I/O EFFICIENT ALGORITHMS AND APPLICATIONS IN GEOGRAPHIC INFORMATION SYSTEMS

by

Andrew Danner

Department of Computer Science
Duke University

Date: _____

Approved:

_____
Lars Arge, Supervisor

_____
Pankaj K. Agarwal, Supervisor

_____
Herbert Edelsbrunner

_____
Helena Mitasova

_____

Dissertation submitted in partial fulfillment of the
requirements for the degree of Doctor of Philosophy
in the Department of Computer Science
in the Graduate School of
Duke University

2006

ABSTRACT

I/O EFFICIENT ALGORITHMS AND APPLICATIONS IN GEOGRAPHIC INFORMATION SYSTEMS

by

Andrew Danner

Department of Computer Science
Duke University

Date: _____

Approved:

_____
Lars Arge, Supervisor

_____
Pankaj K. Agarwal, Supervisor

_____
Herbert Edelsbrunner

_____
Helena Mitasova

_____

An abstract of a dissertation submitted in partial fulfillment of the
requirements for the degree of Doctor of Philosophy
in the Department of Computer Science
in the Graduate School of
Duke University

2006

Abstract

Modern remote sensing methods such a laser altimetry (lidar) and Interferometric Synthetic Aperture Radar (IfSAR) produce georeferenced elevation data at unprecedented rates. Many Geographic Information System (GIS) algorithms designed for terrain modelling applications cannot process these massive data sets. The primary problem is that these data sets are too large to fit in the main internal memory of modern computers and must therefore reside on larger, but considerably slower disks. In these applications, the transfer of data between disk and main memory, or *I/O*, becomes the primary bottleneck. Working in a theoretical model that more accurately represents this two level memory hierarchy, we can develop algorithms that are *I/O-efficient* and reduce the amount of disk I/O needed to solve a problem.

In this thesis we aim to modernize GIS algorithms and develop a number of I/O-efficient algorithms for processing geographic data derived from massive elevation data sets. For each application, we convert a geographic question to an algorithmic question, develop an I/O-efficient algorithm that is theoretically efficient, implement our approach and verify its performance using real-world data. The applications we consider include constructing a gridded digital elevation model (DEM) from an irregularly spaced point cloud, removing topological noise from a DEM, modeling surface water flow over a terrain, extracting river networks and watershed hierarchies from the terrain, and locating polygons containing query points in a planar subdivision. We initially developed solutions to each of these applications individually. However, we also show how to combine individual solutions to form a scalable geo-processing pipeline that seamlessly solves a sequence of sub-problems with little or no manual intervention. We present experimental results that demonstrate orders of magnitude improvement over previously known algorithms.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Flooding from severe weather causes significant damage worldwide each year. In the United States alone, flooding causes billions of dollars in damage annually [32]. Since 2000, over fifty percent of major disaster declarations issued by the US Federal Emergency Management Agency (FEMA), were the result of flooding. FEMA publishes a set of flood hazard maps that indicate the likelihood of flooding in a particular area. These maps are a critical component for determining the need for and the cost of flood insurance. Recent increases in urban development, especially in coastal areas, has created more impervious surfaces that lead to increased runoff and potential increases in flood risk. As a result, many of the flood hazard maps published in the past several decades are outdated and do not accurately reflect current flooding potential. Recently FEMA started a flood map modernization project [63] to develop modern digital flood maps using Geographic Information Systems (GIS) and modern remote sensing techniques. Modern remote sensing methods acquire data at very high resolution quickly and affordably. This makes the acquisition of digital data desirable for many agencies at the federal, state, and local level. The result is a plethora of massive digital geographic data sets. While digital geographic data will result in maps that are more accurate, easier to update, and easier to distribute, processing the massive amount of data poses a number of technical challenges. The development of scalable algorithms for processing massive data sets has not kept pace with the rapid collection of data.

This thesis aims to address the technical challenges of processing massive multi-gigabyte geographic data sets, particularly data sets derived from terrain elevation data acquired using modern remote sensing techniques. The primary challenge is to efficiently manage the transfer of data between large but slow external storage devices and fast but comparatively small internal memory devices where data is actually processed. This thesis focuses on identifying real applications in GIS where current algorithmic methods do not scale to large data sets. We identify several such applications and develop efficient and scalable algorithmic solutions to solve real-world problems. We analyze the complexity of our algorithms in a theoretical framework suitable for large data sets. To demonstrate the practicality and scalability of our solutions, we implement our algorithms and present experimental results based on real-life data sets. Our approach typically processes data sets orders of magnitude larger than previous algorithms.

1.1 Geographic Information Systems

A standard GIS is composed of a number of hardware and software tools that visualize, store, query, and process a variety of spatial data. Examples of popular GIS software providers or products include ESRI, GRASS, IDRISI, and MapInfo. Data objects in a GIS have a spatial component describing their location and an attribute component that describes any non-spatial information about the data objects. A GIS models data in one of two primary ways; as *fields* or as *geometric objects* [70]. In the field representation, each point in space has an attribute value (possibly null) such as elevation, temperature, etc. By overlaying a regular grid over the field and sampling the field value at the center of each grid cell, we obtain the *raster* or *grid* data format. The size of the grid cell or *resolution* controls the level of detail at which we store the field data. In the geometric object representation, features are described by lines, points, polygons, or polytopes. We store the coordinates and connectivity of these features in a *vector* data format.

In addition to simply storing and visualizing data, a GIS provides a set of algorithms for processing, querying, and manipulating data. For example, given a data set describing elevation of

a terrain, GIS algorithms could model water flow over the terrain and extract river networks or compute flooding extent based on river inundation height. These geo-processing algorithms add tremendous value to the data and allow users to answer complex questions about their surroundings. While many raster and vector data sets can be derived from other data sets via geo-processing, there must be an initial source of data to supply to the GIS.

1.2 Data Collection

The initial or *base* data for a GIS can be collected using direct field measurements, digitization of paper maps, or remote sensing methods. Originally digital data for a GIS was manually converted from paper maps or manually entered into a digital format from direct field measurements. These methods are particularly slow and labor intensive. Modern remote sensing methods acquire digital data directly and collect massive amounts of geographical data rapidly without extensive manual intervention.

Remote sensing methods acquire data about an object without direct physical contact, typically using sensors aboard aircraft or satellites. Remote sensing devices methods can be either *passive* or *active*. A passive remote sensing method records ambient electromagnetic energy (e.g., from the sun) reflected from the surface of the Earth or energy emitted (e.g., infrared thermal energy) from the surface. The recent orthophotography and satellite imagery on Google maps is one popular example of passive remote sensing. Alternatively, *active* remote sensing methods generate their own source of energy that interacts with the terrain, backscatters, and is recorded by a sensor. Three common active remote sensing methods are radar, lidar, and sonar. Each of these systems sends a pulse of radiation (or sound in the case of sonar) towards a target and detects the energy reflected from the target. By knowing the location of the sensor and the travel time of the pulse, one can measure the distance between the sensor and the target. The location of the sensor is typically known using global positioning systems (GPS) and/or an inertial navigational system (INS). As the sensor moves forward, remotely sensed data is collected in swaths perpendicular to the direction of travel.

In this thesis, we are particularly interested in remotely sensed data that describes the elevation of Earth's surface. The two most common means of collecting terrestrial elevation data are interferometric synthetic aperture radar (IfSAR) and light detection and ranging (lidar). Sonar is typically used for bathymetric applications (measuring the depth of a waterbody). IfSAR uses a pair of antennas that each record both the amplitude and phase of reflected microwave (3-25 cm wavelength) signals. The phase difference between the two recorded signals is used to measure the elevation of the underlying terrain. The antennas can be mounted on an airplane, a helicopter, or—in the case of NASA's Shuttle Radar Topography Mission (SRTM)—a space shuttle. Airborne IfSAR has a spatial resolution of up to approximately 3 meters when flown at 6100m above ground level and a swath width of 8km [55]. IfSAR systems on spacecraft have a resolution of 25-100 meters.

Lidar uses laser pulses and a single sensor to measure the elevation by recording the round trip time of the laser pulse and knowing the initial position of the beam. Modern systems emit pulses at rates between 10,000 and 50,000 pulses per second. Lidar uses a smaller wavelength (0.90 μm) in the near infrared spectrum. Aircraft with lidar systems fly close to the ground at altitudes of 1200–2400 meters above ground level. Because lidar uses a smaller wavelength and data is collected closer to the ground, lidar can have a spatial resolution better than IfSAR and approaching 1 meter. However, the typical width of a lidar swath is 1800m at a flight altitude of 2400m [55], so more flight passes are needed to cover the same area as IfSAR sensors. For further technical details on IfSAR and lidar refer to [55, 58] and the references therein.

Lidar remote sensing represents the state of the art in modern mapping of elevation. Massive amounts of data can be acquired quickly over a large area, and at an affordable costs. The speed of data acquisition make the collection of multiple time-series data sets at yearly or monthly intervals

possible. The state of North Carolina is mapping the entire state using lidar technology and other states are planning to do the same. However, raw lidar data is collected as a set of scattered x, y, z points and this raw point format is not suitable for terrain analysis. The data must first be converted to a raster or vector terrain model.

1.3 Terrain Modelling

One of the primary benefits of remotely sensed elevation data sets is the ability to build a digital model the Earth's surface and analyze its properties. However, only a fraction of all points represent heights of the surface of the Earth. Pulses can bounce off of treetop canopies, buildings, and even features as small as mailboxes. In some applications, users are interested forest canopies or urban landscapes, but for people interested in studying the underlying terrain, these features are considered noise that obscure the underlying surface. Typically, lidar data providers process the raw data to remove vegetation and buildings and deliver a set of *bare Earth* points to users interested in terrain applications. In areas of dense vegetation, 90% or more of the original point set may be discarded as not part of the bare Earth model. Resulting bare Earth point sets therefore have a heterogeneous point density.

A set of point measurements is not a very good terrain model. Given an arbitrary point in an unordered list of height measurements, it would be difficult to find a point that is geographically nearby. Therefore, the raw elevation point samples are usually converted to more usable digital elevation model, or *DEM*. The two most common terrain or surface models are the grid or triangulated irregular network (TIN). In the grid model, the terrain is modeled as a two-dimensional array of cells, each with an elevation. Each cell is adjacent to at most eight neighboring cells and finding the elevation of nearby regions is as simple as indexing a particular cell in the array.

The TIN model connects all the point samples by a planar triangulation. Any given location in space maps to a single triangle in the TIN. The elevation of that location can be found by approximating the surface as a plane passing through the three-dimensional vertices of the triangle containing the location. Both TIN and grid DEMs are used frequently in GIS applications.

Algorithms for terrain modeling convert a set of remotely sensed points to a grid or TIN DEM. A number of such algorithms have been proposed in the literature and we will later review some of the more common approaches. Most algorithms were designed to work only for hundreds or thousands of points acquired by traditional data collection methods and do not scale to modern remote sensing methods that produce millions or billions of points. For GIS users to benefit for this new hi-resolution data, we must design efficient methods for constructing terrain models from massive data sets. The problem does not end with constructing a terrain model however. The resulting models themselves could be very large and constructing a DEM is only the first step in a number of terrain analysis applications.

1.4 Terrain Analysis

Given a DEM, GIS algorithms can process the terrain to answer complex questions or derive other GIS layers describing various properties of the terrain. We have previously mentioned the ability to estimate flood hazard risk using elevation data as motivating FEMA to collect digital data using remote sensing. Other applications of bare Earth DEMs include modeling erosion, assessing landslide risk, monitoring topographic change over time, and many more. Lidar surface models that are not the bare Earth elevation and include tree canopy height or buildings can be used for ecological studies of forest growth or deforestation, or visibility analysis. We further highlight a few applications in hydrography that are particularly relevant in the context of this thesis.

We first consider the problem of modeling surface water flow over a terrain. Suppose we are given a bare Earth model of a terrain. When water falls on a terrain, it flows downhill until it

reaches a local minimum[1]. As more water flows downhill, small creeks are formed that flow into larger streams and rivers which typically flow into a large global minimum such as a lake, sea, or ocean. If we assume that for each point on the surface of the terrain, water flows in the direction of steepest descent, the collection of paths of steepest descent form river networks. We say that at region at a higher elevation *drains* through a region at a higher elevation if there is a path of steepest descent from the higher region to the lower region. We could compute for any small area in a terrain the total amount of area upslope that drains through the area. This is typically called the *flow accumulation* or *upslope contributing area* of the region. Regions that have a high upslope contributing area are good indicators of rivers and given a digital elevation model, we can use these concepts to automatically extract river networks from the terrain.

Modeling flow over a digital elevation model is complicated by the fact that DEMs are constructed from measured data that is subject to error. Even with highly accurate remote sensing methods, a typical DEM has some level of noise. In hydrologic applications this noise can create a number of spurious local minima (or maxima) that are not present in the actual terrain. The primary source of noise in low-resolution DEMs was usually due to sampling errors and the inability to resolve small features such as narrow canyons. Modern remote sensing methods resolve much smaller features than previous approaches, but this can leads to a number of new problems. As mentioned earlier, lidar often detects buildings, trees, and bridges. When constructing a bare Earth model, lidar data providers attempt to remove these features, which are hydrologically considered noise, but the procedure is not perfect. Imprecise removal of vegetation can produce a rough bare Earth surface with numerous small spurious minima and maxima. A bigger problem is that bridge removal techniques are not completely accurate and a number of bridges exist in many of the bare Earth models. Bridges are particularly troublesome, because they almost always, by definition, intersect the path of natural waterways. Because both the grid and TIN DEMs model the surface as a single valued height function over a two-dimensional plane, the presence of bridges creates local minima upstream of the bridge locations.

If we use the simple model described above that always routes water down the steepest downslope path in the DEM, flow paths will get stuck in the local minima or *sinks* created by the either noise, or real terrain features such as bridges. These sinks are undesirable in most hydrological applications and thus one terrain analysis application is to remove sinks from a DEM. Removing sinks that prevent flow paths from accurately representing real river networks is referred to as *hydrologically conditioning* the DEM. Creating hydrologically conditioned DEMs in hi-resolution data sets in which bridges are present is a new challenge in terrain analysis.

To accurately construct river networks from a set of remote sensed elevation points, we first construct a DEM from remotely sensed data and then hydrologically condition the DEM. After modeling water flow and extracting rivers networks from the terrain, we can further extend our terrain analysis and extract watersheds from the river network. In this application we wish to partition the terrain into a set of non-overlapping regions called *hydrological units* or *watersheds* such that all water in a single in hydrological unit flows through a common outlet. For many hydrology applications, modeling is done using hydrological units rather than using the terrain or river network directly. It is often desirable to create a hierarchy of hydrological units at increasingly finer scales. These levels of detail allow users to select hydrological units at a national, regional, or local scale, depending on their interests.

1.5 Massive Problems with Massive Data

The few applications of terrain analysis highlighted above form a set of basic derived GIS layers that are used in a number of more complex applications. Thus it is important to have efficient

[1]For simplicity, assume the terrain is impervious and water is neither absorbed nor emitted by the surface of the terrain.

algorithms for performing these types of terrain analysis. Furthermore, in this particular set of examples, each application depends on some other application. Watershed extraction depends on river network extraction, which depends on sink removal, which depends on modeling a terrain from a set of raw elevation sample points. An efficient and scalable solution for extracting watersheds on massive terrain data sets is useless in practice if we do not have efficient and scalable solutions for all previous steps in this geo-processing pipeline.

From a computational perspective the problem with massive, multi-gigabyte or terrabyte data sets is that they are too large to fit in the small but fast main memory of a modern computer and must therefore reside on larger, but much slower disks. Computers can only process data that resides in main memory and must move data between memory and disk when processing massive data sets. Because hard disk drives are mechanical devices with movable parts, the access time of data on disk is on the order of milliseconds, six orders of magnitude slower than the nanosecond access time of solid state devices including main memory. The transfer of data between disk and main memory or *I/O* often becomes a bottleneck when processing massive data sets.

The seek time to find a single data element on an external disk is a significant fraction of data access time. Therefore, on modern computer systems, data is transfered between disk and main memory in a sequential block of contiguous items. Because sequential access of data is comparatively faster than the seek time, the amortized I/O cost per item using block I/O is significantly lower than issuing a separate seek for each individual item. If a program can use all items in a block in main memory before replacing it with another block from disk, this *locality of data reference* results in dramatically better performance than programs that do not have such data reference locality. To develop scalable algorithms that can process massive data sets that reside on disk, we must model the behavior of the I/O-bottleneck and develop algorithms that are efficient in this model.

1.6 The I/O Model of Computation

Most algorithms, including those in GIS, are traditionally designed in the random access machine (RAM) model of computation. This is a rather simple model of computation that assumes a single processor and an infinite amount of internal memory. Each item accessed in memory has a constant access cost. The measure of performance and complexity in this model is the number of comparisons or simple arithmetic operations needed to solve a problem. While this model is sufficient for small data sets that fit in main memory, it does not accurately model the multiple levels of memory hierarchy in modern computers. Particular, the assumption of unit data access cost is false in systems where data reside in both a fast internal memory and on slow external disks.

To better model a two-level memory hierarchy, Aggarwal and Vitter [7] proposed the I/O or external-memory model of computation. This model consists of a processor, an finite internal memory of size M items, and an infinite sized external disk. Data is transferred between the internal memory and the external disk in blocks of B items. Such a transfer is considered one *I/O*. The complexity of an algorithm in this model is the total number of I/Os needed to solve the problem in terms of the four parameters N, M, B, and T, where N is the number of items in the problem instance and the output size is T. In addition to the I/O complexity, one often also considers the space complexity in terms of number of blocks used and the CPU work complexity. While this two-level I/O model is still an approximation to a real machine, it captures the main features of block I/O and is the primary theoretical model for developing and analyzing algorithms that scale to large data sets and are thus *I/O-efficient*.

