memory computation time and $O(\text{Sort}(|\phi|))$ I/Os updating flow-rates in $\mathcal{Q}$. Computing the fill-rates at saddle vertices does not require any additional I/Os, since we maintain $R(\beta, \cdot)$ and $\hat{\Phi}_\beta$ in internal memory for each maximal depression β lying below the sweep plane. However, in order to reduce the total internal memory computation time used, we speed up the computation of the fill-rates F'_α and F'_β in internal memory; recall that when computing the fill-rate F'_α (resp. F'_α) we sum over $\hat{\Phi}_{\alpha_i}$ (resp. $\hat{\Phi}_{\beta_i}$) for

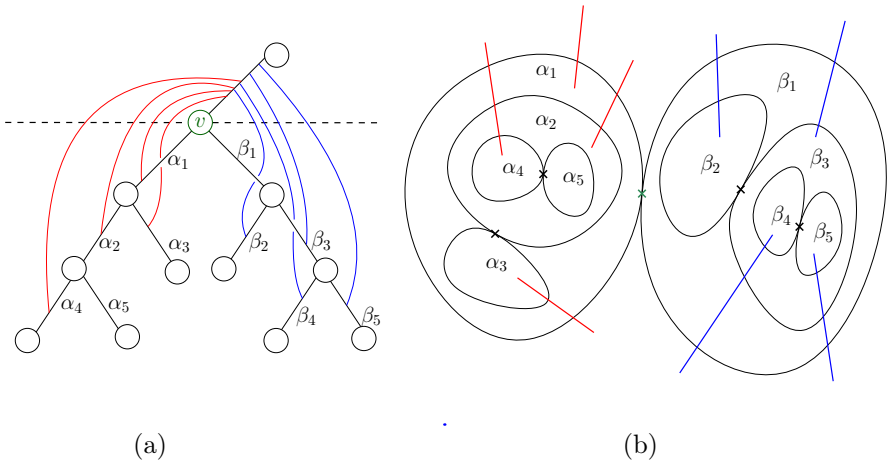


Figure 5.6: An example terrain where the sweep plane is at negative-saddle vertex v delimiting depressions α and β. (a) Merge tree with red and blue edges crossing the sweep line into α and β respectively. (b) Contour of the terrain with the corresponding edges drawn.

each maximal depression α_i (resp. β_i) contained in α (resp. β). Thus, explicitly computing F'_α (resp. F'_β) requires $O(|\alpha|(\chi+k))$ (resp. $O(|\beta|(\chi+k))$) internal computation time, where $|\alpha|$ (resp. $|\beta|$) is the number of maximal depressions contained in α (resp. β). Recall that β_v is the depression delimited by v. Since $F_{\beta_v} = F'_\alpha + F'_\beta$, it suffices to compute only one of the fill-rates F'_α and F'_β. We compute the fill-rate for the depression which contains the fewest number of maximal depressions. This can trivially be implemented by storing the sizes of each subtree in the merge tree T. Letting $T(\beta_v)$ be the total internal memory computation time used to compute fill-rates for all saddles contained in β_v, we can bound $T(\beta_v)$ using the following recursion:

$$T(\beta_v) = O\left((\chi+k)\min(|\alpha|,|\beta|)\right) + T(\alpha) + T(\beta)\,. \tag{5.6}$$

Noting that α and β are disjoint, it follows that $|\alpha|+|\beta| \leq |\beta_v|$. Using this, the recurrence solves to $T(\beta_v) = O((\chi+k)|\beta_v|\log|\beta_v|)$. Thus, we can compute all fill-rates in time $O((\chi+k)\rho\log\rho)$. The total internal computation time used at saddle vertices is thus $O((\chi+k)\rho\log\rho + |\phi|\log|\phi|)$ and the total number of I/Os used at saddle vertices is $O(\mathrm{Scan}(\rho) + \mathrm{Sort}(|\phi|))$.

Theorem 5.2. Given a triangulation of $\mathbb{M}$ with n vertices, a height function $h : \mathbb{M} \to \mathbb{R}$ which is linear on each face of $\mathbb{M}$ and a rain distribution R, a terrain-flow query can be answered in $O(\mathrm{Scan}(n) + \mathrm{Sort}(|\phi|))$ I/Os and $O((\chi+k)(n+\rho\log\rho)+|\phi|\log|\phi|)$ internal computation time $O(\mathrm{Sort}(n))$ using preprocessing assuming $\rho(\chi+k) = O(m)$, where $|\phi|$ is the total complexity of all flow-rate functions which we return, χ is the height of the merge tree, k is

the number of times at which the rain distribution changes, ρ is the number of sinks in $\mathbb{M}$, and m is the size of internal memory.

5.4.2 Assuming smaller internal memory

We now extend the algorithm to relax the assumption on the size of the internal memory from $\rho(\chi + k) = O(m)$ to $\rho = O(m)$, at the cost of a log factor in the number of I/Os. We use the same framework as described previously. However, we do not store fill-rates F_α in memory for each depression α in the sublevel set h_ℓ. Furthermore, we do not maintain the functions $\hat{\Phi}_\beta$ for each maximal depression β lying below the sweep plane. Instead, we use the priority queue $\mathcal{Q}$ to forward fill-rates as well as the edge flow-rates used to compute fill-rates at negative saddle vertices.

Forwarding fill-rates. Let v be a non-negative-saddle vertex, and let α be the maximal depression such that $v \in \mathbb{V}_\alpha$. Let u be the vertex visited after v in the downward sweep, such that $u \in \mathbb{V}_\alpha$. When performing the sweep, we forward F_α from v to u using $\mathcal{Q}$. We note that we can augment v with the height of u using $\mathrm{Sort}(n)$ I/Os in preprocessing, and thus we can forward F_α to u during the sweep. Furthermore, for each negative saddle vertex v delimiting depressions α and β, we forward F_α and F_β to the first vertices visited in α and β, respectively.

Computing fill-rates at negative saddles. Let v be a negative saddle vertex with height ℓ that delimits two depressions α and β. During the execution of the sweep, we compute the fill-rates of depressions α and β. We recall that the pseudo-fill-rate of α can be computed as follows:

$$F'_\alpha = R(v, \cdot) + \sum_{(u,v) \in \boldsymbol{E}_\ell(\alpha)} \phi_{(u,v)} \,. \tag{5.7}$$

We note that $R(v, \cdot)$ and the flow-rates used to compute this sum could be forwarded using $\mathcal{Q}$. However, that would lead to forwarding $\Theta(n\chi)$ functions in the worst-case. Recall that F_{β_v} is forwarded to v using $\mathcal{Q}$. Furthermore, since $F_{\beta_v} = F'_\alpha + F'_\beta$, it suffices to compute either F'_α or F'_β, whichever requires the fewest flow-rates to be forwarded. The number of flow-rates that need to be forwarded to compute F'_α (resp. F'_β), can be precomputed by counting the number of edges crossing the boundaries of α (resp. β). This precomputation step can trivially be implemented by performing a scan of the vertices using $O(\mathrm{Scan}(n))$ I/Os and $O(\rho)$ memory. We, therefore, preprocess for which depressions we compute fill-rates and forward only the flow-rates required for computing those.

We now bound the number of edges forwarded using a similar recurrence as previously; let $|\alpha|$ denote the total number of edges with at least one vertex contained in the depression α and note that $|\alpha| \geq |\boldsymbol{E}_\ell(\alpha)|$. Letting $T(\beta_v)$ be the total number of flow-rates summed to compute the fill-rates for all saddles

contained in β_v,

$$T(\beta_v) = O\big(\min(|\alpha|, |\beta|)\big) + T(\alpha) + T(\beta)\,. \tag{5.8}$$

Noting that the set of edges with vertices in the two maximal depressions α and β are disjoint, it follows that $|a|+|\beta| \leq |\beta_v|$. Using this, the recurrence solves to $T(\beta_v) = O(|\beta_v| \log |\beta_v|)$, and we thus need to forward only a total of $O(n \log n)$ flow-rate functions using the priority queue. Since the complexity of each flow-rate function is bounded by $O(\chi+k)$, we spend a total of $O(\mathrm{Sort}((\chi+k)n \log n))$ I/Os and $O((\chi + k)n \log n \log(kn))$ internal computation forwarding edges and computing fill-rates.

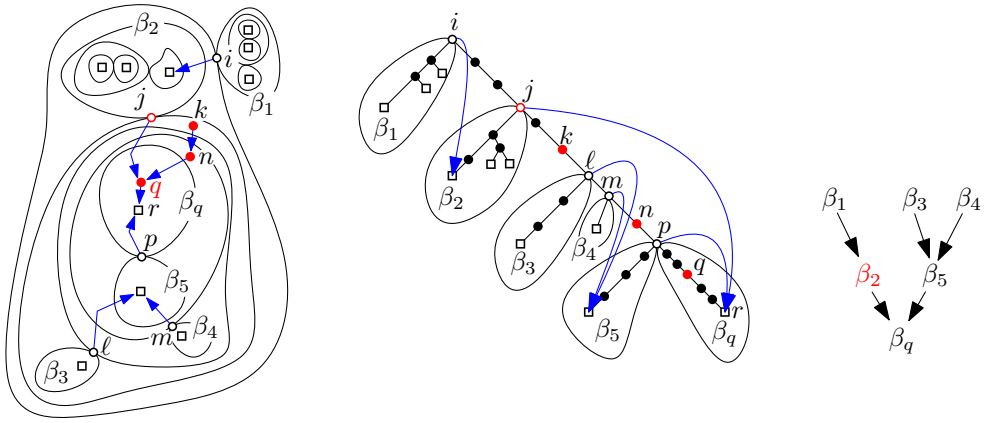
Theorem 5.3. Given a triangulation of $\mathbb{M}$ with n vertices, a height function $h : \mathbb{M} \to \mathbb{R}$ which is linear on each face of $\mathbb{M}$ and a rain distribution R, a terrain-flow query can be answered in $O(\mathrm{Sort}((\chi + k)n \log n))$ I/Os and $O((\chi + k)n \log n \log(kn))$ internal computation time assuming $\rho = O(m)$, where χ is the height of the merge tree, k is the number of times at which the rain distribution changes, ρ is the number of sinks in $\mathbb{M}$, and m is the size of internal memory.

5.5 Vertex-Flow Query

The terrain-flow query computes the flow-rate for every vertex and edge, so one could answer vertex-flow queries by computing the terrain-flow query and then returning $\phi_q(t)$ for the query vertex $q \in \mathbb{M}$. While in practice the query time can be improved, the worst-case running time under the MFD model remains the same as for the terrain-flow query. Under the SFD model, we can improve the running time significantly, building on the fast algorithm for the flood-time query under SFD by Rav *et al.* [89], along with a linear-size data structure supporting constant-time reachability queries in planar directed graphs given by Holm *et al.* [64].

The key idea of the algorithm is that under the SFD model, when water falls on a vertex or spills from a negative saddle, the water flows along a single path to some sink in the terrain. Thus, if we can find vertices and negative saddles from which water follows a path containing q in the flow graph, i.e., upstream vertices of q, then ϕ_q will be the sum of water falling directly on or spilling from these sources. Before describing the algorithm, we introduce a few definitions.

Any given point $q \in \mathbb{M}$ is contained in a sequence of maximal depressions $\alpha_1 \supset \cdots \supset \alpha_k \ni q$. Each α_i is delimited by a saddle v_i and has a corresponding sibling depression β_i. These saddles form a path in T from q to the root. We refer to the maximal depressions $\beta_1, \ldots, \beta_{k-1}$ as the *tributaries of* q and denote them by B_q. See Figure 5.7(b). In the SFD model, all upstream vertices of q must be ancestors in the merge tree. Therefore the only depressions that can



(a) (b) (c)

Figure 5.7: (a) An example terrain, with sinks marked as boxes, and saddles delimiting tributaries of q with open circles. The path in the flow graph from labeled vertices marked blue, and vertices upstream of q marked red. (b) The merge tree with q-tributaries $\beta_1, \cdots, \beta_5$ and arrows pointing from each saddle to the sink they flow to when β_i is full. (c) The tributary tree $\mathcal{T}_q$ rooted at β_q with each vertex denoting a tributary of q

spill and have water reach q are the tributaries. Further, for the purposes of computing the flow rate at q, the behavior inside each tributary is irrelevant. The relevant information is when the tributaries become full, and where the water spills to when they do.

For any point $q \in \mathbb{M}$, we define the *tributary tree* $\mathcal{T}_q$ as follows. $\mathcal{T}_q$ is a directed graph with nodes corresponding to the tributaries of q plus β_q, the smallest maximal depression containing q. There is an edge (β_i, β_j) in $\mathcal{T}_q$ if water spills from the saddle v_i to a sink in β_j when β_i becomes full. Water spills to exactly one sink under the SFD model, so $\mathcal{T}_q$ is a tree rooted at β_q. See Figure 5.7(c).

The set of q-tributaries and $\mathcal{T}_q$ are independent of the rain distribution. Given the set of tributaries the rain distribution initially falls in $\mathcal{B}$, to efficiently compute the vertex-flow query we also make use of the following. Given a tributary tree $\mathcal{T}_q$ and a set $\mathcal{B}$ of tributaries, we say the *compressed tributary tree* $\hat{\mathcal{T}}_q(\mathcal{B})$ is the subtree of $\mathcal{T}_q$ formed by the paths in $\mathcal{T}_q$ from each $\beta_i \in \mathcal{B}$ to the root in $\mathcal{T}_q$, with degree 2 nodes that are not in $\mathcal{B}$ removed. Each node Π in $\hat{\mathcal{T}}_q(\mathcal{B})$ corresponds to a path in $\mathcal{T}_q$ starting at a node with degree greater than 2 or in $\mathcal{B}$, and all other nodes in the path consist of degree-2 nodes that are not in $\mathcal{B}$. We define the volume Π_i to be the sum of volumes of the tributaries in the path of $\mathcal{T}_q$ corresponding to Π_i. See Figure 5.8.

We present an $O(n \log n)$-time algorithm for preprocessing Σ into a linear-size data structure that can answer an vertex-flow query for a given rain

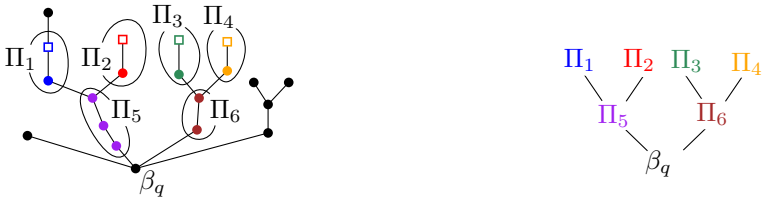


Figure 5.8: (a) the tributary tree $\mathcal{T}_q$ rooted at β_q with each vertex denoting a tributary of q, rain initially falls in the tributaries marked as squares. (b) the compressed tributary tree for the given rain distribution. Each node Π_i stores the volume of all tributaries in the compressed path.

distribution R and query vertex q. The query takes $O(|R| + |\mathcal{B}|k \log n)$ time, where $|\mathcal{B}|$ is the number of q-tributaries with non-zero initial rainfall.

We note, that one could use the vertex query to compute the flow rate of an edge as well. Given a query edge $(q, r) \in \mathbb{M}$ and a rain distribution R, we begin by assuming that water flows from q to r. Since water only flows from each vertex to one neighbor in the SFD model, if this were not the case then we would immediately have that $\phi_{(q,r)} = 0$. Hence, $\phi_{(q,r)} = \phi_q$, so the edge-flow query is equal to the vertex-flow query.

Vertex-flow query algorithm. With these definitions in hand, we are ready to describe the algorithm. It begins by building $\mathcal{T}_q$. Refer to Figure 5.8 for an example. As we have noted, water will reach q in one of two cases: rain falls on a vertex v and follows a path that reaches q, or water spills from a tributary of q and reaches q.

Consider first the case when rain falls only on a single point p contained in some tributary β_{i_1}. Take the path in $\mathcal{T}_q$ from β_{i_1} to the root β_q, $(\beta_{i_1}, \beta_{i_2}, \cdots, \beta_{i_k}, \beta_q)$. For each depression β_{i_j}, let Vol_j denote the depression volume of β_{i_j} and let τ_j be the fill-time of β_{i_j}. The fill-time τ_k, when βi_k begins spilling into β_q, will be when the volume of rain falling on p equals $\sum_{j=1}^{k} \text{Vol}_j$. Moreover, we have

$$F_{\beta_q}(t) = S_{\beta_{i_k}}(t) = \begin{cases} 0 & t < \tau_k, \\ F_{\beta_{i_1}}(t) & t \geq \tau_k. \end{cases}$$

Therefore, instead of computing the fill and spill rates for each tributary along the path, we can compress all the tributaries in this path and treat it as if it were a single depression. Then to answer the query, check whether there is a path in the flow graph from the saddle delimiting β_{i_k} to the query vertex q. If yes, then $\phi_q(t) = S_{\beta_{i_k}}(t)$. If no, then water reaches q only when it gets flooded, so while it is defined $\phi_q(t) = 0$. In either case, we have that $F_{\beta_q}(t) = S_{\beta_{i_k}}(t)$, so we can also determine the time τ_{β_q} at which q becomes flooded, after which ϕ_q is undefined. For simplicity, in the general case, we assume we first compute

the flood-time of q using the algorithm in Rav *et al.* [89]. While that algorithm only considers fixed rain distributions, a straightforward generalization leads to a version that computes the flood-time for a rain distribution that changes at k times in $O(|R| + |\mathcal{B}|k \log n)$ time where $|R|$ is the complexity of the rain distribution and $|\mathcal{B}|$ is the number of tributaries that rain falls directly in.

Now consider the general case where rain falls on many vertices. There are two types of upstream vertices of q that can contribute to ϕ_q: upstream vertices on which rain directly falls, and upstream vertices that are saddles delimiting a tributary β_i into β_q. In the latter case we will say the tributary β_i is upstream of q, as when it becomes full the water spilling from it follows a path in the flow graph to q. Note that only children of β_q in $\mathcal{T}_q$ can possibly be upstream tributaries of q. We then compute ϕ_q as a sum of flow-rate functions on these upstream vertices: namely the rainfall functions on vertices that are upstream of q, and the spill-rates of upstream tributaries of q (e.g. the red vertices in Figure 5.7).

We begin by computing the initial fill-rate of each tributary in which rain falls directly. For each vertex v from which rain flows to a sink contained in β_q, check whether v is upstream of q. If so, add the rainfall on v to the sum. In either case, add the rainfall on v to the fill-rate of β_q. If rain falls in multiple tributaries that have disjoint paths in $\mathcal{T}_q$ to β_q (excluding the root β_q), we can simply perform the single-point rain algorithm multiple times for each path and add to the sum each spill-rate of depressions that reach q . However two paths may merge at a tributary γ before reaching β_u. (e.g. Π_3 and Π_4 in Figure 5.8.) Here, we can compute F_γ as the sum of spill-rates of its children tributaries, and recurse, treating γ as a new single-point source. Cases where more than two paths merge at a single vertex can be handled in a similar manner. Letting $\mathcal{B}$ be the set of tributaries with non-zero fill-rate at $t = 0$, the tributaries where paths merge will be non-leaf nodes in the compressed tributary tree $\hat{\mathcal{T}}_q(\mathcal{B})$.

Now it remains to show how we can implement this algorithm efficiently. There are two main operations needed. First, we must compute the fill and spill-rates of the q-tributaries. Then we must determine which saddles and vertices are upstream of q. To facilitate the former, we build the data structure in [89], which given a set of m vertices on which it is raining, letting $\mathcal{B}$ be the set of tributaries that rain initially flows to, it can return, in $O(m + |\mathcal{B}| \log n)$-time, the compressed tributary tree $\hat{\mathcal{T}}_q(\mathcal{B})$[1]. We omit the details here, but note the key idea that allows these queries to be performed efficiently. Two tributary trees $\mathcal{T}_q$ and $\mathcal{T}_p$ will differ if p and q lie on different edges of T. However the set of tributary trees for all vertices have a high degree of overlap— for two vertices p and q, all tributaries delimited by saddles above $\text{lca}(p, q)$ in T will be the same. This overlap allows us to preprocess T so that compressed tributary trees can be queried efficiently. Then with the compressed tributary tree, rooted at β_q, with each node being a path of tributaries, we process the nodes from the

[1] In [89] this process is referred to as the (B, z)-subtree operation.

leaves towards the root, computing the fill and spill-rates of each compressed set of tributaries.

Computing the initial fill-rates takes $O(|R|)$ time. As in [89] we represent each fill-rate as a sequence of pairs, (t, δ), of times t at which the fill-rate changes by δ. By storing these fill-rate functions as a Fibonacci heap, keyed on time, we can compute the sum of two spill-rate functions in constant time by simply merging the two heaps. Computing a spill-rate function consists of removing a prefix of the spill-rate functions. Each removal corresponds to a delete-min operation which takes $O(\log n)$-time, and we begin with $|\mathcal{B}|$ fill-rate functions each with complexity k. Since we only perform removals, the complexity of the spill-rate function of a depression can never increase compared to the complexity of the fill-rate function. Therefore we can compute the spill-rate functions of all compressed paths in $O(|\mathcal{B}|k \log |\mathcal{B}|k)$-time.

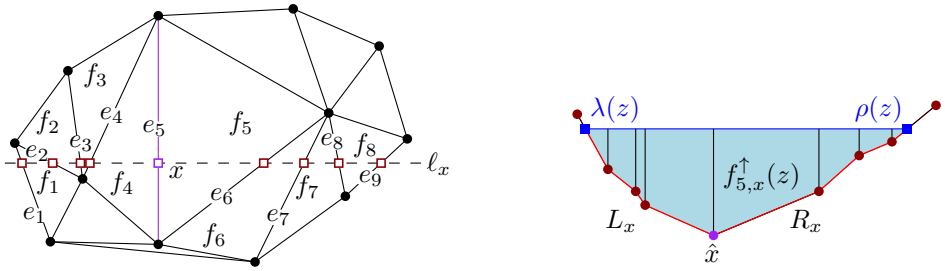
To compute which saddles and vertices water are upstream of q, we build the data structure for planar reachability queries as part of the preprocessing step, as described in [64], on the flow graph. This data structure is linear in size and given a pair of vertices p and q returns in constant time if q is reachable in a graph from p. Water will reach q from a vertex p (i.e. q is downstream of p) if and only if there is a path from p to q in the flow graph. Thus for each of the $O(|R|)$ vertices on which it is raining we can simply perform a constant time query of the data structure to see if the rain at these vertices flows directly over q. Similarly, when we compute the spill-rate function of each tributary, we can also query the data structure to see if the water spills over q. Performing these queries will take a total of $O(|R| + |\mathcal{B}|)$ time.

Theorem 5.4. Given a triangulation $\mathbb{M}$ with n vertices and a height function $h : \mathbb{M} \to \mathbb{R}$ which is linear on each face of $\mathbb{M}$, a data structure of size $O(n)$ can be constructed in $O(n \log n)$ time so that for a (time varying) rain distribution R and a vertex q a vertex-flow query can be answered in $O(|R| + |\mathcal{B}|k \log n)$, where $|R|$ is the complexity of the rain distribution, $|\mathcal{B}|$ is the number of tributaries in which rain is falling directly, and k is the number of times at which the rain distribution changes.

5.6 Extracting 2D Flow Networks

So far we have assumed that water flows along the edges of Σ and computed the flow rate of water along these edges. But in reality, the water flows along 2D channels forming a 2D network of rivers. We first describe a model for determining the 2D channels given a flow along a path of $\mathbb{M}$, and then present an efficient algorithm for computing these channels.

Figure 5.9: A 2D channel from a 1D network using Manning's equation. The 1D network is given in dark blue and cross sections in purple. The figure is generated using SCALGO software [91].



(a) (b)

Figure 5.10: (a) Σ with sweep line ℓ_x. (b) Σ_x with water at height z. The wetted perimeter is marked in red.

5.6.1 Model for 2D channels

We assume that we are given a path P along the edges of $\mathbb{M}$. For each edge e of P, let $\phi_e \in \mathbb{R}_{\geq 0}$ be the flow value along e. Unlike previous sections, we assume that ϕ_e does not vary with time, but it may vary with the edges of P. The goal is to compute a 2D channel $\mathcal{C} := \mathcal{C}(\mathsf{P}, \phi)$ along which water flows.[2] See Figure 5.9. The channel $\mathcal{C}$ is defined by its left and right banks. More precisely, P is parameterized as $\mathsf{P} : I \to \mathbb{R}^2$ where $I = [x_0, x_1]$ is an interval. For every $x \in I$, we define $\mathrm{Lb}(x), \mathrm{Rb}(x) \in \mathbb{R}^2$ as the left and right bank, respectively, of $\mathcal{C}$ at x, and $\Delta(x) = h(\mathrm{Lb}(x)) = h(\mathrm{Rb}(x))$. The locus of points $\mathrm{Lb}(x)$ (resp. $\mathrm{Rb}(x)$), for $x \in I$, traces a curve Lb (resp. Rb), which is the *left* (resp. *right*) *bank* of $\mathcal{C}$. Here we assume that each of Lb and Rb is a non-intersecting curve; if either of them is self-intersecting, then $\mathcal{C}$ has to be defined more carefully.

[2]Intuitively, P and flow values can be constructed from the edge flow-rate computed by the algorithms described in Sections 3 and 4, by fixing a time and thresholding the flow rates.

We overlay $\mathbb{M}$ with Lb and Rb. $\mathcal{C}$ is the portion of this overlay between Lb and Rb. The complexity of $\mathcal{C}$, denoted by $|\mathcal{C}|$, is the number of vertices in $\mathcal{C}$.

To estimate $\mathrm{Lb}(x)$ and $\mathrm{Rb}(x)$, we use Manning's equation [81], a widely used empirical formula relating the channel geometry and flow rate, as follows. Let x be a point on an edge $e \in \mathbb{M}$ with flow value ϕ_e. Let ℓ_x be the line in the xy-plane passing through x and normal to e, and let $\Pi_x = \ell_x \times \mathbb{R}$ be the vertical plane containing ℓ_x. Let $\Sigma_x = \Sigma \cap \Pi_x$ be the cross-section of Σ in Π_x, which we refer to as the *profile* of Σ at x. Σ_x is a polygonal chain whose vertices (resp. edges) are the intersection points of edges (resp. faces) of Σ with Π. See Figure 5.10. Let $\hat{x} = (x, h(x))$ be the vertex on Σ_x corresponding to the point $x \in e$, i.e. $\hat{x}$ is vertically above x.

We divide Σ_x into two polygonal rays L_x, R_x at the vertex $\hat{x}$, with L_x (resp. R_x) lying to the left (resp. right) of P. For a height $z \geq h(x)$, let $\lambda(z)$ (resp. $\rho(z)$) be the first point on L_x (resp. R_x) at height z as we walk along L_x (resp. R_x) starting from $\hat{x}$. Let $A_x(z)$ denote the area of the polygon formed by the segment $\lambda_x(z)\rho_x(z)$ and the portion of Σ_x between $\lambda_x(z)$ and $\rho_x(z)$, and let $P_x(z)$, called the *wetted perimeter*, denote the arc length of Σ_x between $\lambda_x(z)$ and $\rho_x(z)$. If the water has height z at x, then Manning's equation [81] states that the flow rate at x is

$$\phi_x(z) = \frac{A_x(z)^{5/3}\sigma_e^{1/2}}{\mu_e P_x(z)^{2/3}}, \tag{5.9}$$

where σ_e is the slope in the z-direction of the edge of Σ corresponding to e, and μ_e is Manning's roughness coefficient. We assume that we are given the value of μ_e, which depends on the material of the terrain at e (e.g. concrete, dirt, light brush, etc.). Manning's equation is typically used to compute the flow rate $\phi_x(z)$ of rivers given a measurement of the river depth and approximate channel geometry. Here instead we solve an inverse problem: given the flow rate ϕ_x at x, determine the depth and width of the river at x. Let $\Delta(x)$ be the value of z for which $\phi_x(z) = \phi_e$. We set $\mathrm{Lb}(x) = \lambda(\Delta(x))$ and $\mathrm{Rb}(x) = \rho(\Delta(x))$, i.e., the *river bank* points on Σ corresponding to x; Let $\mathcal{C}_x = \Sigma_x[\mathrm{Lb}(x), \mathrm{Rb}(x)]$ be the profile of the channel at x. See Figure 5.10.

We first describe how we compute $\Delta(x)$ for a fixed x and then describe how to track Lb and Rb as we vary x. For simplicity, we make the following two assumptions:

(A1) $\mathcal{C}_x$ is unimodal for all $x \in I$;

(A2) the point $\hat{x}$ is the unique minimum of $\mathcal{C}_x$;

(A3) Lb and Rb are simple curves.

In Section 5.6.4, we discuss how to relax these assumptions.

5.6.2 Computing $\Delta(x)$

Recall that we assume $\mathcal{C}_x$ to be unimodal with $\hat{x}$ as its unique minimum. Without loss of generality, assume that the edge e containing x is parallel to the x-axis, so Π_x is parallel to the yz-plane. We raise the value of z starting from $h(x)$ and stopping at the height of vertices of Σ_x until we find a vertex $\hat{v} = (v, h(v))$ such that $\phi_x(h(v)) \geq \phi_e$. We now know the edges of L_x and R_x that contain $\mathrm{Lb}(x)$ and $\mathrm{Rb}(x)$, so we can then compute the points themselves on those edges.

Next, we describe the procedure in more detail. Let $f_{1,x}, f_{2,x}, \cdots$ be the sequence of edges of R_x, ordered from left to right, and let $e_{i-1,x}, e_{i,x}$ be the endpoints of $f_{i,x}$; $e_{0,x} = x$. Recall that each edge $f_{i,x}$ is the intersection of a face f_i of Σ with the vertical plane Π_x, and each endpoint $e_{j,x}$ is $e_j \cap \Pi_x$ for some edge e_j of Σ. For each edge $f_{i,x}$, let $f^{\uparrow}_{i,x}$ be the semi-infinite trapezoid formed by the edge $f_{i,x}$ and the vertical rays in the $(+z)$-direction from the endpoints $e_{i-1,x}, e_{i,x}$ of $f_{i,x}$. For a value $z_0 \in \mathbb{R}$, we define the trapezoid $f^{\uparrow}_{i,x}(z_0)$ to be the intersection of $f^{\uparrow}_{i,x}$ with the halfspace $z \leq z_0$; $f^{\uparrow}_{i,x}(z_0)$ may be empty, or it may be a triangle; see Figure 5.10b. We define $A_{i,x}(z)$ to be the area of $f^{\uparrow}_{i,x}(z)$ and $P_{i,x}(z)$ to be the length of the bottom edge of $f^{\uparrow}_{i,x}(z)$, which is a portion of the edge $f_{i,x}$. Let $e_{i,x} = (x, a_i(x), b_i(x))$ denote the coordinates of $e_{i,x}$ as a function of x, and set $w_i(x) = a_i(x) - a_{i-1}(x)$ and $h_i(x) = b_i(x) - b_{i-1}(x)$. Then $A_{i,x}$ can be written as

$$A_{i,x}(z) = \begin{cases} 0 & z < b_{i-1}(x), \\ \frac{(z-b_{i-1}(x))^2 w_i(x)}{2h_i(x)} & b_{i-1}(x) < z < b_i(x), \\ w_i(x)(\frac{1}{2}h_i(x) + (z - b_i(x)) & b_i(x) < z. \end{cases} \tag{5.10}$$

Similarly $P_{i,x}$ can be expressed as

$$P_{i,x}(z) = \begin{cases} 0 & z < b_{i-1}(x), \\ \|f_i(x)\| \frac{(z-b_{i-1}(x))}{h_i(x)} & b_{i-1}(x) < z < b_i(x), \\ \|f_i(x)\| & b_i(x) < z. \end{cases} \tag{5.11}$$

We note that $P_{i,x}$ (resp. $A_{i,x}$) is a piecewise-linear (resp. piecewise-quadratic) function of z for a fixed x. We can define $F_{j,x}(z), P_{j,x}(z)$ and $A_{j,x}(z)$ for the edges $f_{j,x}$ of L_x as well. We can now express P_x and A_x as:

$$P_x(z) = \sum_i P_{i,x}(z) \quad \text{and} \quad A_x(z) = \sum_i A_{i,x}(z), \tag{5.12}$$

where the summation is taken over all edges of Σ_x that contain a point of height at most z. P_x and A_x are also piecewise-linear and piecewise-quadratic functions respectively.

Let $z_0 < z_1 < z_2 < \cdots$ be the heights of vertices of Σ_x. $P_x(z_0) = A_x(z_0) = 0$. Assuming $P_{i,x}(z_{i-1}), A_{i,x}(z_{i-1})$ have been computed, $P_{i,x}(z_i)$, $A_{i,x}(z_i)$ can be computed in $O(1)$ time using (5.10)–(5.12).

Let z_k be the first value for which $\phi_x(z_k) \geq \phi_e$. Since $P_{i,x}(z)$ (resp. $A_{i,x}(z)$) is a linear (resp. quadratic) function for $z \in (z_{k-1}, z_k)$, the value of $\Delta(x) \in (z_{k-1}, z_k]$ can be computed in $O(1)$ time by plugging these functional forms in (5.9), assuming the roots of a constant-degree polynomial can be computed in $O(1)$ time. Let $f_{L,x}$ (resp. $f_{R,x}$) be the edge of L_x (resp. R_x) at which the vertical sweep stopped. Then $\mathrm{Lb}(x)$ (resp. $\mathrm{Rb}(x)$) is the point on $f_{L,x}$ (resp. $f_{R,x}$) of height $\Delta(x)$ and can be computed in $O(1)$ time. The total time spent by the procedure is $O(|\mathcal{C}_x|)$. Hence, we obtain the following.

Lemma 5.1. For a given point $x \in \mathsf{P}$, $\Delta(x)$, $\mathrm{Lb}(x)$ and $\mathrm{Rb}(x)$ can be computed in $O(|\mathcal{C}_x|)$ time.

5.6.3 Computing the channel

We now describe our algorithm for computing the channel $\mathcal{C}$ assuming (A1) and (A2) hold. Recall that for any $x \in I$, the vertices of $\mathcal{C}_x$ are intersection points of the edges of Σ with the plane Π_x. Let $\Gamma_x = \langle \gamma_1, \cdots, \gamma_u \rangle$ denote the sequence of these edges, which implicitly define $\mathcal{C}_x$. We refer to Γ_x as the *combinatorial structure* of $\mathcal{C}_x$. We compute $\mathcal{C}$ by varying x continuously from x_0 to x_1 and maintaining $\mathcal{C}_x$. As x varies, $\mathcal{C}_x$ changes continuously, i.e., each vertex of $\mathcal{C}_x$ traces a curve, but the combinatorial structure Γ_x changes only at discrete values of x, called the *events*. The algorithm works by sweeping the line ℓ_x along P, stopping at events as we traverse P. As long as x lies on the same edge of P, ℓ_x simply translates. At vertices of P, where the sweep line ℓ_x shifts from one edge to the next one in P, the algorithm continues by rotating ℓ_x to make it normal to the next edge. We first describe how we sweep along an edge of P and then we describe how we sweep through a vertex of P.

Edges. Fix an edge $e \in \mathsf{P}$. Without loss of generality, assume that e is parameterized as $e : [0, 1] \to \mathbb{R}^2$. As we sweep along e and vary x, the left and right banks Lb, Rb trace curves that lie inside fixed faces of Σ and the remaining vertices of $\mathcal{C}_x$ trace the corresponding edges of Σ. The algorithm encounters the following two types of events at which Γ_x changes (see Figure 5.11):

1. the sweep line reaches an endpoint of an edge (u', v') of Γ_x that is a vertex of Σ, (Figure 5.11a) or

2. Lb or Rb intersects an edge e' (bounding the face containing it) of Σ (Figure 5.11b).

The first event results in one or more edges in $\mathcal{C}_x$ shrinking to the point v', and one or more new edges starting at v'. The second event results in either the addition and/or removal of an extremal edge in $\mathcal{C}_x$ (and thus insertion/deletion

(a) (b)

Figure 5.11: Two types of events in each: (a) ℓ reaches the endpoint of the edge (u', v'), (b) Lb, is marked in red, intersects the edge (u, v) of Σ.

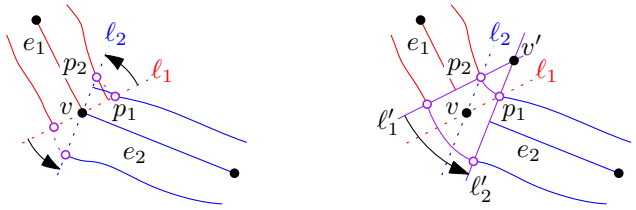
of an edge in Γ_x) depending on whether the height of the channel is increasing or decreasing. (For instance, in Figure 5.11b, $\mathcal{C}_x$ gains an edge at each of p_1 and p_2, but loses an edge at p_3.) We store the events in a priority queue $\mathcal{Q}$.