The two-level memory hierarchy in the I/O model is only an approximation of modern computers which have multiple levels of memory hierarchy including disks, main memory, L2 cache, L1 cache, and registers. Frigo et al. [47] introduced the cache-oblivious model of computation to analyze algorithms in a multi-level hierarchy. In this model, algorithms are developed in the RAM model of computation, but are analyzed in the I/O-model. Because they are developed in the RAM

model, they can not explicitly use the I/O-model parameters M and B in the algorithm design. However, if the algorithm is efficient when analyzed in the I/O-model, then this cache-oblivious algorithm is efficient between any two levels of the memory hierarchy, regardless of the parameters M and B on that level. It would seem that we should aim to develop cache-oblivious algorithms, but it is typically difficult to develop I/O-efficient algorithms without knowledge of M and B. Furthermore, the primary bottleneck when processing massive data is the bottleneck between disk and main memory. If we optimize our algorithm for just these two levels, we can get tremendous improvements in scalability and run-time. While cache-oblivious algorithms are of theoretical and practical interest in certain applications, we find that the I/O-model is suitable for our purpose and will therefore be the primary focus of this thesis.

Using the I/O-model of computation many I/O-efficient algorithms and data structures have been developed. Trivially, the bound for scanning N elements is $\text{scan}(N) = O(N/B)$ I/Os. Aggarwal and Vitter [7] presented algorithms for sorting, permuting, fast Fourier transform, and matrix transposition. The sorting bound is $\text{sort}(N) = \Theta(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$. Numerous basic data structures have been developed in the I/O model including B-trees [28, 35, 56], priority queues [10, 31], kd-B trees [75], R-trees [53, 18], persistent B-trees [29, 88], union-find [6], and many others. Agarwal et al. presented a framework [4] for efficiently constructing a number of spatial data structures using a bulk-loading approach. We use many of these data structures in this thesis and will describe them in more detail as they are needed.

In addition to the basic data structures and algorithms, a number of other I/O-efficient algorithms have been developed. Particularly relevant to this thesis are algorithms for planar point location [51, 25, 27, 86, 17], circuit evaluation and processing of topologically sorted directed acyclic graphs (DAGs) [34, 10], and flow modeling on grid terrains [14]. Many I/O-efficient algorithms have applications in GIS including algorithms for spatial joins [21], contour line extraction [3], algorithms on gridded terrains [23], algorithms for processing line segments [27] and recent work on constrained Delaunay triangulations [5]. A common paradigm for developing many I/O-efficient algorithms for spatial data the use of plane sweep techniques. In several cases, including flow modeling on gridded terrains [14], pre-sorting the data according to the sweep order offers tremendous speedup over naive approaches that work well in internal memory. For a summary of many other I/O-efficient algorithms and data structures, refer to extensive surveys by Vitter [91] and Arge [9].

While many problems have I/O-efficient solutions, efficient algorithms on graphs have been notoriously difficult. On general graphs, even simple problems such as BFS and DFS do not have simple solutions [34, 33, 61]. On planar graphs, many algorithms have good theoretical bounds [13, 20, 84, 24], but are impractical to implement. On trees, many algorithms including DFS, BFS, and Euler tours can be solved in $O(\text{sort}(N))$ I/Os using list-ranking.

Initial work in the I/O-model was mostly theoretical, but there have been several efforts to develop experimental libraries for implementing I/O-efficient algorithms including LEDA-SM [37], TPIE [22, 11], and STXXL [39]. While LEDA-SM is no longer maintained, STXXL and TPIE are both currently used and updated. TPIE [89, 22, 12], is a templated, portable I/O environment written in C++ that abstracts away the details of implementing I/O-efficient algorithms and provides users with a number of built-in algorithms and data structures for managing large data sets. Primitives for data structures such as streams, stacks, queues, priority-queues, and B-trees are part of the TPIE code-base. A number of sorting methods are also provided in TPIE. On top of this core library, a number of applications have been developed. The TerraFlow [14] project developed by Arge et al. uses TPIE to efficiently model surface water flow over massive grid DEMs. Experimental results show that TerraFlow can scale to data sets much larger than previous GIS algorithms. Motivated by the success of TerraFlow, we aim to develop additional I/O-efficient algorithms for processing massive GIS data sets and demonstrate the practicality of our approach with experimental results on real-world data.

1.7 Outline of Thesis

In this thesis, we describe several problems in Geographic Information Systems that would benefit from modern hi-resolution data sets derived from modern remote sensing methods. For each problem, previous algorithm solutions did not scale well to massive data sets. We develop scalable solutions to these problems and show that we can efficiently process data sets orders of magnitude larger than the largest data sets we could process with previous algorithms. For each application, our emphasis is almost equally distributed amongst converting a real-world application to a computational problem, developing an I/O-efficient solution to the problem, implementing this solution, and demonstrating the scalability of the implementation using real-world data sets. In this thesis, we focus on hydrographic applications on grid DEMs.

In Chapter 2 we consider the problem of constructing a grid digital elevation model (DEM) given a set $\mathcal{S}$ of N points sampled from a surface. Our approach [2] is based on a scalable segmentation of the point set using quad-trees and an approximation method for evaluating the value of the surface at an arbitrary point. The design is flexible and can incorporate a number of different interpolation methods. Using over 390 million lidar points sampled from the Neuse river basin in North Carolina, we show that we can process extremely large input sets I/O-efficiently, while prior approaches failed to process more than 25 million points.

Given a grid DEM such as the one we construct in Chapter 2, we discuss previous methods of detecting and removing sinks from grid digital elevation models (DEMs) in Chapter 3. As noted earlier, removal of buildings and vegetation from lidar point sets is often incomplete and artifacts such as bridges often remain in DEMs constructed from bare Earth lidar points. Successfully detecting and removing bridges and other sources of sinks from DEMs will lead to better hydrologically conditioned DEMs. This chapter discusses new methods [6] for sink removal based on topological persistence [45, 44], and reviews the TerraFlow [14] algorithm of Arge et al. for modeling surface water flow on large grid terrains. We also present new ways of handling some degenerate cases in flow modeling such as how to find water flow paths across extended flat surfaces where no locally downslope path exists. Under the likely assumption that a constant number of grid rows fit in memory, our new method is more practical than other algorithms with the same theoretical I/O-bound.

Given a river network extracted from a hydrologically conditioned DEM, we can extract hydrological units. Many hydrological modelling tools in GIS work directly with these hydrological units instead of the original elevation data and it is important to have scalable techniques for computing such watershed hierarchies. Chapter 4 describes our I/O-efficient method [15] for decomposing a terrain into a hierarchy of *hydrological units* or *watersheds*. We implemented our approach, based on the Pfafstetter watershed labeling definition [90] recently presented by Verdin and Verdin. Experimental results demonstrate the scalability of our approach on a number of large watersheds. Neither of the two GIS packages we used (GRASS and ArcGIS) had tools for automatically extracting a hierarchy of watersheds. Most previous methods required users to manually select a number of desired watershed outlets, or could only produce a single layer of the hierarchy automatically.

Given a partition of a terrain into hydrological units and a query point, we would like to quickly identify the hydrological unit containing the query point. This is a specific example of the planar point location problem which is defined as: given a planar subdivision containing N vertices and a query point q in the plane, report the face of the subdivision that contains q. In Chapter 5 we describe an I/O-efficient solution [16] for planar point location based on persistent B-trees [29, 88] and vertical ray-shooting. Our approach is practical enough to implement and has better worst-case performance than a simple heuristic-based bucket approach [86]. We present a number of experimental results on both real and synthetic data that show the practicality of our approach.

Finally, in Chapter 6, we show how to combine the individual results of previous chapters to form an extended geo-processing pipeline that can automatically take remotely sensed elevation data points and extract the watershed hierarchy of the terrain with little or no manual intervention.

All steps of the pipeline are scalable to large data sets and we present experimental results on the Neuse basin to support our claims. The completion of this I/O-efficient pipeline eliminates the need for users to try to divide their data set into smaller manageable pieces when processing massive terrain data sets derived from hi-resolution point sets. This allows users to focus more on modelling issues and less on computational issues.

Chapter 2

Grid DEM Construction

2.1 Introduction

One of the basic tasks of a geographic information system (GIS) is to store a representation of various physical properties of a terrain such as elevation, temperature, precipitation, or water depth, each of which can be viewed as a real-valued bivariate function. Because of simplicity and efficacy, the grid or raster representaion is one of the widely used representations of such data. However, many modern mapping technologies do no acquire data on a uniform grid. Hence the raw data is a set $\mathcal{S}$ of N (arbitrary) points in $\mathbb{R}^3$, sampled from a function $H : \mathbb{R}^2 \to \mathbb{R}$. An important task in GIS is thus to interpolate $\mathcal{S}$ on a uniform grid of a prescribed resolution.

In this chapter, we present a scalable algorithm [2] for this interpolation problem. Although our technique is general to any set of points sampled from a surface, we focus on constructing a grid digital elevation model (DEM) from a very large set $\mathcal{S}$ of N points in $\mathbb{R}^3$ acquired by modern mapping techniques such as lidar [1]. Because these data sets are much larger than main memory we develop an I/O-efficient algorithm for constructing a grid DEM of unprecedented size from these massive data sets.

2.1.1 Related Work

A variety of methods for interpolating a surface from a set of points have been proposed, including inverse distance weighting (IDW), kriging, spline interpolation and minimum curvature surfaces. Refer to [65] and the references therein for a survey of the different methods. However, the computational complexity of these methods often make it infeasible to use them directly on even moderately large points sets. Therefore, many practical algorithms use a segmentation scheme that decomposes the plane (or rather the area of the plane containing the input points) into a set of *non-overlapping areas* (or *segments*), each containing a small number of input points. One then interpolates the points in each segment independently. Numerous segmentation schemes have been proposed, including simple regular decompositions and decompositions based on Voronoi diagrams [78] or quad trees [67, 62]. A few schemes using *overlapping* segments have also been proposed [93, 74].

Many $\Theta(N)$ time algorithms designed in the RAM model of computation that do not explicitly consider I/O use $\Theta(N)$ I/Os when used in the I/O-model. However, the "linear" bound, the number of I/Os needed to read N elements, is only $\Theta(\text{scan}(N)) = \Theta(\frac{N}{B})$ in the I/O model. Recalling that I/O-efficient sorting is roughly three scans, tremendous speedups can often be obtained by developing algorithms that use $O(\text{scan}(N))$ or $O(\text{sort}(N))$ I/Os rather than $\Omega(N)$ I/Os. Agarwal et. al [4] presented a general top-down layered framework for constructing a certain class of spatial data structures–including quad trees–I/O-efficiently. Hjaltason and Samet [54] also presented an I/O-efficient quad-tree construction algorithm. This optimal $O(\text{sort}(N))$ I/O algorithm is based on assigning a Morton block index to each point in $\mathcal{S}$, encoding its location along a Morton-order (Z-order) space-filling curve, sorting the points by this index, and then constructing the structure in a bottom-up manner.

[1] In this chapter, we consider lidar data sets that represent the actual terrain and have been preprocessed by the data providers to remove spikes and errors due to noise.

2.1.2 Our Approach

In this chapter we describe an I/O-efficient algorithm for constructing a grid DEM from lidar points using a quad-tree segmentation. Most of the segmentation algorithms for this problem can be considered as consisting of three separate phases; the *segmentation* phase, where the decomposition is computed based on $\mathcal{S}$; the *neighbor finding* phase, where for each segment in the decomposition the points in the segment and the relevant neighboring segments are computed; and the *interpolation* phase, where a surface is interpolated in each segment and the interpolated values of the grid cells in the segment are computed. In this chapter, we are more interested in the segmentation and neighbor finding phases than the particular interpolation method used in the interpolation phase. We will focus on the quad tree based segmentation scheme because of its relative simplicity and because it has been used with several interpolation methods such as thin plate splines [67] and B-splines [62]. While we focus on the quad-tree, techniques apply to other segmentation schemes as well.

Our algorithm implements all three phases I/O-efficiently, while allowing the use of any given interpolation method in the interpolation phase. Given a set $\mathcal{S}$ of N points, a desired output grid specified by a bounding box and a cell resolution, as well as a threshold parameter $k_{\max}$, the algorithm uses $O(\frac{N}{B}\frac{h}{\log \frac{M}{B}} + \text{sort}(T))$ I/Os, where h is the height of a quad tree on $\mathcal{S}$ with at most $k_{\max}$ points in each leaf, and T is the number of cells in the desired grid DEM. Note that this is $O(\text{sort}(N) + \text{sort}(T))$ I/Os if $h = O(\log \frac{N}{B})$, that is, if the points in $\mathcal{S}$ are distributed such that the quad tree is roughly balanced.

The three phases of our algorithm are described in Section 2.2, Section 2.3 and Section 2.4. In Section 2.2 we describe how to construct a quad tree on $\mathcal{S}$ with at most $k_{\max}$ points in each leaf using $O(\frac{N}{B}\frac{h}{\log \frac{M}{B}})$ I/Os. The algorithm is similar to the bulk-loading framework of Agarwal et. al [4]. Although not as efficient as the algorithm by Hjaltason and Samet [54] in the worst case, our algorithm is simpler and potentially more practical; for example, it does not require computation of Morton block indices or sorting of the input points. Also in most practical cases where $\mathcal{S}$ is relatively nicely distributed, for example when working with lidar data, the two algorithms both use $O(\text{sort}(N))$ I/Os. In Section 2.3 we describe how to find the points in all neighbor leaves of each quad-tree leaf using $O(\frac{N}{B}\frac{h}{\log \frac{M}{B}})$ I/Os. The algorithm is simple and very similar to our quad-tree construction algorithm; it takes advantage of how the quad tree is naturally stored on disk during the segmentation phase. Note that while Hjaltason and Samet [54] do not describe a neighbor finding algorithm based on their Morton block approach, it seems possible to use their approach and an I/O-efficient priority queue [10] to obtain an $O(\text{sort}(N))$ I/O algorithm for the problem. However, this algorithm would be quite complex and therefore probably not of practical interest. Finally, in Section 2.4 we describe how to apply an interpolation scheme to the points collected for each quad-tree leaf, evaluate the computed function at the relevant grid cells within the segment corresponding to each leaf, and construct the final grid using $O(\text{scan}(N)) + O(\text{sort}(T))$ I/Os. As mentioned earlier, we can use any given interpolation method within each segment.

To investigate the practical efficiency of our algorithm we implemented it and experimentally compared it to other interpolation algorithms using lidar data. To summarize the results of our experiments, we show that, unlike previous algorithms, our algorithm scales to data sets much larger than the main memory. For example, using a machine with 1GB of RAM, we were able to construct a grid DEM containing 397 million real grid data cells (occupying 1.2GB of disk space when including 800 million nodata nodes) from a lidar data set of over 390 million points (occupying 20GB) in 53 hours. This data set is an order of magnitude larger than what could be handled by two popular GIS products–ArcGIS and GRASS. In addition to supporting large input point sets, we were also able to construct very large high resolution grids; in one experiment we constructed a one meter resolution grid DEM containing more than 53 billion cells—storing just a single bit for each grid cell in this DEM requires 6GB.

In Section 2.5 we describe the details of the implementation of our theoretically I/O-efficient algorithm that uses a regularized spline with tension interpolation method [66]. We also describe

the details of a prior algorithm implemented in GRASS using the same interpolation method; this algorithm is similar to ours but it is not I/O-efficient. In Section 2.6 we describe the results of the experimental comparison of our algorithm to other implementations. As part of this study, we present a detailed comparison of the quality of the grid DEMs produced by our algorithm and the similar algorithm in GRASS that show the results are in good agreement.

2.2 Segmentation Phase: Quad-Tree Construction

Given a set $\mathcal{S}$ of N points contained in a bounding box $[x_1, x_2] \times [y_1, y_2]$ in the plane, and a threshold $k_{\max}$, we wish to construct a quad tree $\mathcal{T}$ [38] on $\mathcal{S}$ such that each quad-tree leaf contains at most $k_{\max}$ points. Note that the leaves of $\mathcal{T}$ partition the bounding box $[x_1, x_2] \times [y_1, y_2]$ into a set of disjoint areas, which we call *segments*.

2.2.1 Incremental Construction

$\mathcal{T}$ can be constructed incrementally simply by inserting the points of $\mathcal{S}$ one at a time into an initially empty tree. For each point p, we traverse a root-leaf path in $\mathcal{T}$ to find the leaf v containing p. If v contains less than $k_{\max}$ points, we simply insert p in v. Otherwise, we split v into four new leaves, each representing equally sized quadrants of v, and re-distribute p and the points in v to the new leaves. If all $k_{\max}+1$ points are distributed to one leaf, we recursively apply the splitting procedure. If h is the height of $\mathcal{T}$, this algorithm uses $O(Nh)$ time. If the input points in $\mathcal{S}$ are relatively evenly distributed we have $h = O(\log N)$, and the algorithm uses $O(N \log N)$ time.

If $\mathcal{S}$ is so large that $\mathcal{T}$ must reside on disk, traversing a path of length h may require as many as h I/Os, leading to an I/O cost of $O(Nh)$ in the I/O-model. By storing (or *blocking*) the nodes of $\mathcal{T}$ on disk intelligently, we may be able to access a subtree of height $\log B$ (size B) in a single I/O and thus reduce the cost to $O(N\frac{h}{\log B})$ I/Os. Caching the top-most levels of the tree in internal memory may also reduce the number of I/Os needed. However, since not all the levels fit in internal memory, it is hard to avoid spending an I/O to access a leaf during each insertion, or $\Omega(N)$ I/Os in total. Since $\text{sort}(N) \ll N$ in almost all cases, the incremental approach is very inefficient when the input points do not fit in internal memory.

2.2.2 Level-by-level Construction

A simple I/O-efficient alternative to the incremental construction algorithm is to construct $\mathcal{T}$ level-by-level: We first construct the first level of $\mathcal{T}$, the root v, by scanning through $\mathcal{S}$ and, if $N > k_{\max}$, distributing each point p to one of four leaf lists on disk corresponding to the child of v containing p. Once we have scanned $\mathcal{S}$ and constructed one level, we construct the next level by loading each leaf list in turn and constructing leaf lists for the next level of $\mathcal{T}$. While processing one list we keep a buffer of size B in memory for each of the four new leaf lists (children of the constructed node) and write buffers to the leaf lists on disk as they run full. Since we in total scan $\mathcal{S}$ on each level of $\mathcal{T}$, the algorithm uses $O(Nh)$ time, the same as the incremental algorithm, but only $O(Nh/B)$ I/Os. However, even in the case of $h = \log_4 N$, this approach is still a factor of $\log_{\frac{M}{B}} \frac{N}{B} / \log_4 N$ from the optimal $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ I/O bound.

2.2.3 Hybrid Construction

Using the framework of Agarwal et. al [4], we design a hybrid algorithm that combines the incremental and level-by-level approaches. Instead of constructing a single level at a time, we can construct a *layer* containing the top $\log_4 \frac{M}{B}$ levels in a single pass over the data. Because $4^{\log_4 \frac{M}{B}} = M/B < M$, we construct these levels entirely in internal memory using the incremental approach: We scan

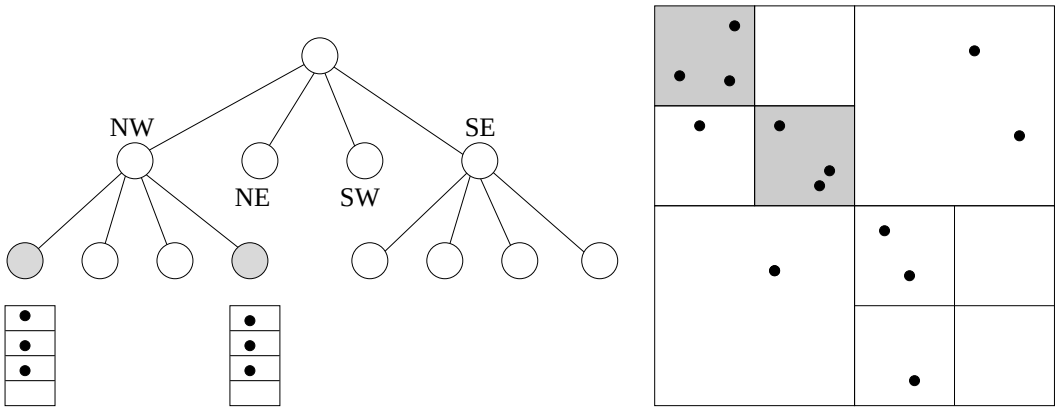


Figure 2.1: Construction of a quad-tree layer of depth three with $k_{\max} = 2$. Once a leaf at depth three is created, no further splitting is done; instead additional points in the leaf are stored in leaf lists shown below shaded nodes. After processing all points the shaded leaves with more than two points are processed recursively.

through $\mathcal{S}$, inserting points one at a time while splitting leaves and constructing new nodes, except if the path from the root of the layer to a leaf of the layer is of height $\log_4 \frac{M}{B}$. In this case, we write all points contained in such a leaf v to a list L_v on disk. After all points have been processed and the layer constructed, we write the layer to disk sequentially and recursively construct layers for each leaf list L_i. Refer to Figure 2.1.

Since a layer has at most M/B nodes, we can keep an internal memory buffer of size B for each leaf list and only write points to disk when a buffer runs full (for leaves that contain less than B points in total, we write the points in all such leaves to a single list after constructing the layer). In this way we can construct a layer on N points in $O(N/B) = \text{scan}(N)$ I/Os. Since a tree of height h has $h/\log_4 \frac{M}{B}$ layers, the total construction cost is $O(\frac{N}{B}\frac{h}{\log \frac{M}{B}})$ I/Os. This is $\text{sort}(N) = O(\frac{N}{B}\log_{\frac{M}{B}} \frac{N}{B})$ I/Os when $h = O(\log N)$.

2.3 Neighbor Finding Phase

Let $\mathcal{T}$ be a quad tree on $\mathcal{S}$. We say that two leaves are neighbors if their associated segments share part of an edge or a corner. Refer to Figure 2.2 for an example. If $\mathcal{L}$ is the set of segments associated with the leaves of $\mathcal{T}$, we want to find for each $q \in \mathcal{L}$ the set $\mathcal{S}_q$ of points contained in q and the neighbor leaves of q. As for the construction algorithm, we first describe an incremental algorithm and then improve its efficiency using a layered approach.

2.3.1 Incremental Approach

For each segment $q \in \mathcal{L}$, we can find the points in the neighbors of q using a simple recursive procedure: Starting at the root v of $\mathcal{T}$, we compare q to the segments associated with the four children of v. If the bounding box of a child u shares a point or part of an edge with q, then q is a neighbor of at least one leaf in the tree rooted in u; we therefore recursively visit each child with an associated segment that either neighbors or contains q. When we reach a leaf we insert all points in the leaf in $\mathcal{S}_q$.

To analyze the algorithm, we first bound the total number of neighbor segments found over all segments $q \in \mathcal{L}$. Consider the number of neighbor segments that are at least the same size as a given segment q; at most one segment can share each of q's four edges, and at most four

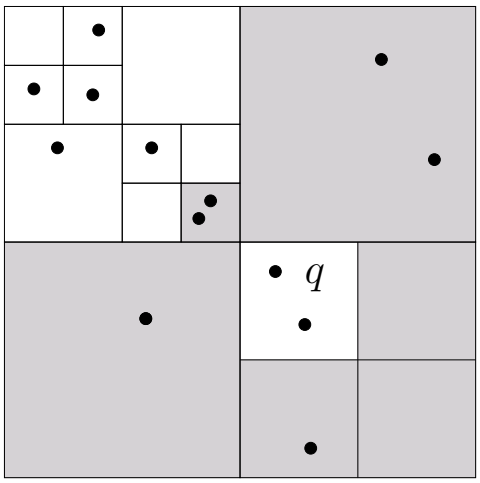


Figure 2.2: The segment q associated with a leaf of a quad tree and its six shaded neighboring segments.

more segments can share the four corner points of q. Thus, there are at most eight such neighbor segments. Because the neighbor relation is symmetric, the total number of neighbor segments over all segments is at most twice the total number of neighbor segments which are at least the same size. Thus the total number of neighbor segments over all segments is at most 16 times the number of leaves of $\mathcal{T}$. Because the total number of leaves is at most $4N/k_{\max}$, and since the above algorithm traverses a path of height h for each neighbor, it visits $O(Nh)$ nodes in total. Furthermore, as each leaf contains at most $k_{\max}$ points, the algorithm reports $O(Nk_{\max})$ points in total. Thus the total running time of the algorithm is $O((h + k_{\max})N) = O(hN))$. This is also the worst case I/O cost.

2.3.2 Layered approach

To find the points in the neighboring segments of each segment in $\mathcal{L}$ using a layered approach similar to the one used to construct $\mathcal{T}$, we first load the top $\log_4 \frac{M}{B}$ levels of $\mathcal{T}$ into memory. We then associate with each leaf u in the layer, a buffer B_u of size B in internal memory and a list L_u in external memory. For each segment $q \in \mathcal{L}$, we use the incremental algorithm described above to find the leaves of the layer with an associated segment that completely contains q or share part of a boundary with q. Suppose u is such a layer leaf. If u is also a leaf of the entire tree $\mathcal{T}$, we add the pair $(q, \mathcal{S}_u)$ to a global list Λ, where $\mathcal{S}_u$ is the set of points stored at u. Otherwise, we add q to the buffer B_u associated with u, which is written to L_u on disk when B_u runs full. After processing all segments in $\mathcal{L}$, we recursively process the layers rooted at each leaf node u and its corresponding list L_u. Finally, after processing all layers, we sort the global list of neighbor points Λ by the first element q in the pairs $(q, \mathcal{S}_u)$ stored in Λ. After this, the set $\mathcal{S}_q$ of points in the neighboring segments of q are in consecutive pairs of Λ, so we can construct all $\mathcal{S}_q$ sets in a simple scan of Λ.

Since we access nodes in $\mathcal{T}$ during the above algorithm in the same order they were produced in the construction of $\mathcal{T}$, we can process each layer of $\log_4 \frac{M}{B}$ levels of $\mathcal{T}$ in scan(N) I/Os. Furthermore, since $\sum_q |\mathcal{S}_q| = O(N)$, the total number of I/Os used to sort and scan Λ is $O(\text{sort}(N))$. Thus the algorithm uses $O(\frac{N}{B} \frac{h}{\log \frac{M}{B}})$ I/Os in total, which is $O(\text{sort}(N))$ when $h = O(\log N)$.

2.4 Interpolation Phase

Given the set $\mathcal{S}_q$ of points in each segment q (quad tree leaf area) and the neighboring segments of q, we can perform the interpolation phase for each segment q in turn simply by using any interpolation method we like on the points in $\mathcal{S}_q$, and evaluating the computed function to interpolate each of the grid cells in q. Typically, $|\mathcal{S}_q|$ is much less than M. If this is not the case, we can choose a maximum number of points $n_{\max} < M$ and interpolate only on these points. We can do this efficiently by scanning the points in $\mathcal{S}_q$ and keep the $n_{\max}$ points closest to the center of q in memory. This is easily done with an internal memory priority queue. Keeping both $n_{\max}$ and $k_{\max}$ small reduces the overall cost of the interpolation phase. Since at most $k_{\max}$ are in each quad-tree leaf and each leaf has eight neighboring leaves on average, $|\mathcal{S}_q| < 8k_{\max}$ for most segments q and we do not need to worry about keeping only $n_{\max}$ points.

Since $\sum_q |\mathcal{S}_q| = O(N)$, we can read each $\mathcal{S}_q$ into main memory and perform the interpolation in $O(\text{scan}(N))$ I/Os in total. However, we cannot simply write the interpolated grid cells to an output grid DEM as they are computed, since this could result in an I/O per cell (or per segment q). Instead we write each interpolated grid cell to a list along with its position (i, j) in the grid; we buffer B cells at a time in memory and write the buffer to disk when it runs full. After processing each set $\mathcal{S}_q$, we sort the list of interpolated grid cells by position to obtain the output grid. If the output grid has size T, computing the T interpolated cells and writing them to the list takes $O(T/B)$ I/Os. Sorting the cells take $O(\text{sort}(T))$ I/Os. Thus the interpolation phase is performed in $O(\text{scan}(N) + \text{sort}(T))$ I/Os in total.

For our initial experiments for the smooth approximation of data, we used the regularized spline with tension method described by Mitasova et al. [68]. This method models the surface as a thin plate spline under tension, and is an example of one of the many different spline methods that has been proposed for surface approximation. While seemingly complicated, this method has many advantages over other simpler approximation schemes. In particular, it can accurately compute secondary surface properties such as slope, profile curvature, and tangentail curvature, which are important in landform analysis and landscape process modeling. We present the details of the regularized spline with tension method here for completeness. The description here also introduces the tension and smoothing parameters whose effects we will explore experimentally in Chapter 6.

Given N input points $\{\vec{r_1}, \vec{r_2}, \ldots, \vec{r_N}\}$, where $\vec{r_i} = (x_i, y_i)$, each with a value z_i, the surface is defined by

$$z(\vec{r}) = a_1 + \sum_{j=1}^{N} \lambda_j R(\rho_j), \tag{2.1}$$

$$R(\rho_j) = -[E_1(\rho_j) + \ln \rho_j + C_E],$$

where $z(\vec{r})$ is the value at a point $\vec{r} = (x, y)$, a_1 is a constant trend, λ_j are a set of coefficients, and $R(\rho_j)$ is a radial basis function. In the function $R(\rho_j)$, $\rho_j = (\varphi|\vec{r} - \vec{r_j}|/2)^2$, where $|\vec{r} - \vec{r_j}|$ is a Euclidean distance function in $\mathbb{R}^2$, $C_E = 0.577215\ldots$ is the Euler constant, $E_1(\rho_j) = \int_{\rho_j}^{\infty} \frac{e^{-u}}{u} du$ is the exponential integral function, and φ is a tunable tension parameter. As $\varphi > 0$ is decreased, the approximation surface is tuned from acting like a rigid metal sheet to a flexible membrane.

The coefficients a_1 and λ_j are found by solving the following linear system of equations

$$a_1 + \sum_{j=1}^{N} \lambda_j [R(\rho_i) + \delta_{ij} w_0 / w_j] = z_i, \quad i = 1, \ldots, N \tag{2.2}$$

$$\sum_{i=1}^{N} \lambda_j = 0, \tag{2.3}$$

where w_0/w_j are positive weights representing a smoothing parameter for each point r_j. Setting the smooting parameter w_0/w_j to 0 results in an interpolation method where the surface must pass through all the input points. Increasing the smoothing for a particular point $\vec{r_i}$ allows the surface to approximate z_i at $\vec{r_i}$. A particular advantage of this method is that in addition to computing an interpolated surface, high order derivatives of the surface can be computed by direct evaluation of the derivative of $z(\vec{r})$.

2.5 Implementation

We implemented our methods in C++ using TPIE [22, 12], a library that eases the implementation of I/O-efficient algorithms and data structures by providing a set of primitives for processing large data sets. Our algorithm takes as input a set $\mathcal{S}$ of points, a grid size, and a parameter $k_{\max}$ that specifies the maximum number of points per quad tree segment, and computes the interpolated surface for the grid using our segmentation algorithm and a regularized spline with tension interpolation method [66]. We chose this interpolation method because it is used in the open source GIS GRASS module `v.surf.rst` [67]—the only GRASS surface interpolation method that uses segmentation to handle larger input sizes—and provides a means to compare our I/O-efficient approach to another segmentation method. Below we discuss two implementation details of our approach: thinning the input point set, and supporting a *bit mask*. Additionally, we highlight the main differences between our implementation and `v.surf.rst`.

2.5.1 Thinning Point Sets

Because lidar point sets can be very dense, there are often several cells in the output grid that contain multiple input points, especially when the grid cell size is large. Since it is not necessary to interpolate at sub-pixel resolutions, computational efficiency improves if one only includes points that are sufficiently far from other points in a quad-tree segment. Our implementation only includes points in a segment that are at least a user-specified distance ε from all other points within the segment. By default, ε is half the size of a grid cell. We implement this feature with no additional I/O cost simply by checking the distance between a new point p and all other points within the quad-tree leaf containing p and discarding p if it is within a distance ε of another point.

2.5.2 Bit Mask

A common GIS feature is the ability to specify a bit mask that skips computation on certain grid cells. The bit mask is a grid of the same size as the output grid, where each cell has a zero or one bit value. We only interpolate grid cell values when the bit mask for the cell has the value one. Bit masks are particularly useful when the input data set consists of an irregularly shaped region where the input points are clustered and large areas of the grid are far from the input points. Skipping the interpolation of the surface in these places reduces computation time, especially when many of the bit mask values are zero.

For high resolution grids, the number of grid cells can be very large, and the bit mask may be larger than internal memory and must reside on disk. Randomly querying the bit mask for each output grid cell would be very expensive in terms of I/O cost. Using the same filtering idea described in Section 2.2 and Section 2.3, we filter the bit mask bits through the quad-tree layer by layer such that each quad-tree segment gets a copy of the bit mask bits it needs during interpolation. If a quad-tree segment spans n_r rows and n_c columns in the output grid, the segment will store $n_r n_c$ bits of the bit mask in the segment. The bit-mask filtering algorithm uses $O(\frac{T}{B}\frac{h}{\log \frac{M}{B}})$ I/Os in total, where T is the number of cells in the output grid, which is $O(\text{sort}(T))$ when $h = O(\log N)$. The bits for a given segment can be accessed sequentially as we interpolate each quad-tree segment.