The first type of events are easy to detect, as they correspond to the vertices of Σ. The second type are more challenging, and we detect them as follows. By maintaining functions representing the area and wetted perimeter of $\mathcal{C}_x$ as a bivariate function of x and z (using (5.10)-(5.12)) and using Manning's equation we can express Δ as a function of x. We then compute when $\Delta(x)$ reaches the top or bottom boundary of the face containing Lb (resp. Rb). This step reduces to computing the zeros of a constant degree polynomial, which we recall we are assuming can be done in $O(1)$ time. These time instances correspond to the second type of event.

Initially $\mathcal{Q}$ contains the event corresponding to the first endpoint of e at $x = 0$. Additionally compute $\Delta(0)$, Lb(0) and Rb(0) using Lemma 5.1. In doing so, we compute the functions $A(x, z)$ and $P(x, z)$ which we store. Recall that $A(x, z)$ and $P(x, z)$ are sum of piecewise functions, one for each face in the profile. We store these functions in a data structure which supports adding or deleting the function for a face from $A(x, z)$ and $P(x, z)$. We can perform these operations in $O(\log n)$ time per insertion/deletion of a term in $A(x, z)$ and $P(x, z)$ by storing the terms in a height balanced tree.

Next we process the events in order, by removing the first event from the priority queue $\mathcal{Q}$. If it is the first type of event corresponding to a vertex v of Σ, we remove from Γ the edges of Σ that end at v and insert into Γ the edges of Σ that start at v. We add the other endpoints of the new edges to $\mathcal{Q}$ as new events. We also remove from $A(x, z)$ and $P(x, z)$ the terms corresponding to the old edges and add terms corresponding to the new edges. For example, in Figure 5.11a, we would remove the functions corresponding to f_1, and add the functions corresponding to f_2. If an edge of the face of Σ containing Lb or Rb changes, we also update the second type of events in $\mathcal{Q}$ as discussed above.

If the event corresponds to Lb or Rb reaching an edge of the terrain, either add the new face it crosses into and/or remove the face it crosses out of. We update Γ as well as $\mathcal{Q}$. We also update $A(x, z)$ and $P(x, z)$ accordingly. At p_2 Lb crosses into f_3, so we add the corresponding functions for this face. At p_3



(a) (b)

Figure 5.12: (a) The river profiles $\mathcal{C}_{(u,v)}$ and $\mathcal{C}_{(v,w)}$. (b) Rotating from ℓ'_0 to ℓ'_1, we get $\mathcal{C}_v$ delimited in purple.

Lb crosses out of f_3, so we delete the corresponding functions for this face.

We continue this process until we reach the event corresponding to $x = 1$, when the endpoint of e is reached. Let $|\mathcal{C}_e|$ be the number of total faces contained in the channel from $\mathcal{C}(0)$ to $\mathcal{C}(1)$. The curves Lb and Rb intersect an edge of Σ only a constant number of times. Therefore the total number of events is $O(|\mathcal{C}_e|)$, giving a total running time to sweep an edge of $O(|\mathcal{C}_e| \log |\mathcal{C}_e|)$.

Vertex. Let e_1 and e_2 be two consecutive edges of P with v as their common endpoint. Using the above algorithm, we can compute the channels $\mathcal{C}_{e_1}$ and $\mathcal{C}_{e_2}$. Since P is not C^1-continuous at v, $\mathcal{C}_{e_1}$ and $\mathcal{C}_{e_2}$ do not meet at the vertex v, i.e., left and right banks are not continuous curves at v (Figure 5.12a). Suppose P makes a left turn at v. Let ℓ_1 be the line normal to e_1 at v and ℓ_2 be the line normal to e_2 at v. A simple way to connect the left and right banks of the two channels is to rotate a line from ℓ_1 to ℓ_2 at v and maintain the left and right banks in the same manner as we did when we translated the line along an edge. But the difficulty with this approach is that the right bank becomes a self-intersecting curve as it traces "backwards" (Figure 5.12a). Instead, we process v as follows:

Let p_1 (resp. p_2) be the intersection point between ℓ_1 at v and the right bank of $\mathcal{C}_{e_2}$ (resp. ℓ_2 with the riverbank of $\mathcal{C}_{e_1}$) at v. We assume that p_i lies on e_i and on the same side of ℓ_i as e_i, for $i = 1, 2$. Then let ℓ'_1 (resp. ℓ'_2) be the line parallel to ℓ_1 at p_2 (resp. ℓ_2 at p_1), and v' be the intersection point of ℓ'_1 and ℓ'_2. We rotate a line at v' from ℓ'_1 to ℓ'_2 and maintain the left and right banks. This will connect the two banks of $\mathcal{C}_{e_1}$ and $\mathcal{C}_{e_2}$; see Figure 5.12b.

In a similar manner in which we swept along an edge, let $\mathcal{C}_{v',\theta}$ be the profile of v' with a line ℓ at angle θ. As we rotate the sweep line, we change the flow rate linearly from ϕ_{e_1} to ϕ_{e_2}. We will similarly maintain $A(\theta, z)$, $P(\theta, z)$ as a sum of functions on each face in the channel. Additionally when computing Manning's equation, as we rotate we will linearly interpolate between slope values as well as the roughness coefficients for the two edges. The main difference is now for each face f_i of Σ, $h_i(\theta, z)$ and $w_i(\theta, z)$ are not linear functions in θ as it was when we swept along an edge. However, we can still compute the events corresponding to the riverbank(s) crossing edges of faces assuming that the

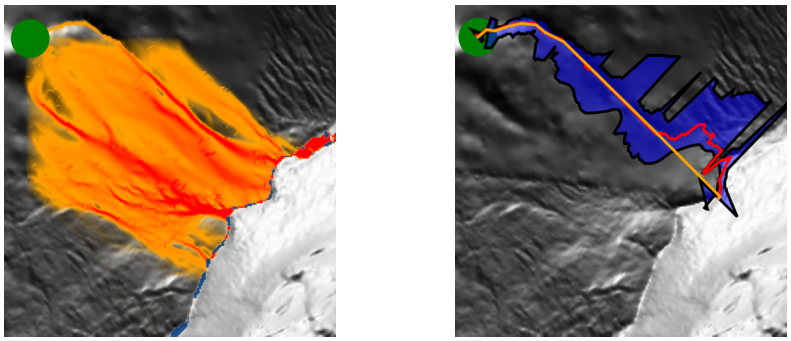
zeroes of corresponding functions can be computed in $O(1)$ time. Letting $|\mathcal{C}_v|$ denote the total number of faces in the channel obtained by sweeping at v, the total time spent is $O|\mathcal{C}_v| \log |\mathcal{C}_v|)$. Putting everything together, we obtain the following:

Theorem 5.5. Given a triangulation $\mathbb{M}$ with n vertices, a height function $h : \mathbb{M} \to \mathbb{R}$ which is linear on each face of $\mathbb{M}$ a path P in $\mathbb{M}$, such that assumption (A1) and (A2) hold, and a flow rate for each edge in P, we can compute the 2D flow network in time $O(|\mathcal{C}| \log |\mathcal{C}|)$ where $|\mathcal{C}|$ is the total number of faces in the channel obtained by sweeping along the edges and vertices of P.

5.6.4 Extensions

We will now show how to modify the algorithm when assumptions (A1) and (A2) do not hold, or when the 1D flow network is a forest instead of a simple path.

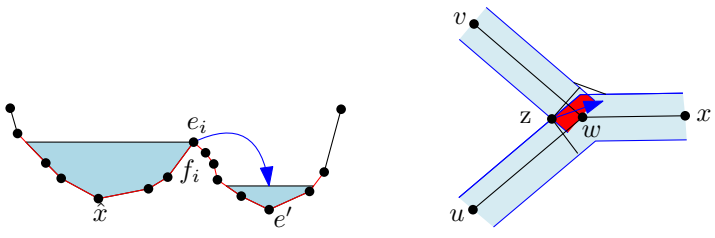
Relaxing assumption (A2). If (A2) does not hold but a local minimum lies near P, we can retract P to the minimum so that it satisfies (A2). However, this approach does not work when water is flowing over a 2D surface, and not along a channel, e.g., over a peak of a mountain or a plateau; see Figure 5.13b. In this case Manning's equation is not the right framework to compute the 2D channels. Instead one has to use a model that distributes water on a 2D surface. There has been some work on computing a 1D network of water from flow on flat areas [44], and these models need to be extended.



(a) (b)

Figure 5.13: Rain falls at the point marked in green down a mountainside. (a) The flow-rate function on vertices when water flows according to MFD model, with red indicating higher flow and orange less. (b) A 2D channel computed using Manning's equation; the 1D path is obained by computing water flow using the SFD model.

Relaxing assumption (A1). When (A1) does not hold, i.e. some cross-section $\mathcal{C}_x$ of the channel is not unimodal. $\mathcal{C}_x$ contains a local maximum in the



(a) (b)

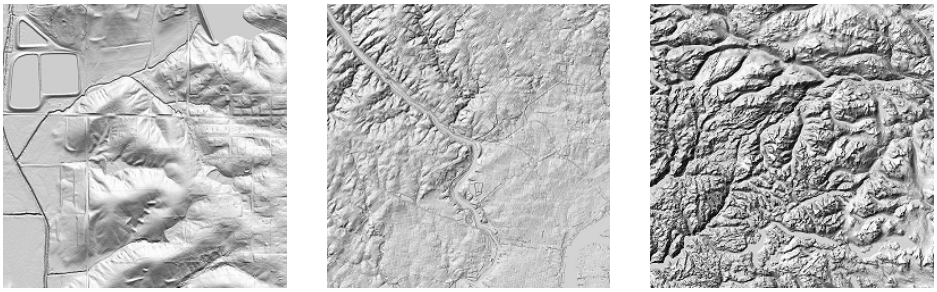
Figure 5.14: (a) A channel profile Σ_x with local maxima y. If the flow in the primary channel on the right causes the channel height to exceed $h(y)$, excess spills to the secondary channel on the left. (b) When two rivers merge, we recompute the extent of the channel for the region where their channels overlap. This region is highligted red, with the new sweep direction, detrmined by an weighted average of the sweep directions of the two rivers, shown as a blue arrow. The 2D channel is then taken to be the union of the initial 2D channels along with the newly computed channel.

profile. In this case, water flows over the ridge into a secondary channel. See Figure 5.14a.[3] We must now account for the decrease of the flow-rate in the primary channel as well as determining the height of water in the secondary channel. Our algorithm can be extended to handle this case by identifying the next local minima and treating it as a spill event. We omit the details from this version.

Forest 1D network. When the 1D flow network is a forest, we must consider how the channels corresponding to two different paths interact (Figure 5.9). If we assume the flow network is derived using a SFD model, that is, two rivers may merge together but they do not split, then here is an approach to handle multiple paths. We also assume for simplicity that only 2 rivers will touch in a given face of $\mathbb{M}$.

We begin by partitioning the forest into a set of paths. Whenever two downslope edges (u, w) and (v, w) in the forest merge at a common vertex w, the path with greater flow rate continues on. On this set of partitioned paths, we use the single channel algorithm, described above, to compute an initial 2D flow network. Next we determine the set of faces of $\mathbb{M}$ that the 2D channel from two paths intersects, see Figure 5.14b. For each such face, we find the region where the 2D channels overlap. In this region, take the flow rate to be the sum of the flow rates of the two paths. The flow direction is taken to be the weighted average of the flow direction of the two paths. We compute the 2D channel using Manning's equation, as we did in the 1D channel, with this new flow rate and direction.

[3]Here $|\mathcal{C}_x|$ contains all faces between the leftmost and rightmost river banks, i.e., all edges marked red.



(a) (b) (c)

Figure 5.15: A rendering of the three data sets used: (a) the Indiana dataset, (b) the Philadelhpia dataset and (c) the Norway dataset.

5.7 Experiments

In this section we present the experiments we have conducted on real terrains to demonstrate the efficacy and efficiency of our algorithms.

We have implemented the terrain-flow algorithm, described in Section 5.3, in C++, along with a simpler version of the 2D channel algorithm, described below. We ran the experiments on a Dell R730 with 2 Intel Xenon ®,E5-2640 v4 2.4GHz 25M Cache and 256 GB RAM running Linux.Since we use grid DEMs, every vertex has constant degree, so the flow-rate functions of edges will be very similar to those of vertices. We therefore focus on the flow rates of vertices.

Data sets. We study the performance of our algorithm on three publicly available grid DEMs:

(i) The *Indiana dataset*, (Figure 5.15a), a 0.89 mi^2 model of an area 0.5 mi northeast of Holland, Indiana, USA, extracted from the publicly available 5 ft resolution DEM of Indiana [65]. The dataset consists of 10^6 grid vertices, and the terrain is relatively flat.

(ii) The *Philadelphia dataset*, (Figure 5.15b), a 225 km^2 model of an urban area in the northwest area of Philadelphia, extracted from the publicly available 3 m resolution DEM of Pennsylvania[88]. The dataset consists of 2.5×10^7 vertices.

(iii) The *Norway dataset*, (Figure 5.15c), a 10000 km^2 model of a mountainous region located in the Jotunheimen National Park, Norway, exacted from the publicly available 10 m resolution DEM of Norway [31]. The dataset consists of 10^8 vertices.

5.7.1 Terrain-flow queries

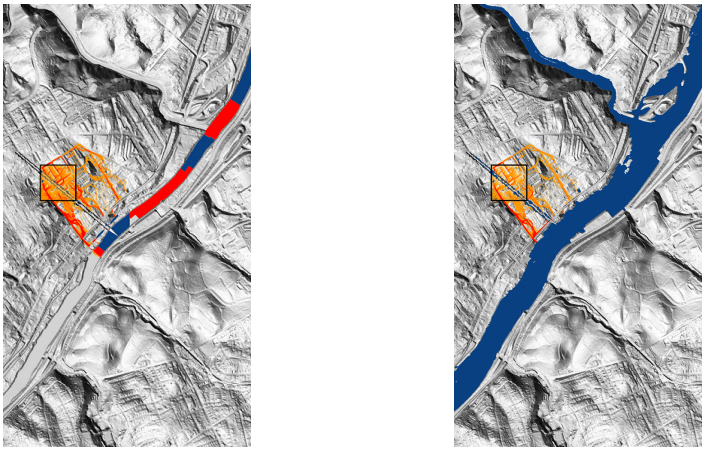
When performing the terrain-flow queries, as the water spreads out over the terrain it is common that a large number of vertices have an exceedingly small amount water flowing over them. As these small flows do not cause

any depressions to become filled and do not contribute meaningfully, in our implementation we apply a thresholding and treat a change in flow-rate of less than 10^{-4} times smaller than the initial flow rate of any vertex.

Complexity of flow functions. For the first set of tests, we considered queries over each dataset, varying the location and size of $|R|$. In each we considered the rain distribution to be rain falling uniformly over squares of side length 10, 100, and 1000 vertices in the DEM. For each dataset we chose 12 points that denote the top left point of the query square over which rain falls. For the queries on the Philadelphia and Norway datasets, we examined the flow-rate queries at each point with each of the three side lengths for a total of 36 queries on each. For the Indiana dataset, we do the same except that a square of side length 1000 encompasses the entire Indiana dataset. Therefore we only have one query for the largest rain region, for a total of 25 queries on the Indiana dataset. Note that holding the rain region constant, as the total volume of rain that falls increases (by either increasing the rain rate or duration) more depressions will begin to fill and spill, which in turn increases the output complexity. With this in mind, for each of the three datasets, we fixed the total volume of rain that falls over all queries, so the rate of rain falling over vertices in each query to be the same regardless of the region size. That is as the rain region increases in size, we decrease the amount it rains on each vertex in the region. However, since the scales of the datasets differ, we use a different total volume of rain for each.

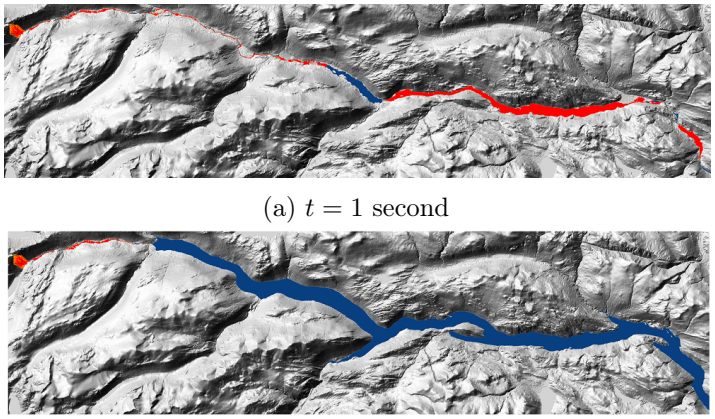
For the Philadelphia and Norway datasets, we considered the rate to be falling at $2\text{cm}/s$ over the largest square (9 km^2 and 100 km^2 respectively) for a duration of one hour, and scaled the rate accordingly on the smaller regions so the total rain-rate was equal, corresponding to a total rainfall of $6.48 \times 10^8 \text{m}^3$ and $7.2 \times 10^9 \text{m}^3$. For the Indiana dataset we considered rainfall to be falling uniformly for one hour, scaled over each query so that total rainfall in each was $1.27 \times 10^7 \text{m}^3$.

Figures 5.16 and 5.17 show an example output of one such query on Philadelphia and Norway respectively, with the flow-rates at one second and one hour after the rain starts. Flooded regions are marked in blue. On the Philadelphia dataset we show a zoomed-in portion of the affected region. Water flows from the rain region towards the Schuylkill river. It then flows along this channel until it reaches the boundary of the dataset and begins filling the depression corresponding to the river. At one second, only small patches of the river are flooded, which correspond to very shallow depressions. At one hour more of the river is flooded, but we omit these features to highlight the region over which it is raining. On the Norway dataset water falls over a region in the west and flows downstream along a valley until reaching the boundary on the east. This valley begins filling with water. For $t = 1$ second only small patches are flooded corresponding to very shallow depressions. For $t = 1$ hour enough rain has fallen to fill up a significant portion of the valley. Smaller depressions



(a) $t = 1$ second (b) $t = 1$ hour

Figure 5.16: An example query on Philadelphia at 1 second and 1 hour: rain is falling inside the square; flow rate is shown in orange and red, with darker red indicating a higher flow rate; flooded regions are shown in blue.



(a) $t = 1$ second

(b) $t = 1$ hour

Figure 5.17: An example query on Norway, at 1 second and 1 hour: rain is falling in the west; flow rate is shown in orange and red, with darker red indicating a higher flow rate; flooded regions are shown in blue.

are filled along the way as water traverses to the boundary. The total volume of water is large enough to fill these depressions and reach a local minima near the boundary. As time progresses, more depressions become full, often flooding regions corresponding to the valleys or rivers the water flowed along.

Figure 5.18 shows the distribution of the number of changes in the (non-zero) flow-rate functions over the queries. We see that for all sizes of $|R|$ the

most common size of $|\phi_v|$ is 1, having a single step. That is, most vertices with non-zero flow rate only have their flow rate increase once when water begins flowing over them, and then the rate stays constant until the rainfall ends. This intuitively makes sense, as for the flow-rate to change multiple times at a vertex, there must be multiple upstream saddles that delimit depressions becoming full. As the region size increases, the size of the output increase as well. Table 5.1 gives the total output complexity for each type of query, with the single 1000×1000 query on the Indiana dataset multiplied by 12, so that each represents 12 queries. As the rain falls over a larger initial region, despite being the same total volume of water, it can reach more vertices in the terrain. Additionally, the region over which it is raining has more complex flow-functions (e.g. there are more depressions filling initially which can spill and contribute the to flow rate of downstream regions.) For each dataset, as the region size increases, the maximum complexity of flow-rate functions increases, as does the number of functions with higher complexities. This also corresponds to an increase in the total output complexity.

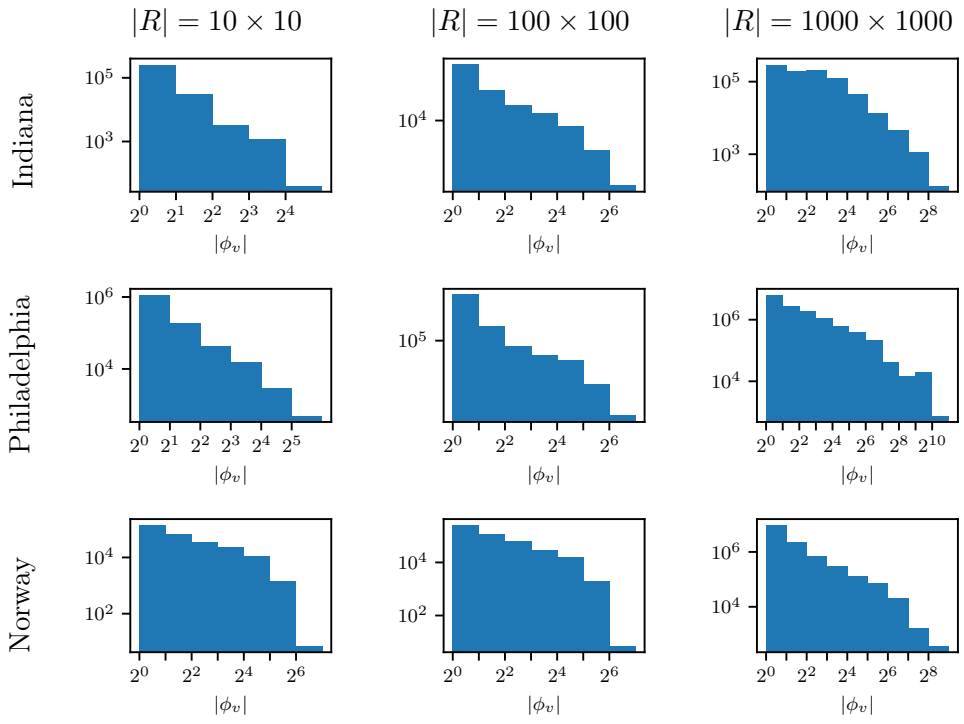


Figure 5.18: Distribution of the complexity of vertex flow rates over queries on Indiana, Philadelphia, and Norway with varying sizes of $|R|$. Both axes are shown in log scale.

Figure 5.19 shows the distribution of the step sizes of the flow-rate functions (i.e. the changes $|\delta_{v,t}|$.) In the Indiana and Philadelphia datasets, particularly

	10×10	100×100	1000×1000
Indiana	3.5×10^5	1.0×10^6	6.4×10^7
Philadelphia	2.0×10^6	3.8×10^6	1.1×10^8
Norway	9.8×10^5	1.6×10^6	3.0×10^7

Table 5.1: Total output complexity $|\phi|$ for each type of query.

in the smallest region of size 10×10 is a noticeable peak at the upper range of the change in flow-rates. This is primarily due to queries where the initial rain-fall mostly flows in a single direction, so the initial change in the flow-rate functions at time zero of all downstream vertices will be roughly equal to the total rate at which rain is falling. On these datasets, when the region increases to 100×100, there is a second peak of smaller step sizes. This most likely corresponds to lower depressions becoming full and spilling mainly in one direction. This is similar to queries on the smaller rain regions, but now more water goes in different directions. At the largest size, we note that the query on the Indiana dataset contains the entire terrain, while the Philadelphia dataset does not. For the Norway dataset, the water tends to quickly spread out more with few vertices containing near the total rainfall. The dataset is very mountainous, and one would expect water on hillsides to spread out quite a bit.

Hydrological conditioning. Terrain data often contains many small depressions, in part due to noise in the measurement of heights, which add noise to flow rates. It is therefore desirable to perform some hydrological conditioning to the raw input data of the terrain. One conditioning step is to remove small depressions by flooding them using topological persistence, as in [21, 44]. We perform the thresholding based on the volume persistence. Roughly speaking, this means given a threshold volume δ, we remove all depressions with a volume less than δ by pre-marking them as "flooded" in the preprocessing step. See [21, 44] for a more precise discussion of the procedure.

While we can use this simplification as described, and treat the small depressions as already having been flooded, this can change the resulting output if the removed depressions represented actual (albeit small) depressions in the terrain. Consider if there were many small divots which were all flooded inside a depression, the sum of the extra water within all of these may be enough to noticeably impact the water level in the depression. So we also consider a slight variation of our flow-query algorithm when we simplify the terrain. We initially treat the pre-flooded depressions as if they were flooded (e.g. when water flows into one of these depressions, it increases the spill-rate at the corresponding saddle.) However, when the sibling depression of one of these pre-flooded depressions becomes flooded (i.e. the parent depression becomes active) we subtract the volume of the pre-flooded depression from the

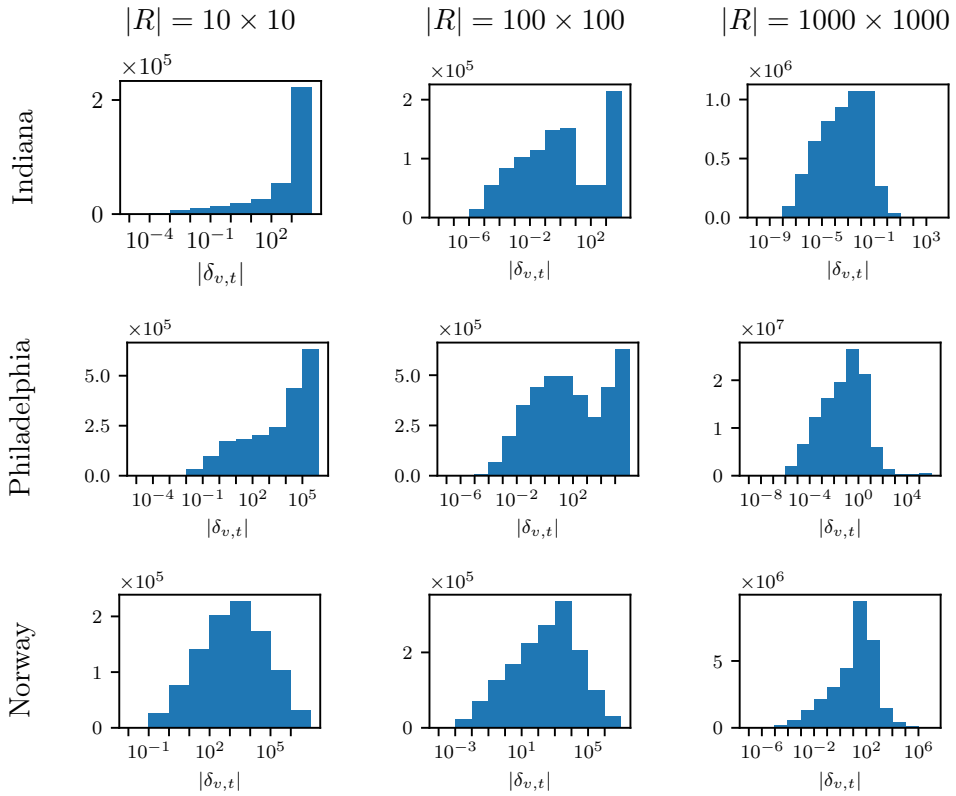
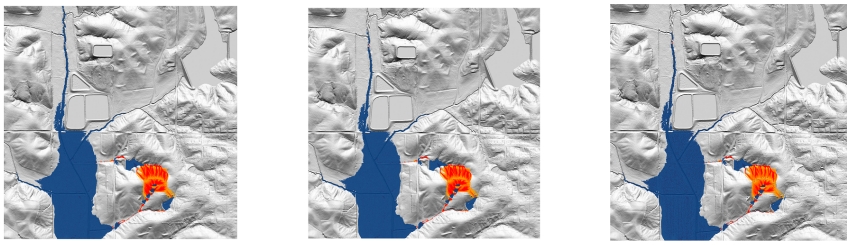


Figure 5.19: Distribution of step size of flow-rate functions (m^3/s) over Indiana, Philadelphia, and Norway with varying sizes $|R|$.

volume of rain in this new active depression. In essence, we loan water to the small depression in the preprocessing step, but reclaim the loaned water when the parent depression begins filling. See Figure 5.20 for an example. If we do not reclaim the water, the flooded portion in the north extends further. But if we do reclaim the water, the flooded portion is almost identical to the flood query on the original terrain, without removing small depressions.

We ran 10 flow-terrain queries on each dataset. Each query was run with 5 levels of simplification, removing depressions with volume 0, 0.1, 1, 10 and 100 m^3. For each, we considered rain falling over a square with side length 100, with the rain falling uniformly at a rate equal to that in the previous queries. We used the version of the algorithm that reclaims pre-flooded water. Table 5.2 shows how many depressions and total volume of rain that is pre-flooded at each threshold. For the Norway dataset the resolution is 10 m, with a height resolution of of 10 cm, so the smallest non-zero depression volume is 10 m^3. Therefore for the resulting tests, we only show the results using a threshold of 0, 10 and 100 m^3, as there are no depressions with volume less than 10m^3 to

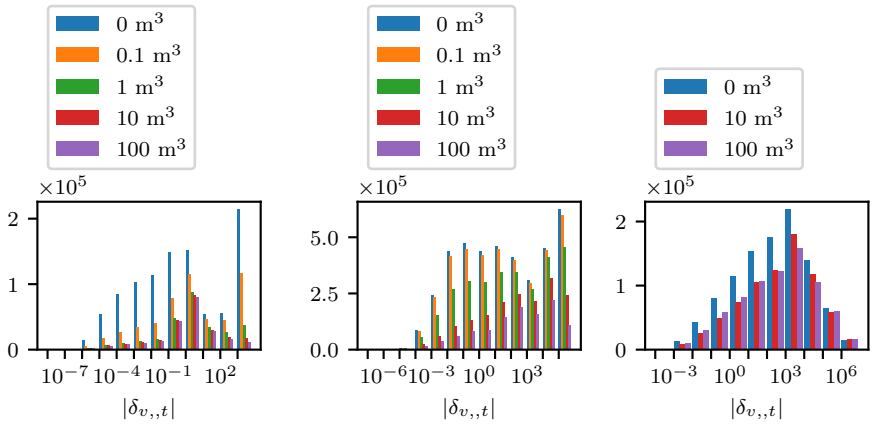


(a) (b) (c)

Figure 5.20: A terrain flow query on the Indiana dataset; (a) and (b) with depressions with persistence volume less than 100m^3 removed. (a) Pre-flooded water is treated as extra rainfall. (b) Pre-flooded water is reclaimed when a parent depression becomes active. (c) The same terrain flow query with no depressions removed.

be removed. Indiana and Philadelphia datasets show a diminishing number of depressions being removed as the threshold increases. Figure 5.22 shows the distribution of the complexity of the flow-rate functions for all the vertices with non-zero flow rate in the outputs. For each datasets, the number of depressions with large complexity decreases as we increase the threshold. Figure 5.21 shows the distribution of the step sizes in the flow rate functions as we increase the threshold. For each dataset, while there is an overall decrease in complexity, there is a significantly large decrease in the number of smaller step sizes. This reduction corresponds to tiny depressions which are no longer filling and spilling very small amounts of water because of hydrological conditioning.

While increasing the threshold decreases the overall output complexity and noise in the regions flooded, we want to balance this with the removal of real features. Figure 5.23 shows the flooded regions of a query on the Indiana dataset at three levels of thresholding, with depressions with volume 0, 10 and 100 m^3 removed respectively. We only mark a depression with volume below the threshold as flooded if its sibling depressions are also flooded to avoid marking all small depressions as flooded even where there is not water around them. They also correspond to pre-flooded depressions that we have begun reclaiming water from. There are several small depressions marked as flooded when we do not perform any thresholding. They are removed and not marked as flooded when we begin pre-flooding small depressions. However, pre-flooding depressions with volume 100m^3 also removes some small pools. So some care is needed when performing hydrological conditioning to remove the noisy small depressions while preserving actual features of the terrain.



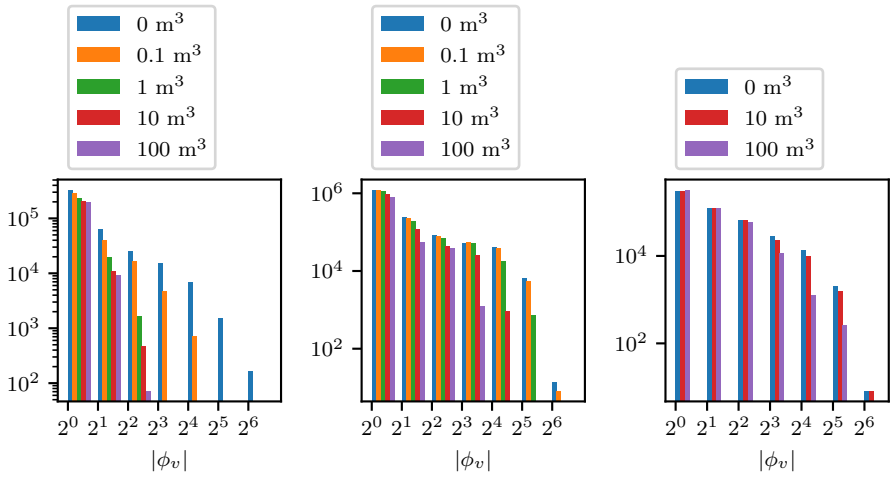
(a) Indiana (b) Philadelphia (c) Norway

Figure 5.21: Distribution of the step sizes of vertex flow rates in 10 queries on the Indiana, Philadelphia, and Norway data sets with varying sizes of depressions pruned from the terrain.

	Threshold (m^3)	0.1	1	10	100
Indiana	# Depressions	1.75×10^4	2.23×10^4	2.29×10^4	2.30×10^4
	Volume	4.27×10^2	1.70×10^3	3.15×10^3	6.23×10^3
Philadelphia	# Depressions	3.70×10^3	2.17×10^4	4.01×10^4	4.65×10^4
	Volume	3.37×10^2	9.23×10^3	7.31×10^4	2.40×10^5
Norway	# Depressions	0	0	6.08×10^3	1.71×10^4
	Volume	0	0	6.08×10^4	5.40×10^5

Table 5.2: Number and sum volume of depressions with volume less than thresholds.

Running time. The time to preprocess the Indiana, Philadelphia and Norway datasets was 2, 62 and 295 seconds, respectively. Figure 5.24 shows the running times of the queries with varying region size plotted against the output complexity $|\phi|$. For all sets of queries the running time is roughly linear in the size of the output. Interestingly, queries on the Philadelphia dataset had the largest output complexity.



(a) Indiana (b) Philadelphia (c) Norway

Figure 5.22: Distribution of complexity of vertex flow rates over vertices in 10 queries on the Indiana, Philadelphia, and Norway data sets with varying sizes of depressions pruned from the terrain.

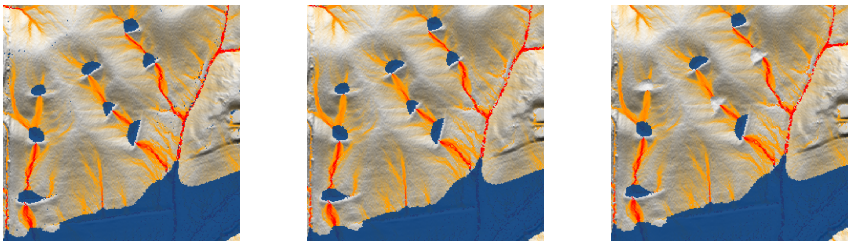


Figure 5.23: The same terrain flow query on the Indiana dataset with depressions with persistence volume less than 0, 10 and 100 m^3 pre-flooded.

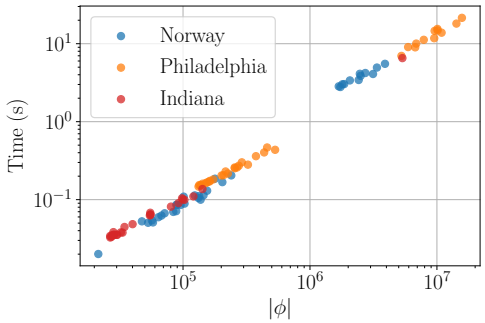
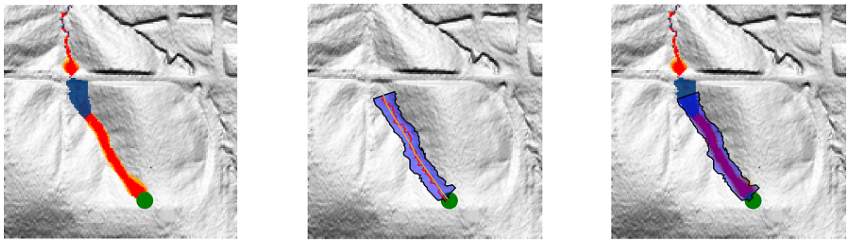


Figure 5.24: Running time of terrain flow queries against the output complexity of the flow-rate functions.