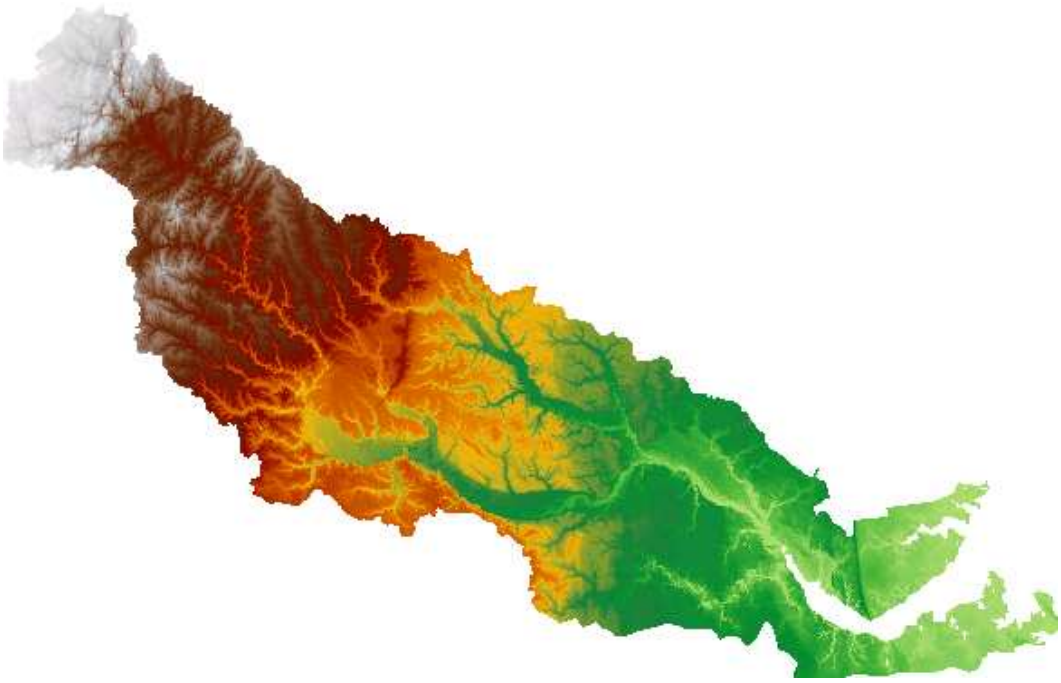


Figure 2.3: Neuse river basin data set

2.5.3 GRASS Implementation

The GRASS module `v.surf.rst` uses a quad-tree segmentation, but is not I/O-efficient in several key areas which we briefly discuss; constructing the quad tree, supporting a bit mask, finding neighbors, and evaluating grid cells. All data structures in the GRASS implementation with the exception of the output grid are stored in memory and must use considerably slower swap space on disk if internal memory is exhausted. During construction points are simply inserted into an internal memory quad tree using the incremental construction approach of Section 2.2. Thinning of points using the parameter ε during construction is implemented exactly as our implementation. The bit mask in `v.surf.rst` is stored as a regular grid entirely in memory and is accessed randomly during interpolation of segments instead of sequentially in our approach.

Points from neighboring quad-tree segment are not found in advance as in our algorithm, but are found when interpolating a given quad-tree segment q; the algorithm creates a window w by expanding q in all directions by a width δ and querying the quad tree to find all points within w. The width δ is adjusted by binary search until the number of points within w is between a user specified range $[n_{\min}, n_{\max}]$. Once an appropriate number of points is found for a quad-tree segment q, the grid cells in q are interpolated and written directly to the proper location in the output grid by randomly seeking to the appropriate file offset and writing the interpolated results. When each segment has a small number of cells, writing the values of the T output grid cells uses $O(T) \gg \text{sort}(T)$ I/Os. Our approach constructs the output grid using the significantly better $\text{sort}(T)$ I/Os.

2.6 Experiments

We ran a set of experiments using our I/O-efficient implementation of our algorithm and compared our results to prior GIS tools. We begin by describing the data sets on which we ran the experiments,

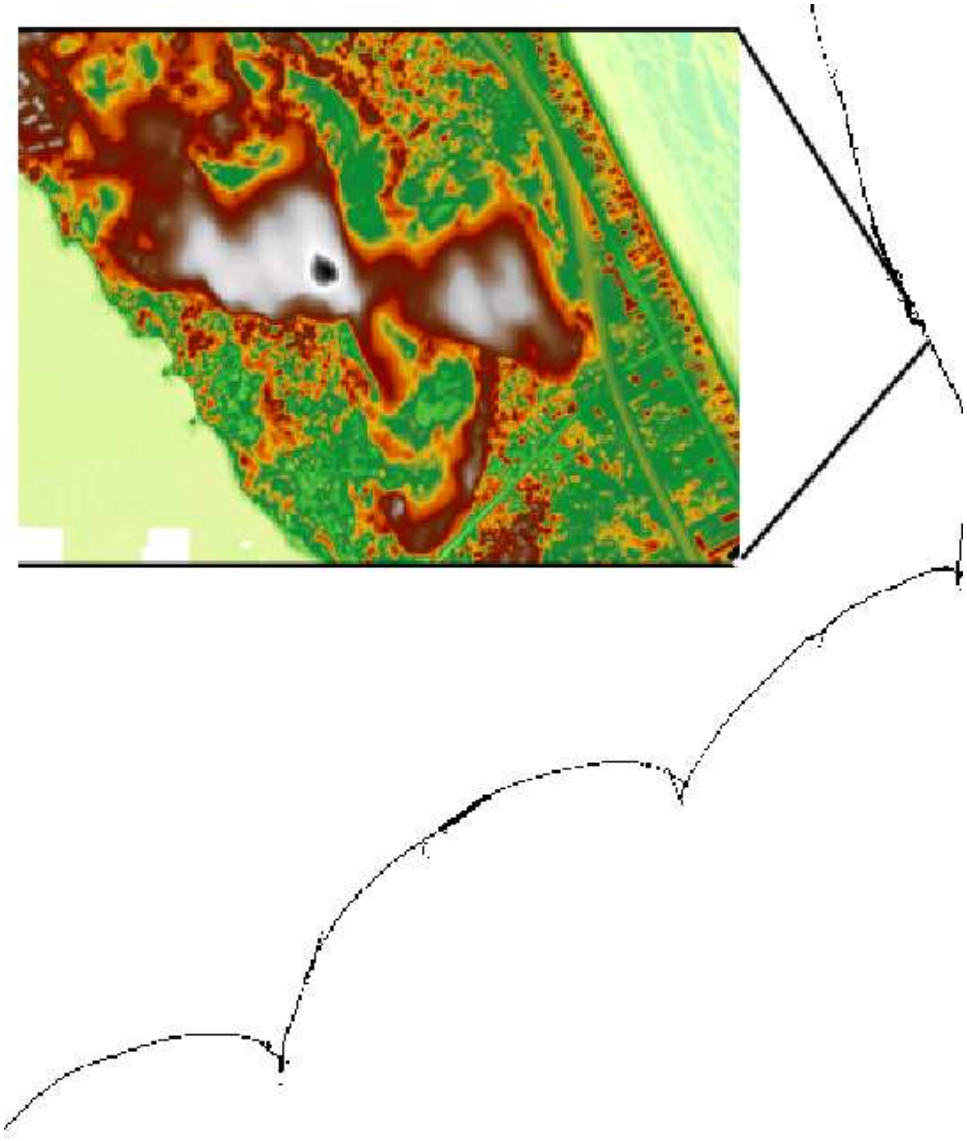


Figure 2.4: Outer Banks data set, with zoom to very small region.

then compare the efficiency and accuracy of our algorithm with other methods. We show that our algorithm is scalable to over 395 million points and over 53 billion output grid cells (where over 51 billion cells were masked out by the bitmask for the irregularly shaped coastal data set)–well beyond the limits of other GIS tools we tested.

2.6.1 Experimental Setup

We ran our experiments on an Intel 3.4GHz Pentium 4 hyper-threaded machine with 1GB of internal memory, over 4GB of swap space, and running a Linux 2.6 kernel. The machine had a pair of 400GB SATA disk drives in a non-RAID configuration. One disk stored the input and output data sets and the other disk was used for temporary scratch space.

For our experiments we used two large lidar data sets, freely available from online sources; one of the Neuse river basin from the North Carolina Floodmaps project [69] and one of the North Carolina Outer Banks from NOAA's Coastal Services Center [71].

Neuse River Basin. This data set contains 500 million points, more than 20 GB of raw data; see Figure 2.5.3. The data have been pre-processed by the data providers to remove most points on buildings and vegetation. The average spacing between points is roughly 20ft.

Outer Banks. This data set contains 212 million lidar points, 9 GB of raw data; see Figure 2.5.3. Data points are confined to a narrow strip (a zoom of a very small portion of the data set is shown in the figure). This data set has not been heavily pre-processed to remove buildings and vegetation. The average point spacing is roughly 3ft.

2.6.2 Scalability Results

We ran our algorithm on both the Neuse river and Outer Banks data sets at varying grid cell resolutions. Because we used the default value of ε (half the grid cell size) increasing the size of grid cells decreased the number of points in the quad tree and the number of points used for interpolation. Results are summarized in Table 2.1. In each test, the interpolation phase was the most time-consuming phase; interpolation consumed over 80% of the total running time on the Neuse river basin data set. For each test we used a bit mask to ignore cells more than 300ft from the input points. Because of the irregular shape of the Outer Banks data, this bit mask is very large, but relatively sparse (containing very few "1" bits). Therefore, filtering the bit mask and writing the output grid for the Outer Banks data were relatively time-consuming phases when compared to the Neuse river data. Note that the number of grid cells in the Outer Banks is roughly three orders of magnitude greater than the number of quad-tree points. As the grid cell size decreases and the total number of cells increases, bit mask and grid output operations consume a greater percentage of the total time. At a resolution of 5ft, the bit mask alone for the Outer Banks data set is over 6GB. Even at such large grid sizes, interpolation—an internal memory procedure—was the most time-consuming phase, indicating that I/O was not a bottleneck in our algorithm.

We also tried to test other available interpolation methods, including `v.surf.rst` in the open source GIS GRASS; kriging, IDW, spline, and topo-to-raster (based on ANUDEM [57]) tools in ArcGIS 9.1; and QTModeler 4 from Applied Imagery [8]. Only `v.surf.rst` supported the thinning of data points based on cell size, so for the other programs we simply used a subset of the data points. None of the ArcGIS tools could process more than 25 million points from the Neuse river basin at 20ft resolution and every tool crashed on large input sizes. The topo-to-raster tool processed the largest set amongst the ArcGIS tools at 21 million points.

The `v.surf.rst` could not process more than 25 million points either. Using a resolution of 200ft, `v.surf.rst` could process the entire Neuse data set in six hours, but the quad tree only

Dataset	Neuse		Outer Banks	
Resolution (ft)	20	40	5	10
Output grid cells ($\times 10^6$)	1360	340	53160	13402
quad-tree points ($\times 10^6$)	395	236	128	66
Total Time (hrs)	53.0	24.4	17.7	6.9
Time spent to... (%)				
Build tree	2.0	3.8	4.5	8.6
Find Neighbors	10.6	15.1	14.5	16.4
Filter Bit mask	0.2	0.3	13.1	8.0
Interpolate	86.4	80.4	52.6	57.8
Write Output	0.8	0.4	15.3	9.2

Table 2.1: Results from the Neuse river basin and the Outer Banks data sets.

contained 17.4 million points. Our algorithm processed the same data set at 200ft resolution in 3.2 hours. On a small subset of the Outer Banks data set containing 48.8 million points, `v.surf.rst`, built a quad tree on 7.1 million points and computed the output grid DEM in three hours, compared to 49 minutes for our algorithm on the same data set.

The QTModeler program processed the largest data set amongst the other methods we tested, approximately 50 million points, using 1GB of RAM. The documentation for QTModeler states that their approach is based on an internal memory quad tree and can process 200 million points with 4GB of available RAM. We can process a data set almost twice as large using less than 1GB of RAM.

Overall, we have seen that our algorithm is scalable to very large point sets and very large grid sizes and we demonstrated that many of the commonly used GIS tools cannot process such large data sets. Our approach for building the quad tree and finding points in neighboring segments is efficient and never took more than 25% of the total time in any of our experiments. The interpolation phase, an internal step that reads points sequentially from disk and writes grid cells sequentially to disk, was the most time-consuming phase of the entire algorithm.

2.6.3 Comparison of Constructed Grids

To show that our method constructs correct output grids, we compared our output on the Neuse river basin to the original input points as well as to grid DEMs created by `v.surf.rst`, and DEMs freely available from NC Floodmaps. Because `v.surf.rst` cannot process very large data sets, we ran our tests on a small subset of the Neuse river data set containing 13 million points. The output resolution was 20ft, ε was set to the default 10ft, and the output grid had 3274 rows and 3537 columns for a total of 11.6 million cells. Approximately 11 million points were in the quad tree.

The interpolation function we tested used a smoothing parameter and allowed the input points to deviate slightly from the interpolated surface. We used the same default smoothing parameter used in the GRASS implementation and compared the distribution of deviations between the input points and the interpolated surface. The results were independent of $k_{\max}$, the maximum number of points per quad-tree segment. In all tests, at least 79% of the points had no deviation, and over 98% of the points had a deviation of less than one inch. Results for `v.surf.rst` were similar. Since the results were indistinguishable for various $k_{\max}$ parameters, we show only one of the cumulative distribution functions (CDF) for $k_{\max} = 35$ in Figure 2.5.

Next, we computed the absolute deviation between grid values computed using `v.surf.rst` and our method. We found that over 98% of the cells agreed within 1 inch, independent of $k_{\max}$. The methods differ slightly because `v.surf.rst` uses a variable size window to find points in neighboring points of a quad-tree segment q and may not choose all points from immediate neighbors of q

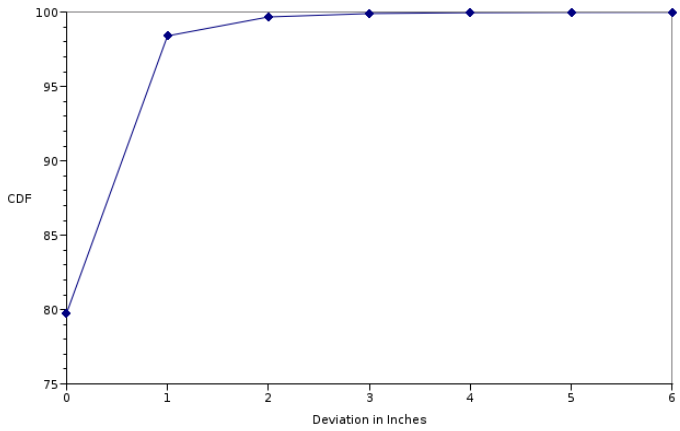


Figure 2.5: Distribution of deviations between input points and interpolated surface ($k_{\max} = 35$).

when the points are dense and may expand the window to include points in segments that are not immediate neighbors of q when the points are sparse. In Figure 2.6.3 we show a plot of the interpolated surface along with an overlay of cells where the deviation exceeds 3 inches. Notice that most of the bad spots are along the border of the data set where our method is less likely to get many points from neighboring quad-tree leaves and near the lake in the upper left corner of the image where lidar signals are absorbed by the water and there are no input data points.

Finally, we compared both our output and that of `v.surf.rst` to the 20ft DEM data available from the NC Floodmaps project. A CDF in Figure 2.8 of the absolute deviation between the interpolated grids and the "base" grid from NC Floodmaps shows that both implementations have an identical CDF curve. However, the agreement between the interpolated surfaces and the base grid is not as strong as the agreement between the algorithms when compared to each other. An overlay of regions with deviation greater than two feet on base map shown in Figure 2.7(a) reveals the source of the disagreement. A river network is clearly visible in the figure indicating that something is very different between the two data sets along the rivers. NC Floodmaps uses supplemental break-line data that is not part of the lidar point set to enforce drainage and provide better boundaries of lakes in areas where lidar has trouble collecting data. Aside from the rivers, the interpolated surface generated by either our method or the prior GRASS implementation agree reasonably well with the professionally produced and publicly available base map. Furthermore, it was recently observed by Hodgson et al. [55], that the mean absolute error and the RMSE of the lidar signals themselves are 8.7 inches and 13.0 inches respectively in smooth open terrain and these errors can be over two feet in forested or mixed cover terrain. Therefore, our deviations are for the most part well within the error bounds of the the original lidar data.

2.7 Conclusions

In this chapter we describe an I/O-efficient algorithm for constructing a grid DEM from point cloud data. We implemented our algorithm and, using lidar data, experimentally compared it to other algorithms. The empirical results show that, unlike prior algorithms, our approach scales to data sets much larger than the size of main memory. Although we focused on elevation data, our technique is general and can be used to compute the grid representation of any bivariate function from irregularly sampled data points.

For future work, we would like to consider a number of related problems. Firstly, our solution is constructed in such a way that the interpolation phase can be executed in parallel. A parallel implementation should expedite the interpolation procedure. Secondly, as seen in Figure 2.7(a), grid

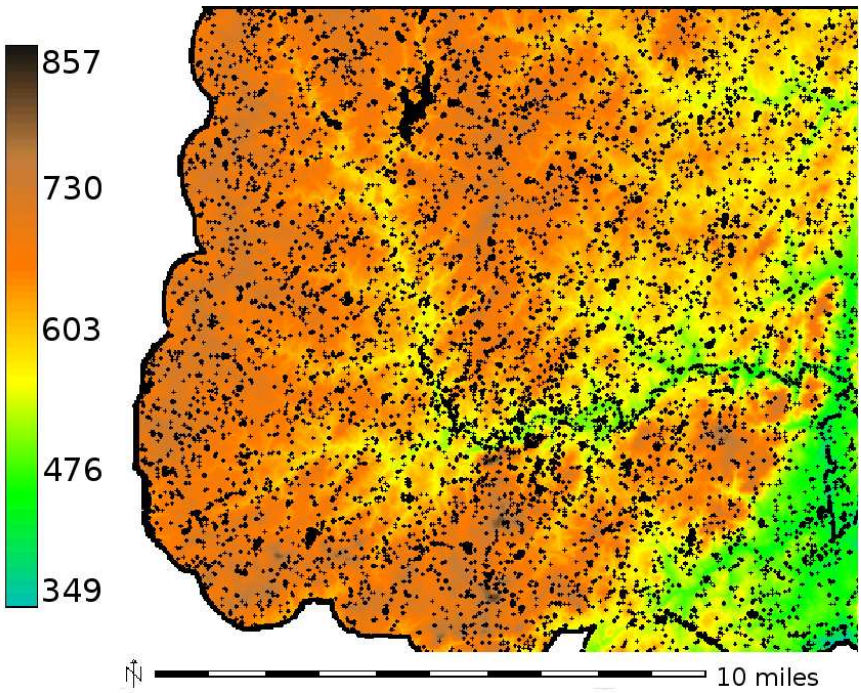
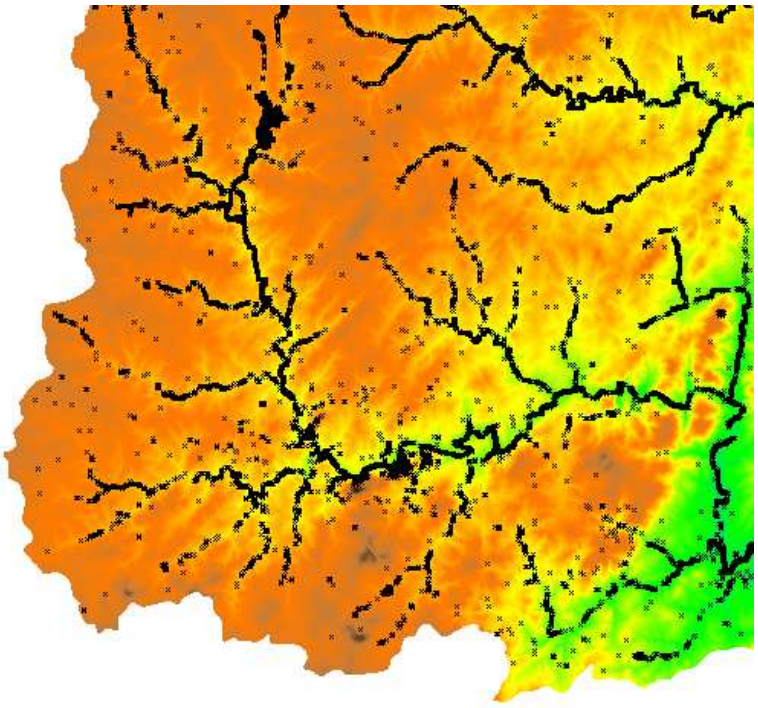


Figure 2.6: Interpolated surface generated by our method. Black dots indicate cells where the deviation between our method and `v.surf.rst` is greater than three inches.

DEMs are often constructed from multiple sources, including lidar points and supplemental breaklines where feature preservation is important. Future work will examine methods of incorporating multiple data sources into DEM construction. Finally, the ability to create large scale DEMs efficiently from lidar data could lead to further improvements in topographic analysis including such problems as modelling surface water flow or detecting topographic change in time series data.



(a)

Figure 2.7: Interpolated surface generated by our method. Black dots indicate cells where the deviation between our method and ncfloodmap data is greater than two feet.

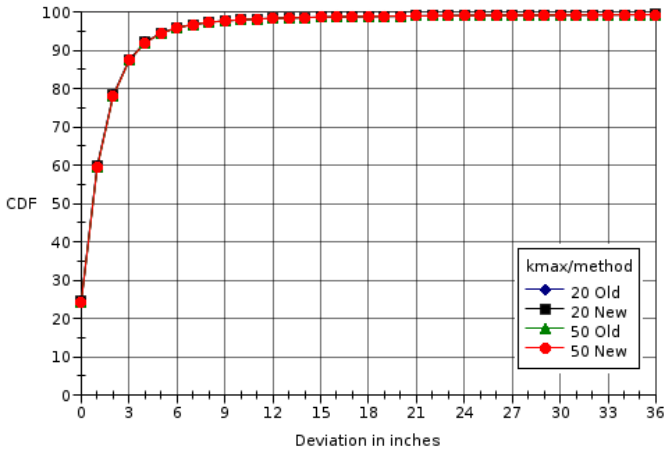


Figure 2.8: Cumulative distribution of deviation between interpolated surface and data downloaded from ncfloodmaps.com. Deviation is similar for both our method and `v.surf.rst` for all values of $k_{\max}$.

Chapter 3

Flow Modelling on Grid Terrains

3.1 Introduction

Given a grid DEM, one can model the flow of water over the terrain. A typical flow modelling approach consists of two phases. In the first phase, called *flow routing*, we compute for each vertex u in the grid a set of directed edges $\{(u, v_1), (u, v_2), \ldots, (u, v_k)\}$ where an edge (u, v) means that a portion of water that arrives in vertex u is routed to the vertex v. The actual means of choosing these edges depends on any one of many flow routing models available. Typically flow is only routed to neighbors with a lower elevation. In the second *flow accumulation* phase, we place an initial amount of water or *flow* in each vertex v and route flow along the directed edges created in the flow routing phase. The flow accumulation of a vertex u is the sum of the initial amount of flow placed in u plus the sum of any incoming flow routed to u along edges created during the flow routing phase. Flow routing and flow accumulation are typically used as input for river network extraction and watershed delineation. By choosing all edges from the flow routing phase whose endpoints have a flow accumulation above a user-specified threshold, one can extract a river network. Extraction of watershed hierarchies from a river network is discussed in Chapter 4.

Typical flow-modeling algorithms assume that water flows downhill until it reaches a local minimum, or *sink*. The DEMs constructed in Chapter 2 and DEMs from other data providers may have many such sinks. Some sinks in the DEM are due to noise in either the elevation data point sample or the construction method. Other sinks are caused by real features such as bridges appearing as dams in the DEM. Still other sinks correspond to real geographic features such as quarries, sinkholes, or closed water basins with no drainage outlet. Sinks due to noise or bridges are problematic because they impede the correct flow of water and result in artificially disconnected river networks. Therefore, it is important to modify DEMs to remove sinks that do not correspond to significant sinks such as quarries and naturally closed drainage basins. We call a terrain that has been modified to make the flow routing and flow accumulation more closely match the real river networks *hydrologically conditioned*, or *hydrologically correct*.

Ideally, only those sinks due to noise should be removed while genuine sinks should be preserved. Agarwal et al. recently proposed a method [6] based on a topological persistence technique developed by Edelsbrunner et al. [45, 44] that assigns a numerical persistence value to each sink in a DEM. Sinks that are likely due to noise or bridges have a lower persistence than significant natural sinks like quarries. This scoring allows the user to remove sinks below a specified persistence threshold. In this chapter, we review the definition of topological persistence, show how to compute the persistence of each sink in a height graph, and describe an efficient method of removing sinks via *flooding*. It is important to note that the method of ranking the sinks is not restricted to a particular sink removal method and it may be possible to use other methods of sink removal other than those described in this chapter. We review methods for routing flow in Section 3.4. These methods only work when a grid vertex has at least one downslope neighbor. On extended flat areas of constant height where at least one vertex in the area has a downslope neighbor, or *spill point*, it is possible to route flow across the flat area towards one or more spill points. We describe a new method for detecting flat areas in Section 3.4.1 and review previous methods for routing flow across a single flat area in Section 3.4.2. In Section 3.5 we review the standard I/O-efficient flow accumulation algorithm which was previously implemented by Arge et al. in the software package TerraFlow [14]. Finally, in Section 3.6, we examine current problems in using previous methods for flow modeling on modern hi-resolution DEMs, and describe potential research directions that could improve flow modeling on

this new generation of DEMs. Before starting the discussion on topological persistence, we review a few preliminary concepts that will be helpful throughout the remainder of the chapter.

3.1.1 Height Graph

To put flow modeling in a more theoretical framework, we first introduce the notion of the *height graph* of a grid DEM. A height graph $G = (V, E)$ is an undirected graph, with a *height* $h(v)$ and an *ID* $id(v)$ associated with each $v \in V$. The IDs are assumed to be unique, but the heights may not be. For any two vertices u and v, we say u is *higher than* v if $h(u) > h(v)$, or $h(u) = h(v)$ and $id(u) > id(v)$; u is *strictly higher* than v if $h(u) > h(v)$. The concepts of *lower* and *strictly lower* are defined similarly. The grid cells of a grid DEM define the vertices of the height graph. The unique ID of a height graph vertex is the the row, column pair of the grid cell location. The edges of the height graph in are not uniquely defined, and they depend on the application. A typical approach is to add all the boundary edges of the grid cells and add one of the diagonals for each grid cell. This results in a *triangulated regular network*. However in some applications, one wishes to connect each grid point to its eight neighbors, in which case we add both diagonals for each grid cell, thereby resulting into a non-planar graph. We do not commit to any particular representation and let the user decide which of the diagonals to add. A grid vertex is on the boundary of a the DEM if it is either on the boundary of the bounding box of the grid, or is adjacent to a vertex with a special *null* or *nodata* value. We add a vertex ξ with $h(\xi) = -\infty$, which is connected to all the vertices on the boundary of the DEM.

3.1.2 Batched Union-Find

Another important data structure for flow modeling is the batched-union find data structure. In the union-find problem we maintain a partition of a set $U = x_1, x_2, \ldots, x_N$ of N elements during a sequence of UNION(x_i, x_j) and FIND(x_i) operations. A FIND(x_i) operation returns a representative element of the set containing x_i and a UNION(x_i, x_j) joins the two sets containing x_i and x_j. In the online version of union-find, the sequence of operations is not known in advance, but in the *batched* union-find problem, the entire sequence of UNION and FIND operations is known. Agarwal et al. [6] recently developed a theoretically I/O-efficient algorithm for batched union-find that runs in $O(\text{sort}(N))$ I/Os, where N is the number of mixed UNION and FIND operations. The authors also present a practical, $O(\text{sort}(N) \log(N/M))$-I/O algorithm. The algorithm can easily be modified such that if each element x_i in the set has a weight $h(x_i)$ then FIND(x_i) returns the element with the lowest weight in the set containing x_i.

3.1.3 Processing topologically sorted DAGs

Suppose we are given a directed acyclic graph (DAG) that is topologically sorted by some totally ordered set. We wish to evaluate a function for each vertex v in the DAG, where the function value depends on the value of vertices whose outgoing edges terminate at v. For our purposes, we make the assumption that the in-degree and out-degree of each vertex in the DAG is bounded by some small constant. A technique called time-forward processing [34, 10] was developed for evaluating such a function on a topologically sorted DAG I/O-efficiently in $O(\text{sort}(N)$ I/Os.

The intuition behind this approach is that if we process vertices according to their topological order, the function for all inputs to a vertex v will be computed before we process v. In essence, when we process a vertex, we can send the results forward in "time" to be processed by other vertices that depend on the result. Arge developed a practical sort(N) algorithm [10] based on a priority queue for solving this problem that improves an earlier solution by Chiang et al. [34]. The basic idea in the priority queue approach is that when we compute the value for a vertex u, we insert the result of the function into the priority queue with priority v for each edge (u, v) in the DAG.

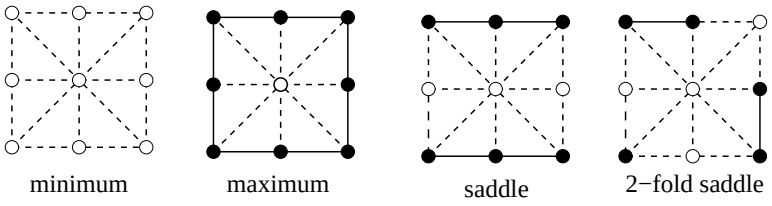


Figure 3.1: Classification of vertices in height graph. Hollow circles are higher neighbors while filled circles are lower.

When we need to process vertex v, all smaller elements have been processed, since we processed the vertices in topological order, and we can extract the inputs for vertex v from the priority queue.

3.2 Topological Persistence

The notion of topological persistence is introduced in the context of Morse functions [45, 44] but can be extended to the planar height graph described in Section 3.1.1. We define a minimum in the height graph is a vertex with no lower neighbors. If we look at the adjacent neighbors of a height graph vertex v in clockwise order, we consider a vertex to be a saddle if there are at least two disjoint sequences of adjacent neighbors of v lower than the height of v (see [42] for the precise definition of a saddle). See Figure 3.1 for the various classifications of height graph vertices. Topological persistence pairs each minimum and maximum with a saddle point. The persistence of a minimum or maximum is the height difference between the minimum or maximum and the saddle with which it is paired. For our purposes, we are only concerned with the pairing of minima and saddles. The maxima-saddle pairing is symmetric. Conceptually, the persistence of minima in a height graph are computed by sweeping a plane bottom-up over the height graph and maintaining the set of connected components in the height graph of vertices below the sweep plane. Each component is represented by an un-paired minimum. When the sweep-plane encounters a minimum, a new component is created. When the sweep-plane encounters a k-fold saddle, $k+1$ components are merged. The persistence of the k highest un-paired minima in the merged component are computed at this point, leaving the lowest minimum u un-paired. The new merged component is represented by u.

Agarwal et al. [6, 95] developed an I/O-efficient algorithm for computing the topological persistence of a height graph using the batch-union find structure [6] described in Section 3.1.2 in $O(\text{sort}(N))$ I/Os. In practice, the practical $O(\text{sort}(N)\log\left(\frac{N}{M}\right))$ algorithm [6] for batched union-find is used. Their algorithm computes for each minimum u, the persistence of u and the representative minimum v that is the representative of the merged component when u is paired with a saddle. For the purpose of sink removal, it will later be helpful to store the results of the persistence computation as a *merge-tree* defined as follows. Each vertex v in the tree is a minimum, and stores the height of the saddle $s(v)$ paired with v. The persistence of each vertex can easily be computed as the difference in heights $h(s(v)) - h(v)$. We create a directed edge (u, v) between a child vertex u and parent vertex v in the merge-tree to indicate that when the component represented by u was paired with a saddle, v was the representative of the merged component. Such a a merge-tree can be constructed in a scan of the persistence computation results.

We use persistence to measure the importance of sinks on a terrain. For a user-specified threshold τ, we declare all sinks with persistence greater than τ to be significant sinks that should be preserved, while other sinks should be removed. The user can change the threshold to control the smallest feature size to be preserved. Thus we get an automated way for marking sinks for removal that preserves major features on the terrain based on their persistence. A sink removal method that removes all sinks are equivalent to setting $\tau = \infty$. Once the user specifies an appropriate threshold

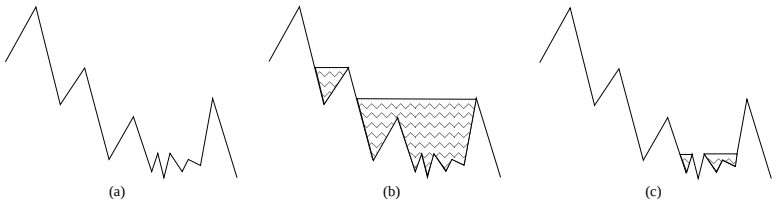


Figure 3.2: (a) Original terrain. (b) Flooding of all internal sinks (c) Partial flood sinks with low persistence.

that may be application dependent, we can remove sinks below the threshold using some sink removal algorithm. We next describe a *flooding* approach [59] that can be made scalable to large data sets.

3.3 Sink removal

Formally, flooding can be defined as follows [14]. Let G be a height graph with one or more significant sinks represented by the lowest minima (or an "outside" vertex) $\zeta_1, \ldots, \zeta_k$ in each significant sink. Let the *height of a path* be the height of the highest vertex on the path, and let the *raise elevation* of a vertex v of G be the minimum height of all paths from v to ζ_i for all $1 \leq i \leq k$. The flooding problem is to find the raise elevations for all vertices in G, after which the height of each vertex is modified to its raise elevation. After flooding, it is clear that the each vertex in G has a path of monotonically non-increasing height to a significant sink. An $O(\text{sort}(N))$-I/O algorithm for flooding when only one significant sink, represented by ζ exists is given by Arge et al.[14] and is used in their scalable flow modeling software project TerraFlow.

In real terrains however, multiple significant sinks exist and a more general approach than that of Arge et al. is needed to flood the terrain. Figure 3.2(a) shows an unmodified terrain that is modified via flooding in Figure 3.2(b) to remove all sinks except the global outside sink. In Figure 3.2(c), only those sinks with a small persistence have been modified via flooding.

Consider classifying the set of minima in the height graph with topological persistence above a user-specified threshold τ to be the representatives of the set of significant sinks. Note this includes the outside sink which has a persistence of ∞. We can flood the height graph to remove all sinks with persistence less than τ as defined above in three phases, such that only the sinks marked as significant remain after flooding. In the first phase we assign a sink label to each vertex in the height graph. If a vertex u has a sink label v then there is a path of monotonically decreasing height in the height graph G from u to a minimum v. If a vertex u has paths to multiple minima $v_1, v_2, \ldots, v_k$, the sink label can be any of the v_i for which such a path exits. In the second phase, we compute for each minimum u in the height graph, the raise elevation of u. We will prove that for any vertex v with sink label u the raise elevation of v is the maximum of the elevation of v and the raise elevation of u. In the third phase, we scan the list of vertices and height graph and raise the elevation of each vertex u to the raise elevation v of its sink label if $h(v) > h(u)$. We describe each of these phases in detail below.

3.3.1 Computing Sink Labels

To compute sink labels for all vertices in the height graph, we can use the technique for evaluating a function on a topologically sorted DAG described in Section 3.1.3. We can consider the height graph G of Section 3.1.1 as a DAG if for each edge (u, v), we direct the edge from u to v if u is lower than v. Since the lower than relation in the height graph defines a total order, the vertices in the DAG are topologically ordered by the lower than relation. Each minimum in the height graph

is a source in the DAG. We assign an initial sink-label u for each minimum u in the DAG. For each internal vertex in the DAG, the inputs to the DAG are one or more sink labels. For each internal vertex v, we chose one of these labels, u_i (for completeness, we choose the smallest such u_i), to be the sink label for v and forward u_i across all outgoing edges starting from v. In this way, we can compute a single sink-label for each vertex in the height graph. To see that the algorithm is correct, note that if a vertex v receives a label u in the circuit evaluation, then there must be a path of monotonically non-increasing height from v to u in the height graph since vertices are processed in topological (height) order and edges only go from vertices of lower height to vertices of equal or higher height. But this is precisely the definition of a sink-label for v, so the algorithm is correct. Since each vertex in the height graph has degree at most eight in the grid case, we can evaluate the DAG on all N vertices in the height graph in sort(N) I/Os as described in Section 3.1.3.