(a) (b) (c)

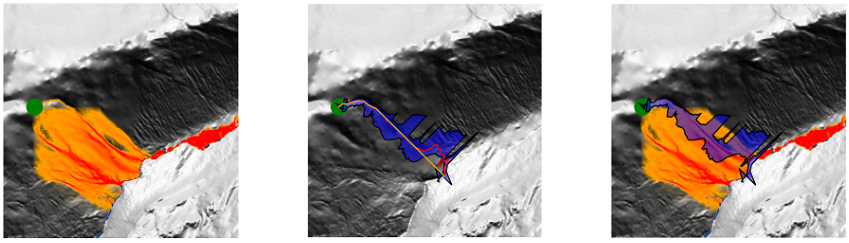
Figure 5.25: (a) the terrain flow query on Indiana with rain falling at the green point. (b) the 2D channel produced, taking the path to be the SFD flow from the green point, the smoothed SFD flow is marked in yellow, with the minima of each cross section marked in red. (c) the 2D channel overlayed on the results of the terrain flow query.

5.7.2 2D channel queries

To compute the 2D channel queries, we used a modified version of the algorithm described in Section 5.6 to avoid computing roots of polynomials. Instead of sweeping along the path and computing the boundary analytically, we discretized the path and computed the boundary of the channel at these discrete points. To this end, we first computed the path of water flowing under the SFD flow model, and fixed a flow rate. We then simplified the curve as described in [4], choosing a threshold ε and finding a subset of points along the polygonal curve that is ε-close to the original curve. Then at each point in this simplified curve, we determine the boundaries in a similar manner as in Section 5.6.2.

Figure 5.25 shows the resulting 2D channel query on the Indiana dataset. We also show the corresponding terrain-flow query, as well as an overlay of the two. The channel terminates where the path reaches a local minima. Note that the 2D channel is wider than the wetted region in the terrain flow query. Additionally if we were to increase or decrease the flow-rate the wetted region would stay the same, while the 2D channel would widen or narrow accordingly. This flow is along a valley of the terrain, so use of Manning's equation to compute the extent of the channel is appropriate.

Figure 5.26 shows the resulting 2D channel query on the Norway dataset. We also show the corresponding terrain-flow query, as well as an overlay of the two. We see here that Manning's equation is not the right framework for modeling the 2D flow on a locally concave or flat region, such as along a mountainside or plateau. Following the water down, we see two regions where the bank expands suddenly. This is due to the cross-section being flatter at those points. As we go down further, we see also that the path of steepest descent is no longer in line with the local minima of the cross section, marked in red.



(a) (b) (c)

Figure 5.26: (a) the terrain flow query on Norway with rain falling at the green point. (b) the 2D channel produced, taking the path to be the SFD flow from the green point, the smoothed SFD flow is marked in yellow, with the minima of each cross section marked in red. (c) the 2D channel overlayed on the results of the terrain flow query.

5.8 Conclusion

In this paper we presented algorithms for a number of flow-routing problems: We developed fast internal-memory as well as I/O-efficient algorithms for the terrain-flow query problem. Next, we presented a faster algorithm for computing the flow rate of only one vertex, after some preprocessing. Finally, given a flow path along the edges of Σ, we proposed an algorithm to determine the 2D channel along which water flows; our algorithm does not make any assumption about the geometry of the channel.

We conclude by mentioning a few directions for future work.

- While we consider the flow rate as a function of time, it only changes when the rain distribution changes or a spill event occurs. That is, the effects of such events are propagated to all reachable vertices instantaneously. While this assumption is reasonable for local effects and flash floods when a large volume of rain falls over a short duration, an interesting question is to make the model more general and account for the time it takes water to flow over the terrain.

- The model of extracting 2D channels leaves a number of open questions. For instance, if the 1D flow network is a forest then channels along different paths will interact. We give a heuristic for how to merge these channels, but a more systematic approach is needed. Another interesting question is how we construct a realistic 1D flow network from the edge flow functions.

- Can the simple geometric models used in this paper be combined with machine-learning techniques more accurately predict flood risk by incorporating historical flooding and river data. There has been some existing work on using machine learning to compute flood risk, e.g. [40, 76, 95],

but they typically focus on the height of water in rivers directly, and generally do not use simulations of the water flow.

Chapter 6

Learning to Find Hydrological Corrections

Abstract

High resolution Digital Elevation models, such as the (Big) grid terrain model of Denmark with more than 200 billion measurements, is a basic requirement for water flow modelling and flood risk analysis. However, a large number of modifications often need to be made to even very accurate terrain models, such as the Danish model, before they can be used in realistic flow modeling. These modifications include removal of bridges, which otherwise will act as dams in flow modeling, and inclusion of culverts that transport water underneath roads. In fact, the danish model is accompanied by a detailed set of hydrological corrections for the digital elevation model. However, producing these hydrological corrections is a very slow an expensive process, since it is to a large extent done manually and often with local input. This also means that corrections can be of varying quality. In this paper we propose a new algorithmic approach based on machine learning and convolutional neural networks for automatically detecting hydrological corrections for such large terrain data. Our model is able to detect most hydrological corrections known for the danish model and quite a few more that should have been included in the original list.

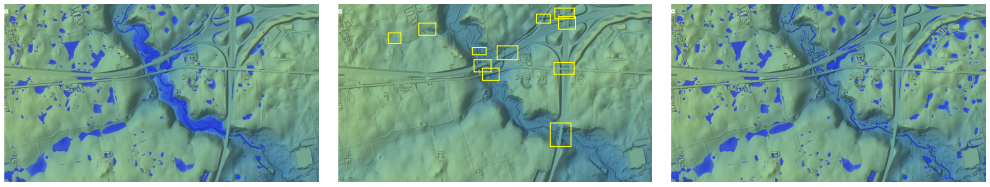
6.1 Introduction

High resolution Digital Elevation models, such as the grid terrain model of Denmark with more than 200 billion measurements available as part of the government's basic data program by the Agency for Data Supply and Efficiency (SDFE) [52], is a basic requirement for several terrain based applications like water flow modeling and flood risk analysis. However, a large number of modifications often need to be made to even very accurate terrain models, such as the Danish model, before they can be used in realistic flow modeling. These

modifications include removal of bridges, which otherwise will act as dams in flow modeling, and inclusion of culverts that transport water underneath roads. For this reason SDFE distribute a detailed set of hydrological corrections for the Denmark model. However, producing these corrections is a very slow and expensive process, since it is to a large extent done manually. This also means that these corrections are of varying quality. Moreover, there are terrain models for many countries that does not come with an official list of hydrological corrections hindering realistic applications of important hydrological analyses.

The most prominent application of terrain data is probably analyzing the risk of flooding, and the importance of this has only increased by efforts to mitigate the consequences of climate changes. Thus the high costs associated with extreme weather events occurring in densely populated areas has spurred an increased effort into developing new hydrological models and methods for analyzing how water flows across terrains in the case of heavy rain and increased sea levels. Consider a classic simulation of how water flows across a terrain in the event of rain fall. The result of a rain fall may be estimated by first adding some water to all (or subset of) the cells of the terrain model, and then simulating what happens as water flows down hill as follows: In each step water is moved from a one cell to a neighboring cell of lower height, usually the lowest neighboring cell. The simulation considers the cells in order of their height, with the highest cell considered first. In this process each cell may be annotated with the amount of water passing through it. This annotation of the cells is known as *flow accumulation* [44, 86] and is used reveal river networks and water ways by extracting the cells with high annotation. The cells that cannot get rid of the water reveal which depressions in the terrain that are flooded [44, 98]. For such a water flow simulation to produce useful and realistic results, the directions that water flow in the simulation has to (approximately) match how water flows over the surface in real life. However, a bridge recorded in the digital elevation model breaks this condition, because in real life the water would pass below it, while in the simulation this path is blocked. Hence, obstacles like a bridge that makes the water flow in a wrong direction in the simulation needs to be handled.

We loosely define a hydrological correction as any connected set of cells in the digital elevation model that relative to the surrounding cells has large heights, thus blocking the flow of water in the simulation, where in real life water would actually flow through these cells. A simple requirement for dealing with the problems created by hydrological corrections is to know where they are. For this reason, lists of hydrological corrections to digital elevation models are sometimes maintained together with the elevation model, and this list can be used to update the digital elevation model before any computations are performed. This can for instance be done by cutting the hydrological correction from the elevation model, replacing the heights of the cells comprising the hydrological correction with interpolated heights of the cells of the flow path the hydrological correction blocks. In Figure 6.1, a set of hydrological corrections



(a) Flash flood simulation without hydrological corrections.

(b) Hydrological corrections.

(c) Flash flood simulation with hydrological corrections.

Figure 6.1: Visualization of flow accumulation with and without considering hydrological corrections. Notice how water accumulates between bridges and on the high way instead of flowing away when hydrological corrections are not considered.

and the results of flash flood simulation [92] with and without considering these hydrological corrections are shown. This figure clearly shows that running analysis that do not consider hydrological corrections returns poor results.

Compiling a list of hydrological corrections is usually a manual process. In particular, the list of hydrological corrections for Denmark was made in a manual process where a group of people manually inspected orthophotos and digital elevation data, focusing primarily on intersections between road and river networks. Such an approach has several issues. First of all manual labor is slow, expensive, imprecise, and very often inconsistent since deciding whether something is in fact a hydrological correction is hard to pin down exactly. Furthermore, the manual process needs to be applied again every time the underlying data is changed, which happens continuously. In Denmark the full terrain model is completely updated every five years, each year updating one fifth of the model. Finally, intersections between road and river networks does not contain all hydrological corrections. For instance, trenches connected with pipes, small streams with small bridges, and tunnels cannot be found this way.

Problem Formulation The goal is to create an algorithm that automatically locates hydrological corrections in a digital elevation model, and thus automating and improving on the process above. The algorithm takes as input a digital elevation model, along with other supporting information, such as location of roads and rivers, and the output of water flow algorithms, and outputs a list of potential hydrological corrections including their positions and shapes. We note that we do not really care about very large hydrological corrections (like large bridges) since a list of these is readily available and easy to discover.

Related Work Carlson and Danner [38] used feature engineering and machine learning for automated detection of bridge-like objects. The approach they took was to manually design local feature maps around each cell in an elevation model and then applying the AdaBoost [55] machine learning algorithm on these features for a cell, trying to predict whether each cell is a part of a hydrological correction. The output of this is then processed by another algorithm that tries to locate the hydrological corrections by grouping areas with many cells predicted as hydrological corrections. The prediction of whether a given cell is part of hydrological correction or not, is based on five kinds of precomputed features. Carlsen and Danner create four local feature maps from the digital elevation model: the first feature is the raw height data, and the next tree features are output of different edge detectors, each based on a 3 x 3 neighborhood around the given cell. The final feature is a global feature, called a fill map, that is made from a water flow simulation of the entire area in consideration. From each of these feature maps, Carlson and Danner extract 102 features like min, max, mean, avg which totals 510 features per cell. The data used in [38] has approximately 6 million cells of 20 feet x 20 feet or 40 feet x 40 feet resolution. To get labeled data, they manually tagged 600 cells of the digital elevation model, 400 negative and 200 positive.

Our Approach Our approach for detecting hydrological corrections along with their position and shape is based on convolutional neural networks. The main ingredient in our algorithm is a convolutional neural network architecture [71] for supervised learning, heavily inspired by convolutional neural networks for image segmentation. Since terrains have high spatial locality, we believe convolutional neural nets that are designed for exactly this situation are the best available tool for the problem, alleviating the need for manually designing features. While the hand designed features designed by Carlson and Danner [38] may to some extent resemble the low level features a convolutional network automatically generate on the same data, convolutional networks are almost always better at learning useful discriminative features from data with spatial locality than people are at designing them.

The convolutional neural net we employ is designed to solve the problem on a fixed size tile. More formally, our tile neural network algorithm takes as input a fixed size tile, potentially with several layers of features, and outputs a new tile of the same size, mapping each cell of the input to the *probability* of whether this cell is a part of a hydrological correction. The prediction for each cell is based on the entire tile, allowing the neural network to learn to take advantage of any relevant features within a large area around each cell. Compared to the 3 x 3 cell neighborhood considered in [38], the neighborhoods we consider are orders of magnitude larger, even when we take into consideration that the cell size in the data we consider is an order of magnitude smaller. The data set of tiles for training the network is initially constructed from the list

of hydrological corrections maintained by SDFE, such that each hydrological correction is contained in at least one tile in the training data. We train a neural network to predict bit maps of the same size as the input tile, where the bits set in the bit map carve out the hydrological corrections contained in the tile.

We solve the full problem of locating all hydrological corrections in an digital elevation model with the tile algorithm as follows: We scan the digital elevation model, splitting it into overlapping fixed size tiles and apply the tile algorithm on these overlapping tiles of the input. The output from the algorithm for these tiles is then combined and used to list all the hydrological corrections and their shapes.

The data set we use are orders of magnitude larger than the data set considered in [38], containing approximately 200 hundred billion cells at a resolution of 0.4 meters by 0.4 meters and the list of hydrological corrections from SDFE just shy of 150 000 hydrological corrections. Hence, the results presented are incomparable to the results achieved by Carlson and Danner [38]. Also, since convolutional nets are considered the state of the art for most image recognition tasks, we have not compared our approach to theirs.

Our Results For the tile problem where the task is to predict the cells that are part of a hydrological correction within the tile, all variations of our algorithm obtain an area under ROC curve (AUC) score between 0.95 and 0.97. The AUC score of an algorithm is equal to the probability it will will rank a randomly chosen cell that is part of a hydrological correction higher than a randomly chosen cell that is not. We note that the bounding boxes of hydrological corrections in the official list maintained by SDFE has non-negligible variation both in terms of size and position when compared with the the digital elevation model and it is not possible to get perfect accuracy. With this in mind we believe our results for the tile problem are very good.

For the more general problem of listing all hydrological corrections in an arbitrary sized digital elevation model, we measure how well our algorithm detects the known hydrological corrections. However, we do not have a notion of true negative for this problem, as we do not output where there is not a hydrological correction. This means that we cannot compute an AUC score for this problem. Since an algorithm can propose an excessive amount of hydrological corrections it is important to consider both *precision:* the number of hydrological corrections found divided by the number of hydrological corrections suggested and *recall:* the number of hydrological corrections found divided by the number of hydrological corrections. Computing the precision and recall statistics is not completely trivial. We need to check if the shape of the hydrological correction output by our algorithm is *close* to the bounding box of a true hydrological correction. This is complicated by the fact that positions and sizes of ground truth hydrological corrections are noisy, and

there may be several hydrological corrections that are close to a proposed hydrological correction. For our applications recall is more important than precision, and we mainly trade off the two in favor of recall. All our algorithm variants achieve high recall, but the cost of this is quite low precision. This may make our results seems less impressive than we believe they are. There are hydrological corrections in the official list that are almost impossible to detect from the data we have. More importantly, after having analyzed a large number of the false positives, it is clear to us that many of the false positive output by our algorithm are in fact actual hydrological corrections that are just not part of the official list maintained by SDFE. It is clear to us that the precision of our algorithm is much higher than the tests on the official lists of hydrological corrections suggests, and is in fact a very good algorithm for the problem. Our algorithm has already been included in the commercial product SCALGO Live [93] where it is being used to detect hydrological corrections in Sweden that does not have an official list of hydrological corrections available.

Paper Outline In Section 6.2, we describe the data we use in more detail. In Section 6.3, we give a short description of the previous work that is the basis for our neural net architecture. In Section 6.4, we describe the neural net architecture we use for segmenting a tile into the cells that are part of hydrological corrections and cells that are not. We then describe in detail how we use this neural net algorithm that work for fixed size tiles to detect and output hydrological corrections for the entire digital elevation model.

In Section 6.5, we show the results of our experiments including several actual hydrological corrections output by our algorithm that are not a part of the list of hydrological corrections maintained by SDFE.

6.2 The Data

In this section we give descriptions of the data and how we construct our initial data set of tiles for the tile algorithm. The main data source we consider is the danish digital elevation model which is made and maintained by The Danish Agency for Data Supply and Efficiency. The digital elevation model is freely available and may be downloaded from [52]. The resolution of the model is 0.4 meters, meaning the digital elevation model contains a tiling of Denmark with $0.4m \times 0.4m$ cells each supplied with the height of that cell. This gives a model off approximately 200 billions cells, including parts of ocean which are not relevant for our task. Besides the digital elevation model, we have extracted road and river network grids from Denmark that we appropriately align with the digital elevation model. Finally, we have made flood computation maps that are also aligned with the digital elevation model. All these we may consider as extra layers of features.

Hydrological Correction Types The Danish Agency for Data Supply and Efficiency also makes and maintains a list of hydrological corrections of different types for Denmark. There are several different kind of hydrological corrections each with different characteristics. See [51] for the official information on hydrological corrections for the digital elevation model for Denmark including a few examples. The list of hydrological corrections also includes underground pipes that do not leave any marks on the digital elevation models. These are not possible to locate from the data available and we do not consider them. The list of these pipes are generated from a separate database that holds the information about such constructed pipe networks. The hydrological corrections we consider has two types that are named Horse Shoes and Lines respectively. There are approximately 22 000 Horse Shoe hydrological corrections, and 125 000 Line hydrological corrections in the list for the Denmark model.

Horse Shoes are hydrological corrections formed by three line segments connected as three sides of a rectangle which resembles the shape of a horse shoe. The Horse Show allow (or disallow) water flowing through an obstacle. A hydrological correction denoted as a line is represented as a single line segment that allow water to flow between the end points. In the data these lines can sometimes be connected into a poly line that lead the water from one end to the other. Such a poly line may be interpreted as one large hydrological correction instead of several small ones but that makes no difference for our purpose, since our algorithm tries to predict all the cells comprising a hydrological correction in the digital elevation model. We preprocess the hydrological corrections and keep only the Horse Shoe and Line corrections that take up more than one cell. Inspecting the hydrological corrections in the list compiled by SDFE, it is clear that the size and position relative to the actual corrections one can deduce from the digital elevation model is varying a great deal. It would of course have been more helpful for us if the true bounding box of every hydrological correction was available, but this is the data that we have. There also seems to be Line corrections that are sitting on top of completely flat areas, leaving no mark on the digital elevation model, and these essentially acts a noise for our model. They may actually be indicating an underground pipe, which we would prefer to remove from the data set, but we cannot deduce it from the information contained in the list of hydrological corrections. We note that the size of the individual hydrological corrections vary greatly, from less than one meter to the hydrological correction for the Great Belt Bridge which is close to 7000 meters. The distribution of hydrological correction lengths is shown in Figure 6.2.