3.3.2 Computing Raise Elevations

To compute the raise elevation for all sinks in the terrain, we use the merge-tree produced by the topological persistence algorithm [6]. The merge-tree has the property that on any root-to-leaf path the heights of the vertices and increase, while the heights of the saddles paired with each minimum decrease along any root-to-leaf path. To see that the heights of vertices increase on any root to leaf path, note that if v is a parent of u then two components of the height graph represented by minima u and v must have merged and, by definition of the merge-tree and the persistence algorithm, the parent of u must be the minima with the lowest height of all merged minima. Furthermore, since v is not paired with saddle when the component of u merges with the component of v, v must merge when the sweep-plane is higher than when e merged with v. Thus the height of the saddle $s(v)$ that is paired with v has an elevation higher than the saddle that is paired with u. Since persistence is the height difference between a minimum and its paired saddle and v is lower than u, the persistence of v, $h(s(v)) - h(v)$ must be higher than $h(s(u)) - h(u)$ and the persistence of vertices also decrease along any root-leaf path in the merge-tree.

Lemma 1 *Let v be a vertex in the merge tree that is not a significant sink, but whose parent is a representative of a significant sink. Let s be the saddle with height $h(s)$ that is paired with v. Let u be any vertex in the sub-tree rooted at v. The raise elevation $r(u)$ of u is $h(s) = r(v)$.*

Proof. All representative minima of significant sinks in the merge tree have persistence above the user-specified persistence threshold τ by definition. Persistence values of vertices in the merge tree decrease on any root-leaf path as argued above, and therefore the set of representative minima of significant sinks form a connected sub-tree in the merge-tree rooted at the root of the merge-tree. Because v is not a significant sink, it is rooted below the sub-tree of significant sinks and no significant sinks can be in the sub-tree rooted at v. See Figure 3.3 for an illustration of the merge-tree structure.

We first show that $r(u) \leq h(s)$. Consider the bottom-up sweep of the height graph. The component represented by v merges with the component represented by the parent of v when the sweep plane is at height $h(s)$ by definition of the minimum-saddle pairing. Before the merge, all vertices in the component of v have height below $h(s)$. This includes all vertices in the component of u since u is in the subtree of v and must have merged into the component of v before the sweep plane was at height $h(s)$. Therefore there is a path from u to any vertex in the component of v with height less than $h(s)$. After the component of v merges with the component of the parent of v at height $h(s)$, there is a path from u to a representative minimum of a significant sink of height $h(s)$, so $r(u) \leq h(s)$.

To show that $r(u) \geq h(s)$, assume that this is not the case and $r(u) < h(s)$. Then there must be a path of height less than $h(s)$ from u to a representative minimum w of a significant sink. Therefore, a component containing u must have merged with a component containing w when

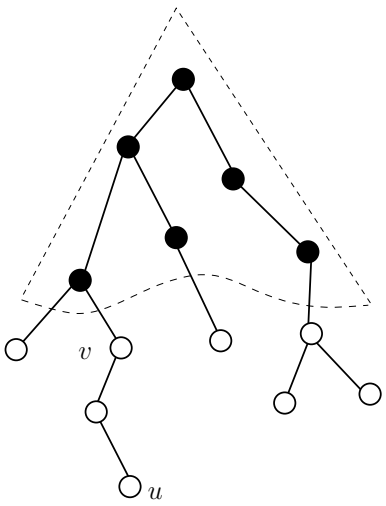


Figure 3.3: An example merge tree where the significant sinks marked in black with persistence greater than a user-specified threshold τ form a sub-tree.

the sweep plane was below $h(s)$. Furthermore, the component containing v must merge with the component containing u when the sweep plane is below $h(s)$ because u is in the sub-tree of v. By our assumption that $r(u) < h(s), w, u$ and v are in the same component when the sweep plane is below $h(s)$. Since w is the representative of a significant sink, it has a higher persistence than either u or v, and v must therefore be paired with a saddle when the component containing v merges with the component containing w. Since this occurs when the sweep plane is below $h(s)$, this saddle must have a height less than $h(s)$, but this contradicts the definition that $h(s)$ is precisely the height of the saddle paired with v. Therefore, $r(u) \geq h(s)$. Because $h(s) \leq r(u) \leq h(s)$, we conclude that $r(u) = h(s)$. □

Lemma 1 gives us a simple way to compute the raise elevations of all minima in the merge-tree. Note that if u is a representative of a significant sink, we do not need to raise u at all and the raise elevation of u is simply the height of u. We simply need to compute the raise elevation of minima representing significant sinks. We process all the edges (u, v) in the tree in increasing order of the height of v. Once we see an edge (u, v) where v is a representative of a significant sink but u is not, we propagate $r(u)$ to all vertices in the subtree rooted at u. The whole process can be easily implemented using BFS if the tree fits in memory. Otherwise, we construct a topologically sorted DAG by directing edges in the merge tree from a parent v to a child u. The height of the minima form a topologically sorted order of the tree because heights increase as we go from a child to a parent. When evaluating the raise elevation of a vertex v, if v is the representative of a significant sink, we set $r(v)$ to be the height of v and propagate a null value along all outgoing edges of the DAG originating at v. If v is not the representative of a significant sink and the incoming circuit value is null, then the parent of v is the representative of a significant sink and $r(v)$ is the height $h(s)$ of the saddle s paired with v. The raise elevation of v is then propagated along all outgoing edges originating at v. If v is not the representative of a significant sink and the incoming value for a DAG vertex is not null, we set $r(v)$ to be the incoming raise elevation and propagate this value along the outgoing edges of the DAG. It is clear that evaluating the DAG in topologically sorted order correctly computes the raise elevation of each minimum in the merge-tree and can be done using I/O-efficient processing of topologically sorted DAGs in $O(\text{sort}(N))$ I/Os.

To compute the raise elevation of all vertices in the height graph, we prove the following lemma.

Lemma 2 *The raise elevation of a vertex u in the height graph with elevation $h(u)$ and sink label v is $r(u) = \max\{h(u), r(v)\}$.*

Proof. Because u has a sink label v, there is a path of monotonically non-increasing height from u to v. By definition, there is a path of height $r(v)$ from v to a significant sink. We consider two cases; $h(u) \geq r(v)$ and $h(u) < r(v)$. In the first case, where $h(u) \geq r(v)$, there is a path of height $h(u)$ from u to a significant sink through v. Since the raise elevation $r(u)$ of u can be no less than the height of u, $r(u) = h(u)$.

In the second case where $h(u) < r(v)$, note that v cannot be a significant sink because if v is a significant sink then $r(v) = h(v)$ and because u has a sink label v there is a path of monotonically non-increasing height from u to v and $h(u) \geq h(v) = r(v)$. So there is at least a path of height $r(v)$ from u to a significant sink through v. Suppose there is a path of height $r(u)$, where $h(u) < r(u) < r(v)$ from u to a significant sink. Then since there is path from v to u of height $h(u)$, then there is a path from v to a significant sink of height $r(u)$ through u. And the raise elevation of v should be no more than $r(u)$. But this contradicts the assumption that $r(v) > r(u)$ and thus $r(u) = r(v)$. □

3.3.3 Flooding the Terrain

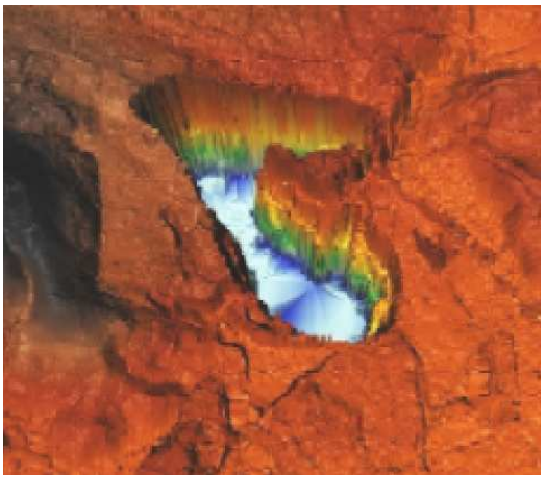
The final phase of flooding is quite easy once we have computed sink-labels for each vertex in the height graph and the raise elevation for each sink in the height graph. We simply sort the vertices of the height graph by sink-label and sort the raise elevations of each sink by their sink-label. Then we simultaneously scan both lists and for each vertex u in the height graph that is below the raise elevation of its corresponding sink-label v, we raise the elevation of u to the raise elevation of v. By Lemma 2 this computes the correct raise elevation for u. The sort and scan can be done in $O(\text{sort}(N))$ I/Os. If needed, we can put the terrain back in grid order by sorting the vertices of the height graph by grid order (row, column) in $\text{sort}(N)$ additional I/Os.

An example is shown in Figure 3.4, which shows a portion of the terrain in the Neuse River Basin. With a persistence threshold of $\tau = 30$ most sinks, particularly in the area in the lower part of the figure, have been removed, while the major features such as the quarry have been preserved. On the other hand, a flooding procedure that removes all sinks has undesirably eradicated some major features, including the quarry.

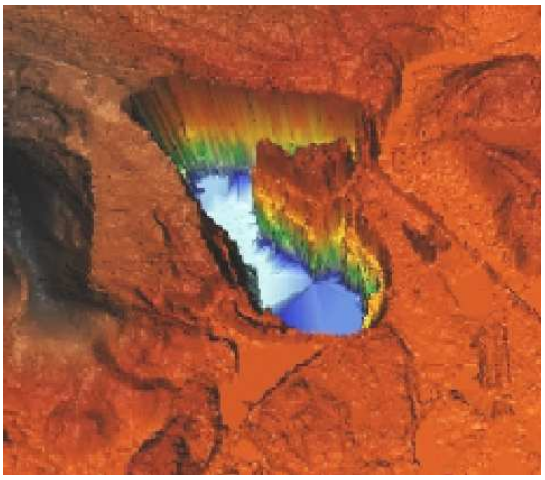
3.4 Flow Routing

In this section we review flow routing models for grid vertices with at least one downslope neighbor. These models cannot route flow on vertices that are flat and have no downslope neighbors. However, it is sometimes possible to route flow across a flat plateau towards a vertex with a downslope neighbor. In Section 3.4.1 we describe a new and practical way of detecting all flat areas containing vertices with no downslope neighbors. In Section 3.4.2 we review methods for routing flow across flat areas.

Given a height graph $G = (V, E)$ of a terrain as described in Section 3.1.1, flow routing constructs a *flow graph* $\mathcal{F}(G) = (V, E_r)$ that is a directed subgraph of G, i.e., $E_r \subseteq E$. An edge (u, v) in $\mathcal{F}(G)$ indicates that water can flow from u to v. We construct E_r from G by looking at each vertex u and its neighbors and applying a flow direction model. A number of such models have been proposed in the GIS literature, see e.g., [59, 72, 46, 94, 82]. We describe two commonly used methods; the *single-flow-direction* (SFD or D8) model and the *multi-flow-direction* (MFD) model. In the SFD model, we select the downslope edge (u, v) which has the steepest non-zero gradient $\frac{h(u)-h(v)}{d(u,v)}$, where $h(u)$ is the height of a vertex u in G and $d(u, v)$ is the distance between the projection of u and v onto the horizontal plane. This SFD model is a convergent flow model as flow can never diverge to two downslope neighbors. The resulting flow graph is a forest of trees. In the MFD model we select



(a)



(b)

(c)

Figure 3.4: (a) Original terrain. (b) Terrain flooded with persistence threshold $\tau = 30$. (c) Terrain flooded with $\tau = \infty$.

all edges (u, v_i) with $h(v_i) < h(u)$. The MFD model can model divergent flow and the resulting flow graph is a DAG which may have multiple disconnected components. Depending on the application, one model may be preferred over another.

In all direction models, flow is routed to neighbors with a lower height. Sinks and flat areas in the terrain will have no outgoing edges according to these flow direction models. If the terrain has insignificant sinks that are not significant terrain features, the flow graph will be broken into a number of disconnected components. In reality, water flows between two vertices in the DEM in the two disconnected components of the flow graph. To avoid this situation and have realistic flow graph connectivity, it is often desirable to remove insignificant sinks. We can use the I/O-efficient algorithm of Section 3.3 to hydrologically condition the DEM and remove these sinks. However, using flooding to remove sinks creates flat areas in which the flow routing models cannot select any edges for the flow graph. To address this issue, we must be able to route flow across flat areas.

3.4.1 Detecting Flat Areas

We define a vertex u in the height graph G to be *flat* if either $h(u) \leq h(v)$ for all neighbors v of u in G, or if u has a neighbor of the same height that has no downslope neighbor. A flat area is connected component of flat vertices in the height graph that have the same height. Flat areas can either be plateaus or sinks. We define a flat to be a plateau if there is at least one vertex in the flat area with a downslope neighbor. We call a vertex on a plateau with at least one downslope neighbor a *spill point*. Intuitively, as water falls on the interior of a plateau, it will spread out and eventually run off the plateau at a spill point. A flat area is an extended sink if there are no spill points on the flat area. In the case of plateaus, we would like to route flow across the plateau and towards the spill point. Figure 3.5 shows a terrain that has been hydrologically conditioned using flooding to remove insignificant sinks. A bridge on the left side created a sink on the right hand side which was flooded. Much of the terrain in the center of the figure is one large flat area as a result of the flooding. There is a spill point in the lower left of the image where water can flow over a bridge. By routing flow across the flat area and towards the spill point, we can connect two components of the flow graph that would be disconnected if we did not route flow across the flat area. In Figure 3.5, if we did not route flow across the flat area there would be a component of the flow graph to the right of the bridge near the flat area that would be disconnected from a component downstream to the left of the bridge. Routing flow across the flat area will connect these two flow graph components and create flow connectivity similar to the actual river network in the terrain. In this section we describe a new algorithm for detecting flat areas on grid DEMs and assigning a unique connected components label to all vertices in the same flat area. This algorithm is more practical than an previous theoretically optimal scan(N) algorithm [23] if a constant number of grid rows fit in memory. For general height graphs from other DEMs besides grids, we can use the I/O-efficient batched union-find structure [6] described in Section 3.1.2 to detect flat areas in $O(\text{sort}(N))$ I/Os.

Before we can assign connected component labels to flat areas, we must first identify all flat cells in a grid. This can be done in a simple scan over the grid while maintaining a 5×5 window of a grid cell and its neighbors in memory. If the vertex in the center of the window has no lower neighbors, we mark it as flat. Otherwise we check if it is adjacent to a cell of the same height with no lower neighbors.

Given a grid DEM in which all flat vertices have been marked, computing connected components can theoretically be done in scan(N) I/Os using an tiling approach by Arge et al. [23], but this algorithm is rather complicated to implement. We have implemented a simple scan(N) algorithm for computing connected component labels of each flat area in the case where a constant number of lines (rows or columns) of the grid fits in memory. Assume without loss of generality that a single row has a fewer cells than a single column. Otherwise we just rotate the grid. A row of the grid fits in memory when $\sqrt{N} \leq M$. This assumption seems reasonable because if M is $2^{22} \approx 4 \times 10^6$ and we need less than 64 bytes of information for each vertex in a given row, we can handle terrains up to

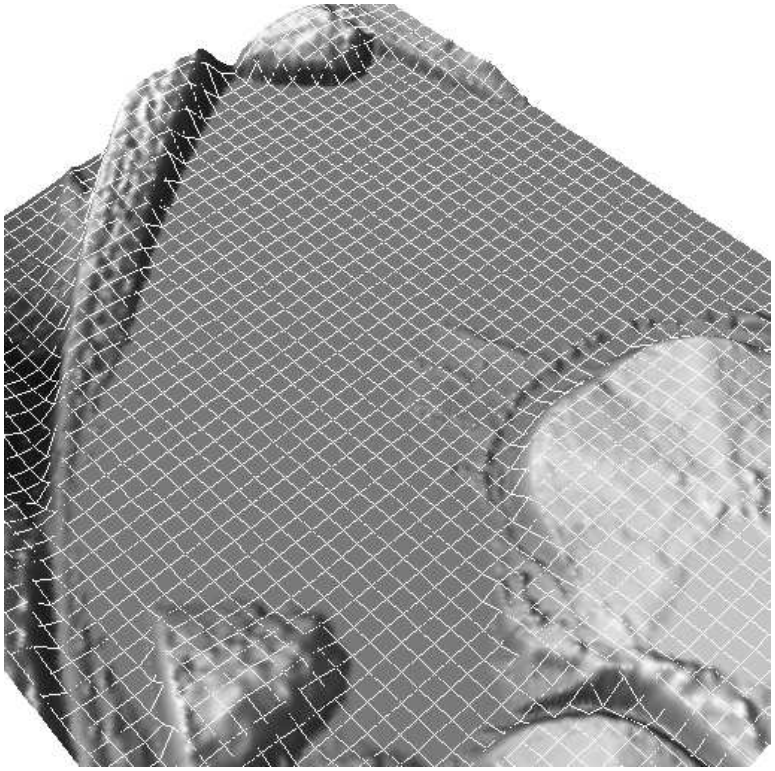


Figure 3.5: A flat plateau created by flooding.

2^{44} cells which occupy more than a 128 TB of space using only 256 MB of memory. Our algorithm consists of two sweeps over the the grid rows. In the first sweep, we sweep down over the rows and assign a temporary labels to connected components. After the down-sweep, a single connected component can have multiple labels, but we then sweep up the rows from bottom to top and assign a single label to all cells in a connected component. For both sweeps, we use an internal memory union-find data structure that maintains connected components of flat area labels. We discuss the details of our sweeps below.