6.3 Segmenting Tiles with Neural Networks

Image classification and segmentation algorithms using convolutional neural networks introduced in [71] for optical character recognition systems has flourished

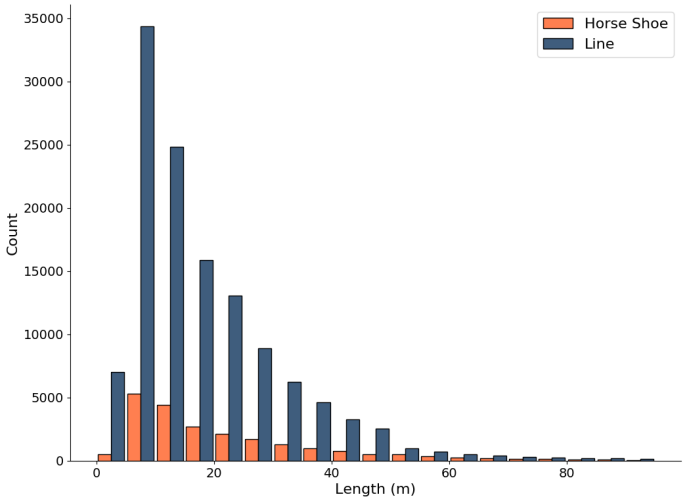


Figure 6.2: Distribution of hydrological corrections lengths.

greatly since the breakthrough paper by Krizhevsky, Sutskever and Hinton [70] that presented a convolutional neural net that outperformed all previous solutions on the famed ImageNet data set by a large margin. Convolutional networks are now the gold standard for several image recognition tasks including image segmentation where the task is to assign the pixels in an image into groups that comprise the relevant different object shown in the image.

As explained in the introduction the goal of our tile algorithm is to segment a tile into the cells that are part of a hydrological correction and the cells that are not. The very similar and more general problem of predicting pixel level segmentation maps from input images is a well studied problem in the Deep Learning Computer vision field, with a wide range of different models, having different trade-offs. On a high level, the main challenge, when moving from an object detection model (is there a dog in the image), to a pixel level segmentation model (return the pixels that comprise the dog), is the large class imbalance that stem from the fact that most objects only take up a small part of the input image, and the issue of integrating both high level information about the overall presence of an object and low level information about the precise geometric form of the object.

Techniques that tackle the first problem generally fall into two categories. First, there are methods that try to separate the problem into two subproblems: 1) constructing an algorithm that searches the input image for candidate locations for objects and 2) predicting pixel maps from crops of the image at these locations, making the problem significantly more class balanced for the second task [63]. Secondly, there are methods that try to modify the loss functions to suppress the contribution from pixels that are not part of any object [73].

For the second problem, the integration of both high and low level infor-

mation about objects, is typically handled through the creation of a feature pyramid. We can separate the feature pyramid network in two processes. First, the *encoder* which increase the channel dimension while decreasing the width and height for increasing layers. Second, the *decoder* which follow up with a decreasing channel dimension and increasing width and height, with concatenated features from the encoder layers. See Figure 6.3 for a depiction of this process. The hope is that the upsampled features will contain high level information about the presence of objects, while the concatenated channels from previous layers will contain precise information about possible edges of objects. Examples of this is found in U-Net [90] and Feature Pyramid Networks [72].

Our solution borrows ideas from all of these; We use the U-Net network architecture [90], the focal loss to suppress the contribution from low loss pixels from [73] and postprocess crops from candidate locations as in Mask R-CNN [63].

6.4 Complete Algorithm

In this section we describe our complete algorithm for detecting hydrological corrections in an arbitrary sized region in detail. We start by explaining how we solve the same problem on fixed sizes tiles and then explain how to use this tile algorithm to analyze an entire region. Our algorithm works even if the only feature layer we have is the digital elevation model. Adding more features is straight forward by adding extra layers to the input data aligned with the digital elevation model.

6.4.1 Tile Algorithm

Here we describe our algorithm for locating hydrological corrections in fixed size tiles. This algorithm is a convolutional neural network inspired by convolutional neural networks for image segmentation as described in Section 6.3.

A Data Sets for cell prediction on tiles In order to train our neural network to locate the hydrological corrections in a tile we need a data set $D = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ which we initially construct as follows. Each feature tile x_i is a fixed size tile with potentially several feature layers always including a layer with elevation data from the digital elevation model. The corresponding ground truth element y_i is a tile with one layer of the same size as the feature tile, encoding all the cells within the tile that are a part of a hydrological correction. This encoding is simply a bit mask, where a cell is given the value one if that cell is a part of a hydrological correction, and zero otherwise. We will refer to such a ground truth tile as a label mask.

For a given region of the digital elevation model to learn from we create a data set of tiles as follows. For every hydrological correction contained in

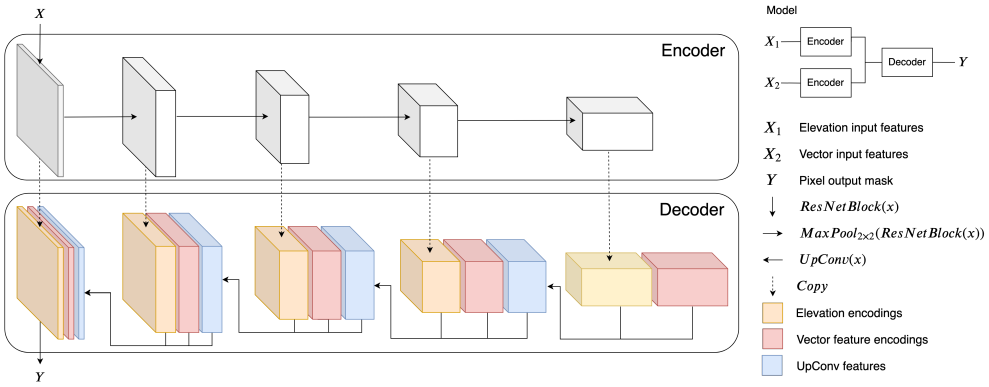


Figure 6.3: The model. Arrows represent operations, blocks represent data. We use an encoder for each input feature type, as depicted by the *model* subfigure. For example, when using only the elevation map as input feature, the vector feature encodings (red blocks) aren't present. The flow of data is as follows. First the input feature encodings are produced (grey blocks), though a series of operations, each of these decreasing the width and height of the features by a factor 2, while increasing the channel count by a factor 2. These are then used as input to the decoder indicated by the yellow/red blocks. The horizontal $UpConv$ operation then integrate lateral information from the encoder(s) along with more global information from the decoder, each of these *increasing* width and height by a factor 2, while *decreasing* the channel count by a factor 2. Lastly the channel count is reduced to 1 though a single $ResNet$ block and a sigmoid elementwise operation, producing Y, representing the probability that a pixel is part of a correction.

the region we construct a feature tile and a corresponding label mask with the hydrological correction placed at the center. The feature tile consists of 752×752 cells from the digital elevation model which we downsample to 376×376 cells of size 0.8×0.8 meters. This is done simply to save computation time. Tests have shown that it has no effect on the quality of our algorithm and speeds up our algorithms considerably. The same upsampling is performed on any extra feature layers included. This means that the tiles we consider are squares of approximately 300 by 300 meters. We note that the size of our tiles is so large that we can fit all hydrological corrections of interest. For a given data tile centered around a hydrological correction, we create a the label mask as follows. We start from the all zero tile of the same size as the feature tile and write one in each cell that intersect any of the hydrological corrections in the list of known hydrological corrections for the region, including the hydrological correction at the center of the tile. For the Horse Shoe hydrological corrections this is done by writing a one in each tile cell intersecting the rectangle defined by the Horse Shoe. The Line hydrological corrections are handled the same

way by adding a small width to the line segment making into a thin rectangle which is then processed like a Horse Shoe.

The Loss Function The goal of the training algorithm is to learn a function f that maps the input tiles x_i to the corresponding labels masks y_i, such that $f(x_i) \approx y_i$. See Figure 6.3 for a full specification of the neural net architecture we employ, which as mentioned earlier is heavily inspired by ideas from image segmentation and computer vision. The parameters of the neural net is fit by minimizing the sum of cross entropy loss between between the cells of predicted masks, $f(x)$ and the label mask y. We use the focal loss function [73] and a special weight map to counter-act the effect of class imbalance in the label masks the algorithm tries to predict. The weight map exist to address two concerns that are important for the quality of the final output of our algorithm:

- On average, only about 1 percent of the cells in a label mask are part of a hydrological correction and set to one. The rest are zeros. If the contribution to the loss from each cell that is part of a hydrological corrections is not higher than the loss associated to the zero valued cells that does not, the learned function becomes heavily biased towards not predicting cells to be part of hydrological corrections.

- Predicting the label of cells close to a hydrological correction correctly is much more important than getting all the vast amount of easily predicted cells far from a hydrological correction correct. Especially cells at the edge of the hydrological correction are important since down-stream processes vectorize hydrological corrections based on the contour of the prediction maps.

- Since hydrological corrections are varying in size, the contribution of large corrections to the loss is much higher than that of small hydrological corrections if each hydrological correction cell is penalized equally. If this is not handled the learning algorithm puts all emphasis on learning the large hydrological corrections and essentially ignores the small.

These concerns give rise to the following definition of the weight-map for scaling the loss to help the neural network focus on the most important areas of the input and to ensure that the algorithm learns to detect hydrological corrections of any size. The weight map W for a data point (x, y) is directly computed from the label mask y as follows.

$$W = \frac{1}{\sqrt{wh}} + E + B + L,$$

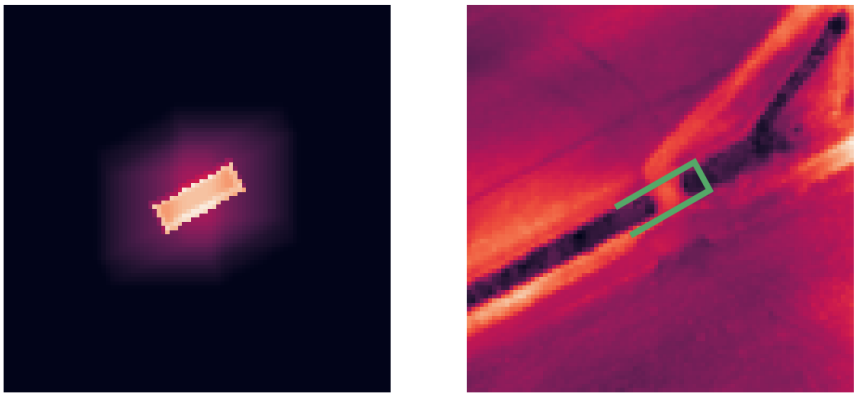


Figure 6.4: An example of the weight matrix of a single data point.

where w and h is respectively the width and the height of the tile counted in cells,

$$E = L * E_{\text{kernel}}, \quad B = E * B_{\text{kernel}} \tag{6.1}$$

$$L_{ij} = \begin{cases} 0 & \text{if } y_{ij} = 0 \\ \frac{1}{\text{number of cells in correction}} & \text{otherwise} \end{cases}$$

$$E_{\text{kernel}} = \begin{matrix} -\frac{1}{8} & -\frac{1}{8} & -\frac{1}{8} \\ -\frac{1}{8} & 1 & -\frac{1}{8} \\ -\frac{1}{8} & -\frac{1}{8} & -\frac{1}{8} \end{matrix}, \quad B_{\text{kernel}} = \begin{matrix} \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \end{matrix}$$

where $*$ is the convolutional operator. Note that cells that are neither part of a hydrological correction nor close to one are weighed as $1/\sqrt{wh}$. See Figure 6.4, for an example of the weight matrix.

The loss for the neural network on a predicted tile is the weighed sum of the losses over the cells of the tile, where the weights are specified by the weight map derived from the label mask. More formally, let $\hat{y}$ be the output mask predicted by the neural network on data point x with label mask y, and let L be the weight map induced by y. Finally let ℓ be the focal loss function from [73]. Then the loss of the network is defined as

$$\sum_{i,j} W_{i,j} \ell\left(\hat{y}_{i,j}, y_{i,j}\right)$$

Training stage Our learning algorithm follows the standard practice in image segmentation tasks to boost the number of samples and adding robustness to the learned function, by for each data point x_i considered we first extract a random crop from the feature tile, and then at randomly decide whether to flip the crop on both the horizontal and vertical axis. The same transformation is done on the label mask to predict, and this transformed data point and label mask is then used for training.

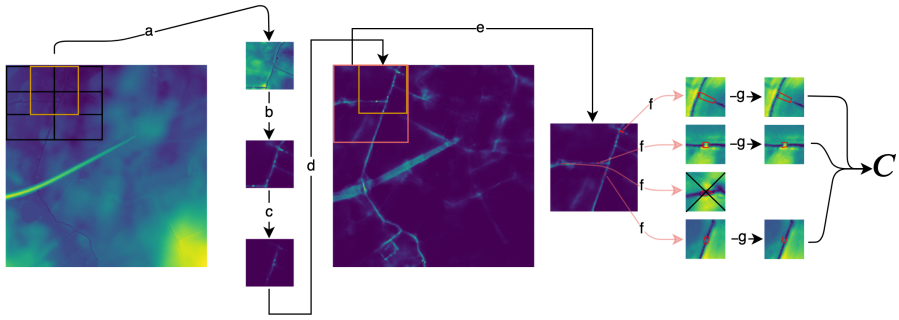


Figure 6.5: Prediction pipeline. (a) Tiles are extracted from the input rectangle in a strided fashion, such that each tile overlap other tiles 50 percent. (b) Cell-level probability of hydrological correction membership is predicted for each tile using our trained model for the tile problem defined in Section 6.4.1. (c) Each tile is then weighed through a monotone window function such that center pixels are weighted 1 and corner pixels are weighted 0. (d) Tiles are added to a probability map of the same size as the input, creating a cell-level probability map of the entire input rectangle. Weighing the tiles with a window function ensure independence of the actual tiling of the region. (e) As the probability map is filled, a different crop (red rectangle), independent of the tiling in (a), is extracted and polygons containing possible corrections are extracted using contours at a fixed threshold. (f) Each possible correction is evaluated and filtered according to different heuristics. In the above example a correction is filtered because the median probability within the polygon is too low. (g) Finally, polygons are converted to horse shoe shapes and added to the output.

6.4.2 Algorithm For General Region

While we were very successful at recognizing hydrological corrections in the tiles, as we show in Section 6.5, this does not solve the actual problem posed. Here we describe how our algorithm finds hydrological corrections in an arbitrary sized region given the algorithm we just described for fixed sized tiles. First, we cannot just tile the region arbitrarily into fixed size tiles, since that may split hydrological corrections in several pieces, making recognition of them impossible. Such a tiling may also cause an algorithm to report the same hydrological correction several times. Finally, there may be several hydrological corrections in one tile which complicates things further.

Without loss of generality we assume the input region to analyze is a, potentially very large, rectangle M including the necessary feature layers that corresponds with the features used in the tile neural network algorithm as described above. The basic idea is to use the tile algorithm on overlapping tiles to generate a new estimated probability map P, a rectangle of the same size of M, where each cell is associated with the probability of being a part of a

hydrological correction, exactly as we did in the tile algorithm. This large map P of probabilities is then processed by searching for areas of high probability and then applying several heuristics to determine if each area found this way is indeed a hydrological correction. Finally, if a hydrological correction has been found, we create a best fit Horse Shoe hydrological correction and add to the set of hydrological corrections that is output at the very end. The full process is visualized and described in more detail in Figure 6.5.

Formally our algorithm works as follows.

Creating Probability Map First we process the input M in a overlap-add fashion, extracting fixed sized crops that fit with the tile algorithm using a stride of s (we sample tiles, s cells apart), creating a set of fixed size tiles that we input into the tile algorithm and save into a list of predicted tiles $X_{i,j}$:

$$\begin{aligned} X_{ij} &= \text{nnet}\left(M_{i:(i+2s),j:(j+2s)}\right) \\ i &= \{0, s, 2s, \ldots, h-2s\} \\ j &= \{0, s, 2s, \ldots, w-2s\} \ , \end{aligned}$$

where nnet is the neural net we created for the tile problem, and w, h is the width and the height of the input rectangle M.

The tile predictions are then inserted in prediction map P as follows

$$P_{i:(i+2s),j:(j+2s)} += H \odot X_{ij}, \quad P \in \mathbb{R}^{h \times w} \ ,$$

where $H = hh^T, h_i = \frac{1}{2}\cos\left(\frac{\pi i}{s}\right)$ is a scaling map that ensures that mainly the predictions for the cells around the center of the tiles are added to the probability map, the further a cell is from the center from the center the more it is scaled down, and $\odot$ is element-wise multiplication. This finishes the construction of the map of probabilities P.

Extracting Hydrological Corrections From Probability Map To find the actual hydrological corrections and their shapes we start by creating a contour map on P, using a fixed threshold. Each contour polygon in this contour map represent a possible hydrological correction. We then filter these candidates using the following heuristics designed from manual inspection.

- Very small and very large contours are dropped, as most of them are false positives.
- Contours with small variance in the elevation data are dropped, as these are mostly false positives. They may also have negligent negative effect on water flow simulations.
- The median pixel probability is used as a threshold to control the tradeoff between precision and recall.

Outputting Horse Shoes Next step is to modify the shape of the contour polygons that the algorithm has decided constitute a hydrological correction. A given polygon found by our algorithm, describing a hydrological correction, is processed as follows. First we increase the size of the polygon by lowering the probability threshold used in the contour map to gain slightly more context to work with. Then we extract a crop C from the digital elevation model around the polygon. The cell heights in this tile of elevation data is then mapped to a probability distribution based on their heights with the lowest values getting the highest probabilities. We transform the elevation values in the crop C by negating the values, translating them such that the min height is zero and then normalizing by dividing each height value by the sum of heights.

We then sample points from this distribution and fit a Gaussian mixture model with two components to extract the two depressions that the hydrological correction is connecting. This is achieved by picking the mean of the components μ_1, μ_2 output by by the algorithm as the centers of the two depressions The line between μ_1 and μ_2 form the skeleton of the connection, while the width is extended in perpendicular direction to the line until it intersect the contour polygon. This give us the resulting horseshoe.