Algorithm Description. In the down-sweep, we process rows top-down and keep two rows of the terrain in memory; the current row we are processing and the row immediately above it. For each grid vertex in memory we maintain the elevation of the cell, a flag to indicate if the cell is flat, and a possibly undefined connected component label. For each row r, we scan from left to right and for each flat cell u in the row we do the following. We create a new unique label $l(u)$ for u. Each label consists of an unique ID and the row number r at which this label was created. After defining a label for u, we check if u has more than one flat neighbor. If so, for each neighbor v where $l(v) \neq l(u)$, we union $l(u)$ and $l(v)$. After we scan the row from left to right and assign labels to each flat cell in the row, we scan through the row a second time and for each flat cell u we update the label of u by setting $l(u) = \textsc{Find}(l(u))$. For the algorithm to work correctly, we require that the representative of a component in the union-find data structure to be the label that was created first in the component and thus has the lowest row number. If two separate labels in a union-find component were created on the same row and unioned, the label with the lowest ID is the representative. We refer to the representative label of a union-find component as the *top-most* label of the component. The $\textsc{Find}(l(u))$ operation returns the representative of the union-find component containing $l(u)$. See Algorithm 1 for pseudo-code for the down-sweep.

Algorithm 1 Downsweep

```
1: for all rows r in top-down order do
2:    for all flat cells u in row do
3:       Create new unique label l(u)
4:       for all flat neighbors v with a label l(v) ≠ l(u) do
5:          UNION(l(u), l(v))
6:       end for
7:    end for
8:    for all flat cells u in row do
9:       l(v) = FIND(l(v))
10:   end for
11: end for
```

In the up-sweep, we process rows bottom-up and keep two rows of the terrain in memory; the current row we are processing and the row immediately below it. For each row r, we scan from left to right and for each flat cell u in the row we do the following. We check if u has any flat neighbor v in the row below u. If such a neighbor exists, and $l(u) \neq l(v)$, we union the two labels. After scanning this row once we scan through the row a second time and for each flat cell u we update the label of u by setting $l(u) = \textsc{Find}(l(u))$. See Algorithm 2 for pseudo-code for the up-sweep.

Correctness. To prove the correctness of this algorithm, we must show that after the up-sweep, all cells in the same flat area have the same unique label. To prove this, we first show that the down-sweep satisfies the following lemma.

Algorithm 2 Upsweep

```
1: for all rows r in bottom-up order do
2:    for all flat cells u in row do
3:       if u has a flat neighbor v in row below u then
4:          UNION(l(u), l(v))
5:       end if
6:    end for
7:    for all flat cells u in row do
8:       l(v) = FIND(l(v))
9:    end for
10: end for
```

Lemma 3 *After processing row r in the down-sweep, if any two cells u and v are on row r, are in the same flat area, and there exists a path from u to v completely contained in the flat area where each cell on the path is on or above row r, then u and v have the same label.*

Proof. First consider two adjacent cells u and v on the same row. If u and v are assigned different labels in the first scan of the row during the down-sweep, then the two labels will be unioned. After the second scan, the labels will be set to $\textsc{Find}(l(u)) = \textsc{Find}(l(v))$ because the labels were merged. Figure 3.6(a) shows this case on the left hand side. This case can easily be extended to the case of two cells u' and v' in the same flat area that are in contiguous block of cells on the sweep-line (see Figure 3.6(a)) to show that $l(u') = l(v')$. We call a contiguous block of cells on the sweep line in the same flat area an *anchor*.

We can prove the lemma by induction on the rows, using the anchors in the first row as the base case. Because there are no lines above the first row and each anchor on the first row has the same label as just described, we conclude that the lemma holds after processing the first row. Now assume that the lemma holds after processing row r and consider row $r+1$. Using the analysis above, we can show that each anchor on row $r+1$ will have a single label after processing row $r+1$. We next show that if two cells u and v are in two distinct anchors and are connected via a path of cells that are on or above the the sweep-line, then u and v will be assigned the same label. Since there exists a path from u to v, there must be two cells u' and v' along this path above the sweep-line and adjacent to a cell in the anchor of u and v respectively. Suppose the path from u' to v' is completely above the sweep-line as shown in Figure 3.6(b). We call such a path entirely above the sweep-line an *arc*. An arc is adjacent to two anchors on the sweep-line. By the inductive hypothesis, u' and v' have the same label l' because they are connected via a path on or above row r. Therefore when we scan row $r+1$, we will union the label l' with the label of some cell in both the anchors of u and v. When we scan the line again, u and v will get the same label $\textsc{Find}(l(u)) = \textsc{Find}(l(v))$.

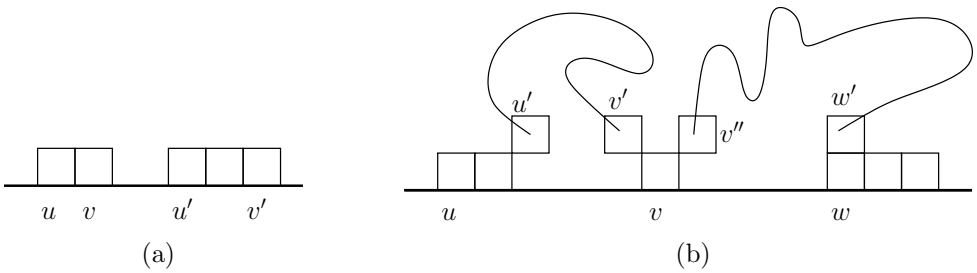


Figure 3.6: (a) Two anchors of contiguous cells on the sweep-line. (b) Anchors on the sweep line with arcs above sweep-line connecting anchors.

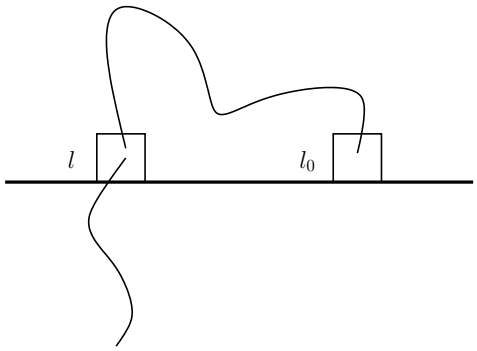


Figure 3.7: Proof of Lemma 4. The sweep-line shown is at the last occurrence of l_0 and some other label l in the same connected component is propagated below the sweep-line. l_0 is connected to l by a path on or above the sweep-line.

In the most general case, the path from a cell u to a cell w can pass through multiple arcs and anchors (see Figure 3.6(b)), but the first scan of row $r+1$ will merge all distinct labels in rows r and $r+1$ that are on the path from u to w into a single component through a series of unions equivalent to the repeated application of two cases above. On the second scan of row $r+1$, FIND($l(u)$) and FIND($l(w)$) will return the same label, and the proof is complete. □

To show that the up-sweep generates a single label for each flat area, we need to prove the following lemma regarding the down-sweep.

Lemma 4 *Consider all the labels in a flat area $\mathcal{C}$ after the down-sweep. All cells on the bottom-most row of $\mathcal{C}$ have the same label as the top-most label in $\mathcal{C}$.*

Proof. Assume this is not the case and that the top-most label l_0 in $\mathcal{C}$ is not the same as the label of cells in the bottom-most row r_b of $\mathcal{C}$. The labels of all cells in $\mathcal{C}$ in the bottom-most row must have the same label by the lemma on the down-sweep. Consider the bottom-most row $r < r_b$ in which the label l_0 appears in the flat area $\mathcal{C}$. Because l_0 does not appear on row $r+1$ we know there are no cells in $\mathcal{C}$ that are in row $r+1$ and adjacent to cells labeled l_0. Otherwise, these cells on row $r+1$ would have merged in the down-sweep with label l_0 and been replaced with the label l_0 during the second scan of row $r+1$ during the down-sweep. Furthermore, we know that some cell u with label $l \neq l_0$ on row r is in the same flat area as cells labeled l_0 and has a flat neighbor in row $r+1$. Otherwise, a label different from l_0 could not propagate below row r. Let v be a cell with label l_0 on row r. Since v and u are in the same flat area, and no path between v and u can extend below row $r+1$ without propagating label l_0 below row r, all cells on the path connecting u and v must be on or above row r. See Figure 3.7 for an illustration of this situation. But by the lemma above, u and v must have the same label and the label of both cells must be the representative in the connected component of the union-find, which is precisely l_0. Since $l(u) = l \neq l_0$ and $l(u) = l(v) = l_0$ is a contradiction, our earlier assumption that l_0, the top-most label in $\mathcal{C}$ is not the same as the label of cells in the bottom-most row r_b of $\mathcal{C}$ is wrong, and the proof is complete.

□

For the up-sweep we prove the following lemma.

Lemma 5 *After processing row r in the up-sweep, all cells in the same flat area that are on or below r have the same label. This label is the top-most label in the flat area.*

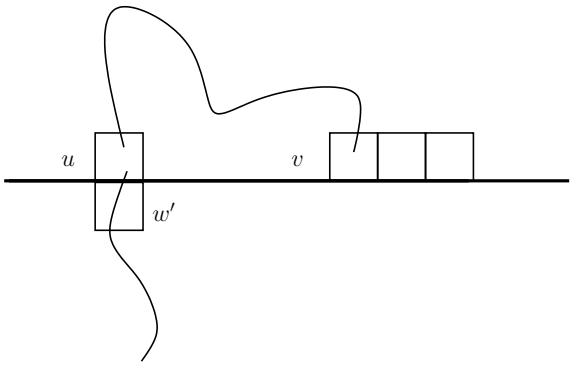


Figure 3.8: Two anchors on the sweep-line during up-sweep. The anchor containing u is connected to flat neighbor below the sweep-line. The anchor containing v is connected via a path on or above the sweep-line to the anchor containing u.

If we can prove the lemma holds for all rows, the correctness of the algorithm immediately follows because when we are finished with the sweep, the lemma states that all cells in the same connected component have the same label.

Proof. We proceed inductively. After processing the bottom row, all cells in the same flat area on the bottom row have the same label by the down-sweep lemma. Furthermore, the labels of any components on the bottom row are top-most label in their corresponding components by Lemma 4. Therefore, the lemma is true for the base case. Assume the lemma holds for row r. We want to show that after processing row $r-1$ above row r that the lemma still holds. Consider the set of anchors on row $r-1$ that are in the same flat area. If a cell u is in an anchor that has at least one cell w with a flat neighbor w' in row r, then all labels in cells in the anchor of u will be merged with the label of w'. Since $l(w')$ it the top-most label of the flat area containing u by the up-sweep lemma, all cells in the anchor of u will be assigned the label $l(w')$ in the second scan of row $r-1$ during the up-sweep. Thus in the case where u is in an anchor connected to a cell w' in row r, the lemma holds for row $r-1$.

If a cell v is in an anchor on row $r-1$ that does not have any cell adjacent to flat neighbor on row r then the anchor containing v is either in the bottom-most row of a newly encountered flat area in the up-sweep or there is a path of cells on or above the sweep-line connecting the anchor containing v to an anchor that has a cell with flat neighbors in row r. In the case that the anchor is in the bottom-most row of a flat area, the cells in this anchor have the minimum label of all cells in the component by Lemma 4 and the lemma holds. Otherwise, v in an anchor connected by a path of cells on or above the sweep-line to an anchor that has a cell u with flat neighbors in row r. See Figure 3.8 for an example. Because of the down-sweep lemma we know that u and v had the same labels after the down-sweep. Furthermore, since u merges its down-sweep label with the label of a vertex w' on row r, cells in both the anchor of u and the anchor of v which are in the same flat area as w' will have the same label as w' after the second scan of $r-1$. The label of w' is correct by the up-sweep lemma and thus the lemma holds for row $r-1$.

□

Analysis. We have shown that the algorithm correctly assigns the same unique label to all flat cells in the same connected component. We now show that the I/O-complexity of our algorithm is scan(N), where N is the number of grid cells. Identifying flat cells is done in one scan of the grid. Visiting each cell in the grid during the down-sweep and up-sweep is also done in scan(N) I/Os. The only remaining part of the algorithm to analyze is how the union-find structure is implemented.

In the description of the down-sweep algorithm, we say that a unique ID is generated for each new label. In the worst case, this could generate $O(N)$ unique ID labels. Since we only assume that a $O(1)$ rows of at most $\sqrt{N}$ cells, fit in memory, we cannot maintain a union-find structure on $O(N)$ unique labels in memory. Note however that during any point of the down-sweep or up-sweep, only two rows are in memory. The number of unique labels that can appear on these rows is at most $2\sqrt{N}$. We say that a label ID is active during the sweep if it appears in one of the two rows currently in memory. If we maintain a union-find structure only on these active labels, the structure will fit in memory. We can do this by building a new union-find structure when processing each row and deleting the structure after processing the row. In the up-sweep we initialize the union-find structure with all active labels. In the down-sweep, we may generate up to $\sqrt{N}$ new labels. We therefore initialize the union-find structure with all labels that are currently active and a set of $\sqrt{N}$ new labels that will be used if a new label is needed on the current sweep-line. Because the union-find structure on $2\sqrt{N}$ labels fits in memory, it does not require any I/Os. Therefore the total I/O-cost of the algorithm is scan(N).

3.4.2 Routing on a Single Flat Area

Once we have a unique flat area label for all vertices in each flat area, we sort all flat vertices in the height graph by their label so that vertices in each flat area are stored consecutively on disk. We can then route flow across each flat area by applying a flat routing model to each flat area.

In TerraFlow [14], flow routing on plateaus is performed using a *breadth-first* traversal of the area. The source vertices for the BFS are all spill-points of a given plateau. Since these vertices have a downslope neighbor, there exists an edge in the flow graph that routes flow from a spill point to a downslope neighbor. These spill-points are marked as visited in the BFS and the BFS algorithm visits other vertices in the flat area. We add an edge (u, v) to G if u has not been previously visited by the BFS but v has been visited and thus has a path towards the spill point. In this way, each cell on the flat area is routed towards the nearest (in the number of vertices) spill point.

Flat areas without spill points are different from plateaus in that incoming flow cannot leave. TerraFlow does not address this issue because TerraFlow removes all sinks in the terrain, regardless of importance. Because the persistence algorithm may preserve some sinks, our approach on sink flat areas is to detect vertices of the flat area that have at least one neighbor vertex with higher elevation. These vertices are then used as the initial sources in the BFS traversal. Direction are assigned as we visit vertices in the BFS, except that flow is routed away from previously visited vertices instead of towards them. In this way, we route flow towards the middle of the flat sink.

To analyze the complexity of this flow routing phase, note that if the vertices of a flat area fit in memory we can do the BFS routing internally, and create edges in E_r as outlined above. In our experience with high resolution floating point elevation data, each flat area is small and fits in memory. In the case where the flat areas are larger than memory, a $O(\text{sort}(N))$ algorithm for grids are described in [23]. The total I/O cost of flow routing on flat areas is $O(\text{sort}(N))$.

Alternative methods. While BFS is a commonly used method of routing flow on flat surfaces, it can create artificially looking parallel flow lines. See for example Figure 3.9(b) where the meanders of the river in the adjacent figure are gone after hydrologically conditioning the DEM using flooding. A number of parallel flow lines also appear in the conditioned terrain, especially in the lower right of the figure. Garbrecht and Martz developed an algorithm for flow routing on flat surfaces [49] that does two BFS traversals. The first BFS starts from all flat vertices that have a higher neighbor and computes the minimum distance $d^+(u)$ in length of the BFS path from a cell u to a cell with a higher neighbor. The second BFS starts from all flat vertices with a spill point and computes the minimum distance $d^-(u)$ in length of the BFS path from a cell u to a vertex with a spill point. The algorithm then computes $d(u) = d^-(u) - d^+(u)$ for all vertices in the flat area and creates and edge in the flow graph from a vertex u to a vertex v if $d(v) < d(u)$ where v has the minimum value of d

for all neighbors of u. Intuitively this method routes flow away from high areas and towards spill points and in practice generates fewer parallel streams. However, some vertex u may not have any neighbor v with $d(v) < d(u)$ and the approach must be iterated over remaining flat vertices with no outgoing edge.

The approach of Garbrecht and Martz was later improved by Soille [80] using the concept of geodesic distance functions. Soille's approach does not need to iterate over the flat area multiple times. Both of these methods tend to reduce the number of parallel flow lines and create more realistic looking networks. These methods could be used as replacements to the BFS approach described above in the case where the flat areas fit in memory, but it is not known if these algorithms can be made I/O-efficient.

In summary, we can construct the flow graph from G for grid DEMs including routing flow on flat areas using $O(\text{sort}(N))$ I/Os. In practice, we use the simpler $O(\text{scan}(N))$ algorithm described in the previous section for the common case in which a constant number of lines of the grid fits in memory, instead of the I/O-efficient but impractical $O(\text{scan}(N))$ algorithm for computing connected components [23].