6.4.3 Bootstrapping our algorithm

As described above, the distribution of zeros and ones in the label masks the neural network for the tile problem must learn to predict, is highly unbalanced. This problem increases significantly when we need to predict hydrological corrections on the entire region considered. In this case the ratio of cells that are part of hydrological corrections is extremely small, much much smaller than in the training data set. The weight map and the focal loss we use to counter this problem help, however with the neural network learned on the initial data set the full algorithm is not able get high recall without predicting relatively many false positives. To counter this, we analyze the output of the first run of the complete algorithm, and sample new important tiles to learn from for the tile problem. This is achieved by creating tiles centered around false positives, where the predictions of the tile algorithm is close to the decision boundary we use to determine the contour map for the full algorithm. From manual inspection, the false positives far from the decision boundary tend to be actual corrections, revealing incompleteness in the set of manually created corrections. Including these as false positives in our tile algorithm would then make our algorithm worse. With these extra tiles defined we simply restart the training with the new data set, creating a new tile prediction neural network algorithm. We show the results for both in the next section.

6.5 Experiments and Results

In this section we describe our experiments. For training and evaluating our algorithm we use data from the island of Funen, which we have separated along the north-south axis in 2 splits. The training split, which comprise 70 percent of the total area and validation split which comprise 20 percent of the total area[1]. Funen has 9000 corrections, split in 5758 Lines and 3299 Horse Shoes. From these splits of Funen, we generate the following data sets:

`bl` Baseline experiment using only the digital elevation model and training only on tiles centered at the hydrological correction.

`bs` Bootstrap version of the baseline experiment (Section 6.4.3), with extra tiles centered at locations where the median probability of predicted polygons, using a trained baseline model, is within the range $.435 - .45$. We call these extra locations *bootstrapped* locations.

`ff` Like `bs` but with flash flood features [92]. These features may help the model since flash flood simulations accumulate water at the edge of a correction.

`vv` Like `bs` but with tiles rasterizing road and river vectors as extra layers of features. This is expected to help as most intersections between rivers and roads are hydrological corrections.

`bs_wz` Like `bs` with extra ground truth tiles from the island of Zealand. Zealand has 26651 extra hydrological corrections to consider.

`vv_wz` Like `vv` with extra ground truth tiles from the island of Zealand. Zealand has 26651 extra hydrological corrections to consider.

The neural network is implemented in Tensorflow, and training on all experiments is done using the ADAM [67] optimizer with a learning rate of .0001. We use a batch size of 32 and train on each data set for 50 epochs. After each epoch, the model is evaluated on the validation set of tiles and the model is saved if the cost has improved.

6.5.1 Results

We report results for both the tile algorithm and for the algorithm that detects hydrological corrections for an entire input region. For the tile algorithm, the validation set of tiles we consider is generated the same way as the training set just for a different region. This means tiles centered around a hydrological correction and tiles centered at the bootstrapped locations, except for `bs`, that

[1]We set aside the last 10 percent as a test set if we decide to do hyperparameter optimization as future work.

	AUC	mP	recall
`bl`	0.969	0.2231	**0.9126**
`bs`	0.9692	0.3056	0.853
`ff`	0.9542	0.3603	0.7845
`vv`	**0.977**	**0.5126**	0.7482
`vv_wz`	0.9761	0.3012	0.8498
`bs_wz`	0.9663	0.2624	0.8626

Table 6.1: Results for the different data sets. AUC is the area under ROC curve on the validation set of tiles that are generated the same way as the training set. mP is the average precision of the centroids of the generated polygons within the validation split of Funen, evaluated at a set of thresholds weighted by the change in recall, eg: $mP = \sum_n \left[(Recall_n - Recall_{n-1})/Precision_n\right]$. *Recall* is the maximal possible recall in the validation region by our full algorithm. That is, how many ground truth corrections are close to a proposed correction, when including all the proposed corrections from prediction pipeline.

only contain tiles centered at hydrological corrections. This validation score can be evaluated fast, since the area of the tiles is much smaller than the entire region. The quality of the tile algorithm for predicting which cells in a tile is part of a hydrological correction is evaluated using the area under ROC curve (AUC) score. Reporting pure accuracy is uninformative because of the large class imbalance.

For the full problem of locating hydrological corrections in an entire region, it is not possible to use AUC since the pipeline can propose any number of corrections and "true negatives" are not well defined. Instead we report *precision*: the ratio between the amount of proposed hydrological corrections close (center-distance < 25 meters) to a true correction (true positive), and all the proposed corrections (true positive + false positive), and *recall*: the ratio between the amount of proposed hydrological corrections close (center-distance < 25 meters) to a true correction (true positive), and the amount of hydrological corrections in the validation region (true positive + false negative). Notice that, in image segmentation tasks, one would usually apply mean intersection over union (mIoU) to determine if a proposed region corresponds to the ground truth shape, but as most of our ground truth hydrological corrections are line shaped, and therefore don't have a well defined area, we use distance to center instead.

The distinction between these two problems is important. We note that while we are ultimately only interested in the performance on the entire region, it is impractical to train on the entire region by including an excessive amount of extra tiles without any hydrological corrections. That would also add significantly to the label imbalance problem discussed in Section 6.4.

Detecting hydrological corrections on an entire region is a significantly

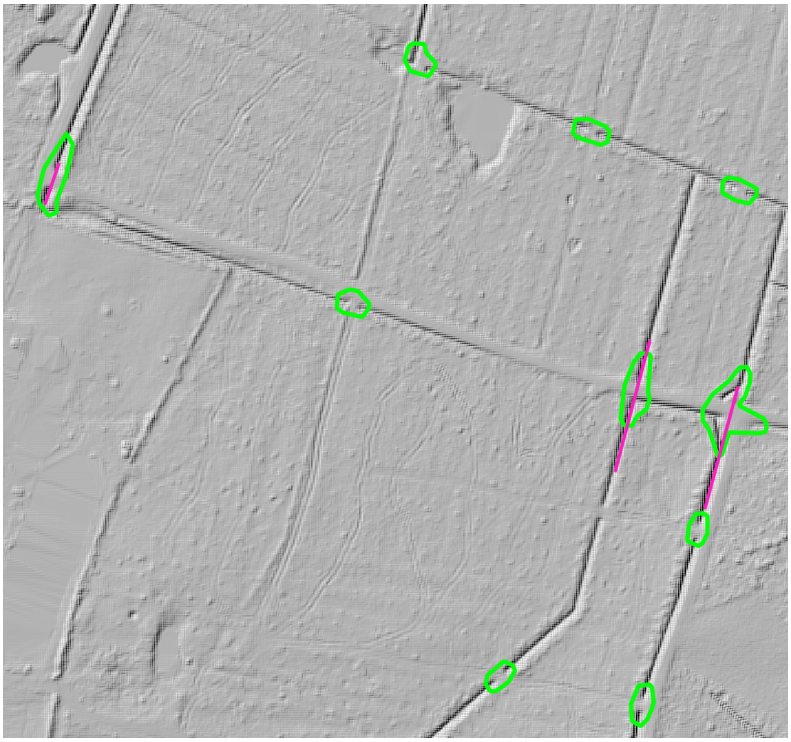


Figure 6.6: An example actual corrections not in the ground truth set. The purple lines are hydrological corrections from the official list of hydrological corrections and the green polygons are hydrological corrections proposed by our algorithm.

harder problem than predicting pixel probabilities on tiles, since hydrological corrections are very rare and the distribution of non-correction locations is suspected to be complex. As mentioned earlier, we try to handle this problem, by including non-correction tiles in the training and validation set, whose centers have median probability close to the decision boundary. Perhaps surprisingly, we do not sample false positive locations which have median probability above .45, since, manual inspection reveal that many such false positive locations, are in fact true positives. See Figure 6.6 for an example with several false positives that are actually true positives. Including these as non-correction tiles in the bootstrapping, would only degrade performance. Predicting hydrological corrections on the region of Funen takes between 30 minutes and an hour, depending on the number of predicted hydrological corrections, on a dual NVIDIA 1080ti GPU's and a Intel Xeon E5-1650 CPU. Training the neural network for the tile algorithm takes approximately six hours when only considering tiles from Funen.

The discrepancy between the problem of predicting cells in the validation tiles and predicting shapes of corrections on the entire region is shown in

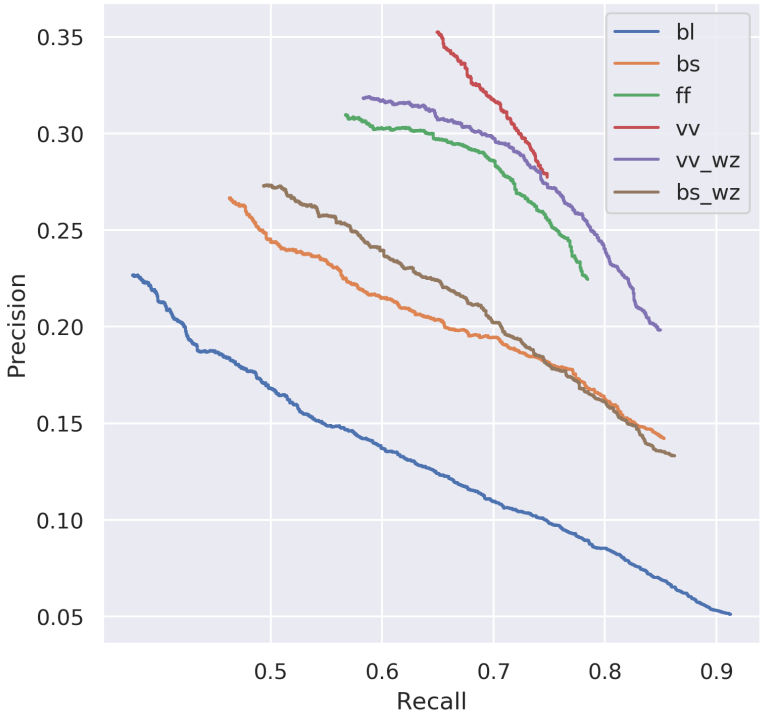


Figure 6.7: Precision/Recall curves. The precision and recall trade off is determined by the median pixel probability within each proposed polygon.

Table 6.1, where all experiments show good performance on the validation tiles; all within $0.95 - 0.97$ AUC. But we also see, that the performance on the validation tiles, does not necessarily translate to good performance on the entire region. For example, the `vv` experiment has the best AUC (0.977), but, when using this model in the prediction pipeline, it proposes too few hydrological corrections, resulting in lowest recall of all experiments. On the other hand, the baseline experiment actually have the best recall of all the experiments, but not very good precision. To gain better understanding of this relationship we show the different trade-off curves in Figure 6.7 for the full algorithm based on the neural network trained on the different data sets.

Inspecting Figure 6.7, it not clear that any model is ultimately better, since they all trade maximal recall for precision. One exception is `vv_wz` which achieves the same maximal recall as both `bs` experiments while maintaining a much better precision. Its also clear that including the bootstrapped locations improve precision significantly.

6.6 Conclusion and Future Work

In this paper we have described a new approach for detecting hydrological corrections that automates and improves the existing manual process. Our algorithms find almost all known hydrological corrections, and finds many more that should have been included in the list. The many missing hydrological corrections from the list maintained by SDFE is a problem both in terms of reporting how well an algorithm actually works but it is also a significant issue because the labels the algorithm learn from become noisy. As mentioned above, another issue with the official data is that the exact position and shape of the hydrological corrections in the list vary greatly when compared with the underlying digital elevation model. This makes both tile problem and the full problem harder. From our experiments our algorithm for the tile problem seems to be fairly robust to this problem. An industrial strength version of our algorithm have been implemented and incorporated into the commercial product of SCALGO Live [93]. This algorithm uses only digital elevation model which is often the only data available. Our algorithm is currently only used for Sweden that does not have any official list of known hydrological corrections. To help our algorithm we have acquired 1500 hydrological corrections from three different Swedish cities and added to the hydrological corrections from Denmark to train on. We use the bootstrapped version of our algorithm which gives the best tradeoff between precision and recall. The Swedish model has a resolution of $2m \times 2m$ and it took 3 days on a standard, single GPU work station to run our full algorithm on the entire country.

There are several avenues to explore for further improvement of our algorithm mainly to improve precision. We believe the most promising strategy is to improve the quality of the list of hydrological corrections since this will help all parts of the process, from the learning algorithm, to reporting more truthful precision and recall statistics. The latter is very important since it is hard to improve on our algorithm when the measure we use to compare algorithms is noisy. For this reason we are currently running a project where different experts and end users in the field are shown the false positives output by our algorithm and then has to decide by manual inspection whether the false positive is actually a hydrological correction or not.

Bibliography

[1] P. K. Agarwal, L. Arge, and A. Danner. *From Point Cloud to Grid DEM: A Scalable Approach*, pages 771–788. Springer, 2006. doi:10.1007/3-540-35589-8_48. 19

[2] P. K. Agarwal, L. Arge, and K. Yi. I/O-efficient construction of constrained Delaunay triangulations. In *Algorithms - ESA 2005, 13th Annual European Symposium, Proceedings*, volume 3669 of *Lecture Notes in Computer Science*, pages 355–366. Springer, 2005. doi:10.1007/11561071_33. 18, 57, 67

[3] P. K. Agarwal, L. Arge, and K. Yi. I/O-efficient batched union-find and its applications to terrain analysis. *ACM Transactions on Algorithms*, 7(1):11:1–11:21, December 2010. doi:10.1145/1868237.1868249. 19, 22

[4] P. K. Agarwal, S. Har-Peled, N. H. Mustafa, and Y. Wang. Near-linear time approximation algorithms for curve simplification. *Algorithmica*, 42(3):203–219, May 2005. doi:10.1007/s00453-005-1165-y. 123

[5] P. K. Agarwal, J. Matousek, and M. Sharir. On range searching with semialgebraic sets. II. *SIAM Journal on Computing*, 42(6):2039–2062, November 2013. doi:10.1137/120890855. 83

[6] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, September 1988. doi:10.1145/48529.48535. 8, 9, 21, 34, 40, 56, 80, 81

[7] D. Ajwani, U. Meyer, and V. Osipov. Breadth first search on massive graphs. In *The Shortest Path Problem, Proceedings of a DIMACS Workshop*, volume 74 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 291–307. DIMACS/AMS, 2006. doi:10.1090/dimacs/074/11. 35

[8] C. Alexander, L. Arge, P. K. Bøcher, M. Revsbæk, B. Sandel, J.-C. Svenning, C. Tsirogiannis, and J. Yang. Computing river floods using massive terrain data. In *Geographic Information Science - 9th International Conference, GIScience 2016, Montreal, QC, Canada, September 27-30, 2016,*

Proceedings, volume 9927 of *Lecture Notes in Computer Science*, pages 3–17. Springer, 2016. doi:10.1007/978-3-319-45738-3_1. 18

[9] A. Alexandrov, R. Bergmann, S. Ewen, J. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinländer, M. J. Sax, S. Schelter, M. Höger, K. Tzoumas, and D. Warneke. The Stratosphere platform for big data analytics. *VLDB J.*, 23(6):939–964, May 2014. doi:10.1007/s00778-014-0357-y. 35

[10] L. Arge. The buffer tree: A new technique for optimal I/O-algorithms. In *Proceedings of the 4th International Workshop on Algorithms and Data Structures WADS*, volume 955 of *Lecture Notes in Computer Science*, pages 334–345. Springer, 1995. doi:10.1007/3-540-60220-8_74. 9, 10, 21

[11] L. Arge. *External Memory Data Structures*, chapter 9, pages 313–357. Springer, 2002. ISBN 978-1-4615-0005-6. 8, 35

[12] L. Arge, G. S. Brodal, J. Truelsen, and C. Tsirogiannis. An optimal and practical cache-oblivious algorithm for computing multiresolution rasters. In *Algorithms - ESA 2013 - 21st Annual European Symposium, Sophia Antipolis, France, September 2-4, 2013. Proceedings*, volume 8125 of *Lecture Notes in Computer Science*, pages 61–72. Springer, 2013. doi:10.1007/978-3-642-40450-4_6. 28, 35, 37

[13] L. Arge, J. S. Chase, P. N. Halpin, L. Toma, J. S. Vitter, D. Urban, and R. Wickremesinghe. Efficient flow computation on massive grid terrain datasets. *GeoInformatica*, 7(4):283–313, December 2003. doi:10.1023/A:1025526421410. 20, 21, 22

[14] L. Arge, A. Grønlund, S. C. Svendsen, and J. Tranberg. Learning to find hydrological corrections. In *Proceedings of the 27th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 464–467. ACM, 2019. doi:10.1145/3347146.3359095. 5

[15] L. Arge, A. Grønlund, S. C. Svendsen, and J. Tranberg. Learning to find hydrological corrections. *CoRR*, abs/1909.07685, 2019. URL: http://arxiv.org/abs/1909.07685. 5, 25

[16] L. Arge, A. Lowe, S. C. Svendsen, and P. K. Agarwal. 1D and 2D flow routing on a terrain. *ACM Transansactions on Spatial Algorithms Systems*, 2021. In submission. 5

[17] L. Arge, M. Rav, S. Raza, and M. Revsbæk. I/O-efficient event based depression flood risk. In *Proceedings of the Ninteenth Workshop on Algorithm Engineering and Experiments, ALENEX*, pages 259–269. SIAM, 2017. doi:10.1137/1.9781611974768.21. 10, 23, 82

[18] L. Arge, M. Rav, M. Revsbæk, Y. Shin, and J. Yang. Sea-rise flooding on massive dynamic terrains. In *17th Scandinavian Symposium and Workshops on Algorithm Theory, SWAT 2020, June 22-24, 2020, Tórshavn, Faroe Islands*, volume 162 of *LIPIcs*, pages 6:1–6:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.SWAT.2020.6. 16, 18

[19] L. Arge, M. Rav, S. C. Svendsen, and J. Truelsen. External memory pipelining made easy with TPIE. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 319–324. IEEE, 2017. doi:10.1109/BigData.2017.8257940. 5, 26, 68

[20] L. Arge, M. Rav, S. C. Svendsen, and J. Truelsen. External memory pipelining made easy with TPIE. *CoRR*, abs/1710.10091, 2017. URL: http://arxiv.org/abs/1710.10091. 5

[21] L. Arge and M. Revsbæk. I/O-efficient contour tree simplification. In *Algorithms and Computation, 20th International Symposium, ISAAC*, volume 5878 of *Lecture Notes in Computer Science*, pages 1155–1165. Springer, 2009. doi:10.1007/978-3-642-10631-6_116. 19, 22, 86, 96, 118

[22] L. Arge, M. Revsbæk, and N. Zeh. I/O-efficient computation of water flow across a terrain. In *Proceedings of the 26th ACM Symposium on Computational Geometry*, pages 403–412. ACM, 2010. doi:10.1145/1810959.1811026. 16, 19, 23, 80, 82, 86

[23] L. Arge, Y. Shin, and C. Tsirogiannis. Computing floods caused by non-uniform sea-level rise. In *Proceedings of the Twentieth Workshop on Algorithm Engineering and Experiments, ALENEX*, pages 97–108. SIAM, 2018. doi:10.1137/1.9781611975055.9. 10, 18

[24] L. Arge, L. Toma, and J. Vitter. I/O-efficient algorithms for problems on grid-based terrains. *Journal of Experimental Algorithmics*, 6:1, December 2000. doi:10.1145/945394.945395. 82

[25] L. Arge, L. Toma, and J. S. Vitter. I/O-efficient algorithms for problems on grid-based terrains. *ACM Journal of Experimental Algorithmics*, 6:1, December 2001. doi:10.1145/945394.945395. 20, 28

[26] L. Arge, L. Toma, and N. Zeh. I/O-efficient topological sorting of planar DAGs. In *Proceedings of the Fifteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 85–93. Association for Computing Machinery, 2003. doi:10.1145/777412.777427. 10, 56

[27] L. Arge, J. Truelsen, and J. Yang. Simplifying massive planar subdivisions. In *2014 Proceedings of the Sixteenth Workshop on Algorithm Engineering*

and Experiments, ALENEX, pages 20–30. SIAM, 2014. doi:10.1137/1.9781611973198.3. 28, 35, 37

[28] L. Arge, F. van Walderveen, and N. Zeh. Multiway simple cycle separators and I/O-efficient algorithms for planar graphs. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 901–918. Society for Industrial and Applied Mathematics, 2013. doi:10.1137/1.9781611973105.65. 10, 56, 64

[29] L. Arge and N. Zeh. I/O-efficient strong connectivity and depth-first search for directed planar graphs. In *Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science*, pages 261–270. IEEE, 2003. doi:10.1109/SFCS.2003.1238200. 10, 56

[30] F. Aurenhammer and R. Klein. Voronoi diagrams. In *Handbook of Computational Geometry*, pages 201–290. North Holland / Elsevier, 2000. doi:10.1016/b978-044482537-7/50006-1. 18

[31] Norwegian Mapping Authority. Height DTM 10, 2013. URL: https://kartkatalog.geonorge.no/metadata/kartverket/dtm-10-terrengmodell-utm33/dddbb667-1303-4ac5-8640-7ec04c0e3918. 114

[32] M. J. Bannister, W. E. Devanny, D. Eppstein, and M. T. Goodrich. The Galois complexity of graph drawing: Why numerical solutions are ubiquitous for force-directed, spectral, and circle packing drawings. In *Graph Drawing*, volume 8871 of *Lecture Notes in Computer Science*, pages 149–161. Springer, 2014. doi:10.1007/978-3-662-45803-7_13. 57

[33] P. D. Bates and A. P. J. De Roo. A simple raster-based model for flood inundation simulation. *Journal of hydrology*, 236(1-2):54–77, September 2000. doi:10.1016/S0022-1694(00)00278-X. 81, 83

[34] A. Beckmann, R. Dementiev, and J. Singler. Building a parallel pipelined external memory algorithm library. In *23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS*, pages 1–10. IEEE, 2009. doi:10.1109/IPDPS.2009.5161001. 36

[35] T. Bingmann, M. Axtmann, E. Jöbstl, S. Lamm, H. C. Nguyen, A. Noe, S. Schlag, M. Stumpp, T. Sturm, and P. Sanders. Thrill: High-performance algorithmic distributed batch data processing with C++. In *2016 IEEE International Conference on Big Data, BigData*. IEEE, 2016. doi:10.1109/BigData.2016.7840603. 35

[36] G. S. Brodal, R. Fagerberg, and K. Vinther. Engineering a cache-oblivious sorting algorithm. *ACM Journal of Experimental Algorithmics*, 12:2.2:1–2.2:23, June 2007. doi:10.1145/1227161.1227164. 26

[37] G. S. Brodal and J. Katajainen. Worst-case external-memory priority queues. In *Proceedings of the 6th Scandinavian Workshop on Algorithm Theory, SWAT*, volume 1432 of *Lecture Notes in Computer Science*, pages 107–118. Springer, 1998. doi:10.1007/BFb0054359. 9, 96

[38] R. Carlson and A. Danner. Bridge detection in grid terrains and improved drainage enforcement. In *Proceedings of the 18th ACM SIGSPATIAL International Symposium on Advances in Geographic Information Systems, ACM-GIS*, pages 250–259. ACM, 2010. doi:10.1145/1869790.1869827. 25, 130, 131

[39] H. A. Carr, J. Snoeyink, and U. Axen. Computing contour trees in all dimensions. *Computational Geometry*, 24(2):75–94, February 2003. doi:10.1016/S0925-7721(02)00093-7. 19, 85, 86

[40] L.-C. Chang, H.-Y. Shen, and F.-J. Chang. Regional flood inundation nowcast using hybrid SOM and dynamic neural networks. *Journal of Hydrology*, 519:476–489, November 2014. doi:10.1016/j.jhydrol.2014.07.036. 124

[41] L. P. Chew. Constrained Delaunay triangulations. *Algorithmica*, 4(1):97–108, June 1989. doi:10.1007/BF01553881. 18

[42] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. Erik Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 139–149. ACM/SIAM, 1995. 9, 10

[43] K. L. Clarkson, D. Eppstein, G. L. Miller, C. Sturtivant, and S. Teng. Approximating center points with iterated Radon points. In *Proceedings of the Ninth Annual Symposium on Computational Geometry*, pages 91–98. Association for Computing Machinery, 1993. doi:10.1145/160985.161004. 68, 72

[44] A. Danner, T. Mølhave, K. Yi, P. K. Agarwal, L. Arge, and H. Mitásová. TerraStream: from elevation data to watershed hierarchies. In *Proceedings of the 15th ACM International Symposium on Geographic Information Systems, ACM-GIS 2007*, page 28. ACM, 2007. doi:10.1145/1341012.1341049. 19, 27, 112, 118, 128

[45] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, January 2008. doi:10.1145/1327452.1327492. 35

[46] R. Dementiev, L. Kettner, and P. Sanders. STXXL: standard template library for XXL data sets. *Software - Practice and Experience*, 38(6):589–637, August 2008. doi:10.1002/spe.844. 26, 28, 34, 35

[47] R. Dementiev, P. Sanders, D. Schultes, and J. F. Sibeyn. Engineering an external memory minimum spanning tree algorithm. In *Exploring New Frontiers of Theoretical Informatics*, volume 155 of *IFIP*, pages 195–208. Kluwer/Springer, 2004. doi:10.1007/1-4020-8141-3_17. 35

[48] S. Dong, Y. T. Lee, and K. Quanrud. Computing circle packing representations of planar graphs. In *Proceedings of the Thirty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2860–2875. Society for Industrial and Applied Mathematics, 2020. doi:10.1137/1.9781611975994.174. 57

[49] H. Edelsbrunner, J. Harer, and A. Zomorodian. Hierarchical morse complexes for piecewise linear 2-manifolds. In *Proceedings of the Seventeenth Annual Symposium on Computational Geometry*, pages 70–79. ACM, 2001. doi:10.1145/378583.378626. 13, 86

[50] The Danish Agency for Data Supply and Efficiency. Danmarks højdemodel, 2021. URL: https://sdfe.dk/hent-data/danmarks-hoejdemodel/. 3, 6, 22, 25, 57, 68

[51] The Danish Agency for Data Supply and Efficiency. GeoDanmark specifikation 6.0, 2021. URL: http://geodanmark.nu/Spec6/HTML5/DK/StartHer.htm. 29, 133

[52] The Danish Agency for Data Supply and Efficiency. Styrelsen for dataforsyning og effektivering, 2021. URL: https://sdfe.dk. 127, 132

[53] G. N. Frederickson. Fast algorithms for shortest paths in planar graphs, with applications. *SIAM Journal on Computing*, 16(6):1004–1022, December 1987. doi:10.1137/0216064. 10, 56, 64

[54] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615, July 1987. doi:10.1145/28869.28874. 94

[55] Y. Freund and R. E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119–139, August 1997. doi:10.1006/jcss.1997.1504. 25, 130

[56] H. N. Gabow and R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of computer and system sciences*, 30(2):209–221, April 1985. doi:10.1016/0022-0000(85)90014-5. 94

[57] GeoDanmark. Det hydrologiske tilpasningslag, 2021. URL: https://www.geodanmark.dk/wp-content/uploads/2019/10/One-Page_Hydro.pdf. 25

[58] M. T. Goodrich, J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *34th Annual Symposium on Foundations of Computer Science*, pages 714–723. IEEE, 1993. doi:10.1109/SFCS.1993.366816. 18

[59] L. J. Guibas, D. E. Knuth, and M. Sharir. Randomized incremental construction of Delaunay and Voronoi diagrams. *Algorithmica*, 7(4):381–413, June 1992. doi:10.1007/BF01758770. 18

[60] S. Har-Peled. *Geometric Approximation Algorithms*. Mathematical surveys and monographs. American Mathematical Society, 2011. ISBN 978-0-8218-4911-8. 60

[61] S. Har-Peled and M. Sharir. Relative (p, ε)-approximations in geometry. *Discrete & Computational Geometry*, 45(3):462–496, February 2011. doi:10.1007/s00454-010-9248-1. 60, 61

[62] H. J. Haverkort and J. Janssen. Simple I/O-efficient flow accumulation on grid terrains. *CoRR*, abs/1211.1857, 2012. URL: http://arxiv.org/abs/1211.1857. 21, 27, 28, 58, 67, 68

[63] K. He, G. Gkioxari, P. Dollár, and R. B. Girshick. Mask R-CNN. In *IEEE International Conference on Computer Vision, ICCV*, pages 2980–2988. IEEE, 2017. doi:10.1109/ICCV.2017.322. 134, 135

[64] J. Holm, E. Rotenberg, and M. Thorup. Planar reachability in linear space and constant time. In *2015 IEEE 56th Annual Sympos. Foundations of Computer Science*, pages 370–389. IEEE, 2015. doi:10.1109/FOCS.2015.30. 101, 105

[65] Indiana Spatial Data Portal. Indiana orthophotography (RGBI), LiDAR and elevation, 2013. URL: http://gis.iu.edu/datasetInfo/statewide/in_2011.php. 114

[66] M. Isenburg, Y. Liu, J. R. Shewchuk, J. Snoeyink, and T. Thirion. Generating raster DEM from mass points via TIN streaming. In *4th International Conference on Geographic Information Science, GIScience*, volume 4197 of *Lecture Notes in Computer Science*, pages 186–198. Springer, 2006. doi:10.1007/11863939_13. 19

[67] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. In *Conference Track Proceedings of the 3rd International Conference on Learning Representations, ICLR*, 2015. URL: http://arxiv.org/abs/1412.6980. 142

[68] P. Koebe. Kontaktprobleme der konformen Abbildung. *Ber. Sächs. Akad. Wiss. Leipzig, Math.-Phys. Kl.*, 88:141–164, 1936. 58

[69] M. Kreveld, R. Oostrum, C. Bajaj, V. Pascucci, and D. Schikore. Contour trees and small seed sets for isosurface traversal. In *Proceedings of the Thirteenth Annual Symposium on Computational Geometry*, pages 212–220. ACM, 1997. doi:10.1145/262839.269238. 85

[70] A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, May 2017. doi:10.1145/3065386. 134

[71] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998. doi:10.1109/5.726791. 130, 133

[72] T. Lin, P. Dollár, R. B. Girshick, K. He, B. Hariharan, and S. J. Belongie. Feature pyramid networks for object detection. In *Proceedings of the 2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR*, pages 936–944. IEEE, 2017. doi:10.1109/CVPR.2017.106. 135

[73] T. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár. Focal loss for dense object detection. In *IEEE International Conference on Computer Vision, ICCV*, pages 2999–3007. IEEE, 2017. doi:10.1109/ICCV.2017.324. 134, 135, 137, 138

[74] R. Lipton and R. Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, July 1979. doi:10.1137/0136016. 56

[75] Y. Liu and J. Snoeyink. Flooding triangulated terrain. In *Developments in Spatial Data Handling, 11th International Symposium on Spatial Data Handling*, pages 137–148. Springer, 2004. doi:10.1007/3-540-26772-7_11. 16, 17, 23, 81

[76] A. K. Lohani, N. K. Goel, and K. K. S. Bhatia. Improving real time flood forecasting using fuzzy inference system. *Journal of Hydrology*, 509:25–41, February 2014. doi:10.1016/j.jhydrol.2013.11.021. 124

[77] A. Lowe and P. K. Agarwal. Flood-risk analysis on terrains under the multiflow-direction model. *ACM Transactions on Spatial Algorithms Systems*, 5(4):26:1–26:27, September 2019. doi:10.1145/3340707. 10, 11, 17, 23, 24, 28, 80, 82, 83, 84, 86, 96

[78] A. Lowe, P. K. Agarwal, and M. Rav. Flood-risk analysis on terrains. *Communications of the ACM*, 63(9):94–102, September 2020. doi:10.1145/3410413. 16, 82

[79] A. Lowe, S. C. Svendsen, P. K. Agarwal, and L. Arge. 1D and 2D flow routing on a terrain. In *Proceedings of the 28th International Conference*

on Advances in Geographic Information Systems, SIGSPATIAL, pages 5–14. ACM, 2020. doi:10.1145/3397536.3422269. 5, 83

[80] A. Maheshwari and N. Zeh. I/O-efficient planar separators. *SIAM Journal on Computing*, 38(3):767–801, May 2008. doi:10.1137/S0097539705446925. 10, 56

[81] R. Manning. On the flow of water in open channels and pipes. *Transactions of the Institution of Civil Engineers of Ireland*, pages 161–207, 1891. 29, 83, 107

[82] U. Meyer and V. Osipov. Design and implementation of a practical i/o-efficient shortest paths algorithm. In *Proceedings of the Eleventh Workshop on Algorithm Engineering and Experiments, ALENEX*, pages 85–96. SIAM, 2009. doi:10.1137/1.9781611972894.9. 35

[83] G. L. Miller, D. Talmor, S. Teng, and N. Walkington. A Delaunay based numerical method for three dimensions: Generation, formulation, and partition. In *Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing*, pages 683–692. Association for Computing Machinery, 1995. doi:10.1145/225058.225286. 57, 67

[84] G. L. Miller, S. Teng, W. Thurston, and S. A. Vavasis. Separators for sphere-packings and nearest neighbor graphs. *Journal of the ACM*, 44(1):1–29, January 1997. doi:10.1145/256292.256294. 57, 58, 59, 68, 72

[85] T. Mølhave. Using TPIE for processing massive data sets in C++. *ACM SIGSPATIAL Special*, 4(2):24–27, July 2012. doi:10.1145/2367574.2367579. 26, 34, 35

[86] J. F. O'Callaghan and D. Mark. The extraction of drainage networks from digital elevation data. *Computer Vision, Graphics, and Image Processing*, 27:323–344, August 1984. doi:10.1016/S0734-189X(84)80011-0. 20, 128

[87] G. L. Orick, K. Stephenson, and C. Collins. A linearized circle packing algorithm. *Computational Geometry*, 64:13–29, August 2017. doi:10.1016/j.comgeo.2017.03.002. 57

[88] Pennsylvania Spatial Data Access. PAMAP program DEM mosaics by lidar delivery zones, 2008. URL: http://www.pasda.psu.edu/uci/SearchResults.aspx?Keyword=PAMAP. 114

[89] M. Rav, A. Lowe, and P. K. Agarwal. Flood risk analysis on terrains. *ACM Transansactions on Spatial Algorithms Systems*, 5(1):2:1–2:31, June 2019. doi:10.1145/3295459. 23, 80, 82, 84, 101, 104, 105

[90] O. Ronneberger, P. Fischer, and T. Brox. U-Net: Convolutional networks for biomedical image segmentation. In *Proceedings of the 18th International Conference on Medical Image Computing and Computer-Assisted Intervention, MICCAI*, volume 9351 of *Lecture Notes in Computer Science*, pages 234–241. Springer, 2015. doi:10.1007/978-3-319-24574-4_28. 135

[91] SCALGO, 2019. URL: www.scalgo.com. 106

[92] SCALGO. Flash flood map, 2021. URL: https://scalgo.com/en-US/scalgo-live-documentation/analysis/flash-flood-map. 129, 142

[93] SCALGO. SCALGO live, 2021. URL: https://scalgo.com/live/. 132, 146

[94] S. C. Svendsen. Practical I/O-efficient multiway separators. *CoRR*, abs/2107.02570, 2021. URL: https://arxiv.org/abs/2107.02570. 5

[95] M. S. Tehrany, B. Pradhan, S. Mansor, and N. Ahmad. Flood susceptibility assessment using GIS-based support vector machine model with different kernel types. *Catena*, 125:91–101, February 2015. doi:10.1016/j.catena.2014.10.017. 124

[96] M. Thorup. Integer priority queues with decrease key in constant time and the single source shortest paths problem. *Journal of Computer and System Sciences*, 69(3):330–353, November 2004. doi:10.1016/j.jcss.2004.04.003. 94

[97] TPIE technical documentation, 2021. URL: http://www.madalgo.au.dk/tpie/doc. 51

[98] K. L. Verdin and J. P. Verdin. A topological system for delineation and codification of the earths river basins. *Journal of Hydrology*, 218(1):1–12, May 1999. doi:10.1016/S0022-1694(99)00011-6. 128

[99] J. S. Vitter. External memory algorithms and data structures. *ACM Computing Surveys*, 33(2):209–271, June 2001. doi:10.1145/384192.384193. 8, 35

[100] M. Wood, J. C. Neal, P. D. Bates, R. Hostache, T. Wagener, L. Giustarini, M. Chini, G. Corato, and P. Matgen. Calibration of channel depth and friction parameters in the LISFLOOD-FP hydraulic model using medium resolution SAR data and identifiability techniques. *Hydrology and Earth System Sciences*, 20(12):4983–4997, December 2016. doi:10.5194/hess-20-4983-2016. 81

[101] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud*. USENIX Association, 2010. 35