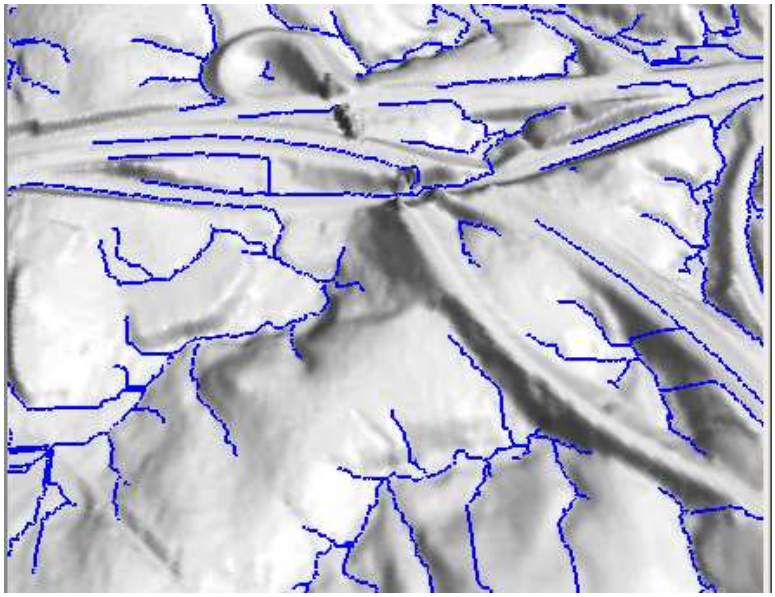
3.5 Flow Accumulation

Given a flow graph G computed as described above and an initial amount of flow for each vertex v in G, the flow accumulation algorithm is described as follows. For each vertex v in G, we compute the sum of the initial flow of v and flow along incoming edges (u_i, v) in G, and partition this sum across all outgoing edges (v, w_i). In the SFD model, all flow is distributed to the single outgoing edge. In the MFD model flow is distributed proportional to the vertical gradient along an edge, where edges with a steeper gradient receive more flow.

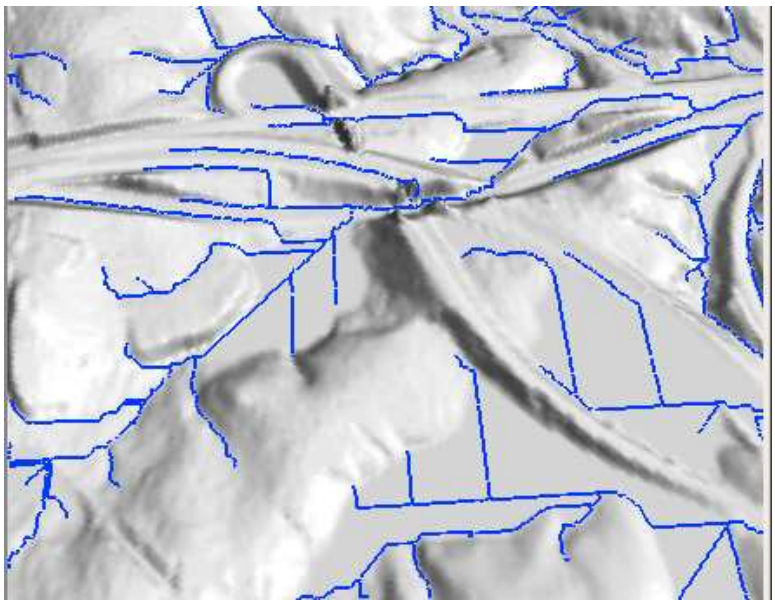
In the case of SFD or MFD flow routing, the flow graph is a tree or a DAG, respectively. Temporarily ignoring flat areas, the elevation of the source vertex in a flow graph edges defines a topological order of G. In the case of flat areas, auxiliary information must be used to topologically order the flat edges of G that are the same height. Because we assume the flat areas fit in memory, any topological sort algorithm can be used to create this auxiliary information. Arge et al. [23] describe an I/O-efficient algorithm that processes the vertices of the flow graph in topological order to compute the flow accumulation of each vertex in a flow graph. Their method sweeps a plane top-down over the terrain and process each vertex as it appears in the topological order. At each vertex u, the input flow from each incoming edge to u is known by extracting data from a priority queue, and flow for edges leaving u can be computed and pushed along the edges of a DAG. The flow accumulation for a flow graph with N vertices can be computed in $O(\text{sort}(N))$ I/Os. As a post processing step, we can extract edges (u, v) in the flow graph for which the flow accumulation of u and v exceed a user-specified threshold. These edges form a river network that can be used for various additional studies.

3.6 Issues on Modeling High Resolution Terrains

The most common method of sink removal used in most GIS software is the method described above that floods the terrain. This method is used in both TerraFlow and the new persistence sink removal method described in this chapter. However, flooding is not an ideal sink removal in certain situations. Consider the terrain is shown in Figure 3.9(a) that contains a number of bridges. Because the actual terrain is modeled as a surface we cannot assign multiple elevations to a single height graph vertex. Thus these bridges act as barriers that impede water flow when in reality, water flows under the bridge along the surface of the river. If we we consider the persistence of sinks in the terrain, bridges create sinks upstream of the bridges with high persistence values. If a user



(a)



(b)

Figure 3.9: (a) Terrain and flow graph edges shown in blue with flooding of only low persistence sinks (b) Terrain and flow graph edges with flooding of all sinks.

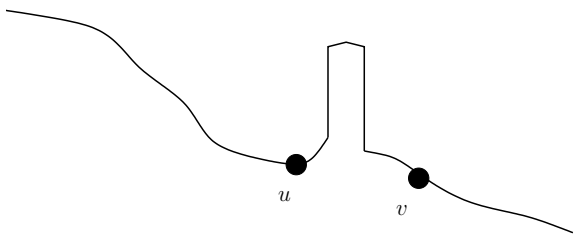


Figure 3.10: A sink u blocked by a bridge. u is close to the sharp upstream edge of the bridge. A vertex v close to the downstream edge of the bridge is lower than u.

specifies a small persistence threshold for removing sinks and then computes the flow graph G, the graph becomes disconnected at the bridges. Removing sinks with a high persistence threshold using flooding will raise the heights of terrain in the sink to the lowest height on or near the bridge. An example of such extensive flooding of the terrain shown in Figure 3.9(b). The flow graph is more connected and has fewer components, but the terrain has been significantly modified, especially in the lower right corner where river valleys have been flooded extensively to flow over the bridge. Furthermore, since the flooded sinks create large flat areas, the breadth-first search flat routing method as described above creates a number of artificially looking parallel streams and deviates from the flow graph edges in the original terrain which has much more meandering paths.

Because digital elevation models derived from modern remote sensing methods frequently resolve hi-resolution features such as bridges, we would like to modify DEMs in such a way that water can flow across bridge features but we do not significantly change the elevations in the DEM. We have already seen that flooding of sinks created behind bridges is not an ideal solution as flooding results in unrealistic flow graphs. An interesting open problem is how to accurately and efficiently identify sinks blocked by bridges bridges and how to make minimal modifications to the terrain so that the connectivity of flow graph is similar to the connectivity of river in the real terrain. This problem can be divided into two sub-problems; identifying sinks that are created by a bridge blocking a downslope path, and modifying the terrain to allow water to flow through the bridge. We consider some methods below that may lead to improved methods of detecting minima blocked by bridges and modifying the terrain to allow flow routing across the bridge. While we did not implement these algorithms in this thesis, we consider them potential directions for future work.

3.6.1 Detecting Sinks Blocked by Bridges

Persistence can provide an initial hint as to which sinks are blocked by bridges as such sinks typically have a high persistence value. However, persistence alone cannot distinguish between a sinks that is blocked by a bridge and a quarry or other natural sink such as the one depicted in Figure 3.4. Flow should not be routed from inside a quarry to the outside, but flow should be routed across the bridge. Two sources of auxiliary information may be able to help detect sinks blocked by bridges and distinguish these sinks from significant sinks with high persistence that should not be removed.

For the first source of auxiliary information, we observe that water in the actual terrain will flow downhill and the lowest point on a river upstream of a bridge will be near that bridge. Also, the elevation of terrain along the river downstream of bridge will be lower than elevations along the river upstream of the bridge. Thus sinks blocked by bridges in the DEM will typically be located close to an edge of the bridge. Furthermore, because these DEMs are of high resolution, there will be sharp edges in the terrain on the edges of the bridge. By extracting a small neighborhood of the terrain from around each sinks with high persistence, we can look for these features described above and illustrated in Figure 3.10 using standard image processing techniques to help identify sinks blocked by bridges.

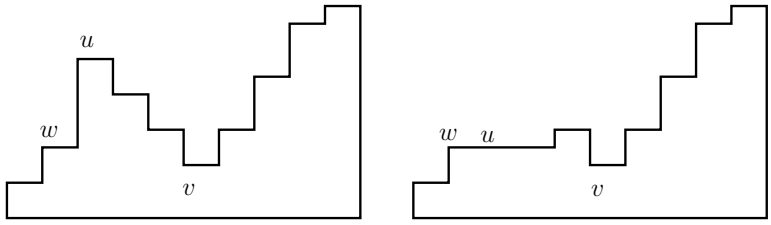


Figure 3.11: The Martz and Garbrecht approach lowers the spill point u of a sink represented by v by lowering vertices along a sub-path from u to v the lowest height vertex w adjacent to u and not in the watershed of v. The breaching length, or maximum length of the sub-path is two in this example. Original terrain is shown on the left and the modified terrain is shown on the right.

A second source of auxiliary information that can help detect sinks blocked by bridges is the flow direction and flow accumulation derived from DEM that has had noise below a small persistence threshold τ_0 removed. Remaining sinks in this DEM will have a persistence above τ_0. We run the flow routing and flow accumulation on this terrain. Then, in addition to looking at the elevation in the neighborhood of sinks with high persistence, we could also consider the flow accumulation and flow directions in this neighborhood. While the true flow graph will still be disconnected by the bridges, partial information about the general direction and amount of flow in the area around bridges. can provide hints that could indicate the existence of bridges. A river that is blocked by a bridge is likely to have a path of flow directions with high flow accumulations flowing perpendicular to the upstream edge of a bridge. A similar path flowing perpendicularly away from the bridge on a downstream edge is a good indication that the flow graph should be connected across the bridge.

3.6.2 Removing Minima Blocked by Bridges

By combining persistence and auxiliary information, we may be able to identify those sinks which are blocked by bridges. Next, we must determine a way to modify the terrain that removes the sinks and allows the flow modeling algorithms to route flow through the bridge. As we have seen in Figure 3.9(b), flooding provides one possible solution, but makes extensive modifications to the terrain. One possible alternative is to modify the terrain by lowering the elevation across vertices on the bridge instead of raising the elevation of the vertices in the sink upstream of the bridge. We describe two approaches described in the GIS literature.

Martz and Garbrecht [64] describe an algorithm that partitions a height graph into *watersheds* where each watershed is represented by a local minimum. A vertex u is the watershed of a sink v if there is a path of non-decreasing height from u to v. A vertex u is on the boundary of watersheds v and w if u is in both watersheds. For each watershed v, the algorithm finds the vertex u on the boundary of v that has the lowest elevation. Since u is on the boundary of two or more watersheds, there must be a vertex w that is not in the watershed of v and is adjacent to u. If multiple such vertices exist, w is the lowest such vertex. The algorithm of Garbrecht and Martz lowers the elevation of vertices along a sub-path of a path from u to v to the height of w. The sub-path is a path that starts at u and travels along the path of steepest descent towards v and stops when either the path reaches a vertex with height less than the height of w, or when the path reaches a user-specified breaching length. Figure 3.11 shows the result of lowering the elevations in the watershed of v when the breaching length is two. Note that the algorithm may not remove all sinks, but it can significantly reduces the topological persistence of v and will reduce the amount of flooding needed to remove the sink containing v.

Soille developed an algorithm [81] that uses a morphological approach called *carving* to modify the terrain so that no flooding of sinks is needed. The algorithm marks a sub-set of sinks as being real, and all other sinks as being spurious. The algorithm then does a bottom-up sweep

while maintaining the components of all sinks. When a component of a significant sinks merges with a spurious component, the algorithm carves a path from the sinks of the spurious component towards the sink of the real component. The height of vertices along this path is the minimum of the vertices' original height or the height of the sink in the spurious component. The merged component is considered a component of the significant sink. In a related paper, Soille presents a sink removal method [80] that computes a cost for partially carving a sink and partially flooding a sink and minimizes the cost of terrain modifications for each sink. While methods similar to the ones above could possibly be suitable for modifying terrain near sinks blocked by bridges, all of these alternatives to flooding were developed in the RAM model of computation. Similar ideas could potentially be modified to develop I/O-efficient alternatives to flooding for sink removal on grid DEMs.

For modern hi-resolution grid DEMs, it is clear that flooding all sinks regardless of importance can produce unrealistic looking terrains and can significantly alter the original terrain. In this Chapter we described how persistence can be used to decide which sinks to remove and how it may be possible to use persistence to identify sinks blocked by bridges. By developing improved I/O-efficient methods for noise removal, one could create much more realistic terrain models that accurately model the true course of water over the terrain.

Chapter 4

Watershed Decomposition

4.1 Introduction

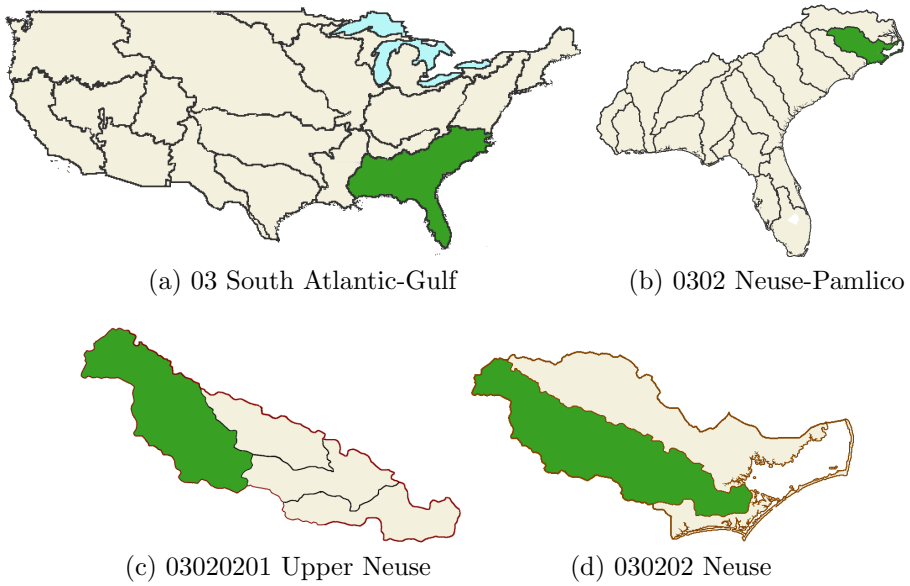
Over millions of years, rainfall has been slowly etching networks of rivers into the terrain. Today, studying these river networks is important for managing drinking water supplies, tracking pollutants, creating flood maps, and more. Hydrologists can use large-scale digital elevation models of the terrain along with a Geographic Information System to automate much of such studies. Often it is not necessary to study the entire terrain or river network at once; frequently one is only interested in regions that are downstream of a particular river, or the upstream areas that contribute flow to a particular river. By decomposing the terrain into a set of disjoint *hydrologic units*—regions where all water within the region flows towards a single, common outlet—one can quickly identify areas of interest without having to examine the entire terrain. The Pfafstetter labeling scheme described by Verdin and Verdin [90] defines a hierarchical decomposition of a terrain into arbitrarily small hydrological units, each with a unique label. These *Pfafstetter labels* also encode topological properties such as upstream and downstream neighbors, making it possible to automatically identify hydrological units of interest based on the Pfafstetter label alone.

In this chapter, we describe an efficient algorithm [15] for computing Pfafstetter labels efficiently on grid DEMs. Our algorithm is capable of handling massive high-resolution DEMs that are too large to fit in main memory of even a high-end machine. With the recent progress in remote sensing technology, such as lidar, such DEMs are increasingly becoming available. A method of constructing such DEMs from lidar point clouds was presented in Chapter 2. Existing methods for determining hydrological units on grid DEMs use either manual methods [77], local topological filters [73, 59] or complete modelling of water flow over a terrain [72] to identify terrain features and extract watersheds. While the manual methods are often very ad-hoc, some of the main disadvantages of the current automatic methods is that they do not naturally define a hierarchical decomposition or a hierarchy that encodes topological properties such as upstream and downstream neighbors. Furthermore, the existing algorithms cannot handle massive grid DEMs.

4.1.1 USGS Hydrologic Unit System

An example of a frequently used hydrological unit terrain decomposition is the Hydrologic Unit System developed by the Water Resources Division of the United States Geological Survey (USGS) [77]. The Hydrologic Unit System is a hierarchical decomposition of the terrain in the United States. At the top level, the US is divided into 21 regions which are further divided into 222 sub-regions. Each sub-region is completely contained within exactly one larger region. Sub-regions are further divided into basins, sub-basins, watersheds and sub-watersheds, offering a total of six levels of decomposition. The USGS assigns a hydrologic unit code (HUC) to each hydrologic unit. A HUC is a two to twelve digit code where each pair of successive digits indicate the region, sub-region, basin, sub-basin, watershed, and sub-watershed IDs, respectively. For example, HUC 03020201 corresponds to sub-basin 01 (Upper Neuse River) in basin 02 (Neuse River) of sub-region 02 (Neuse-Pamlico) in Region 03 (South Atlantic-Gulf). Refer to Figure 4.1. Maps with HUC labels at the sub-basin, or eight-digit level, are currently available and ten to twelve digit HUC maps are in development.

While the USGS Hydrologic Unit System provides a hierarchical decomposition of the terrain in the United States, it has some disadvantages. First, while the HUC boundaries are available for download, there is no automatic way to compute the USGS hydrological units given a digital



(a) 03 South Atlantic-Gulf (b) 0302 Neuse-Pamlico

(c) 03020201 Upper Neuse (d) 030202 Neuse

Figure 4.1: A region, sub-region, basin and sub-basin in the USGS Hydrologic Unit System.

elevation model. As the quality and resolution of digital elevation models improve, the published HUC boundaries may not exactly match the boundaries suggested by the data. Second, HUCs at the sub-basin level may be too large for a particular application. Further sub-levels are in development but are not complete at this time. Third, HUCs are only available for the United States. Other countries and organizations have other coding methods [90]. Finally, the digits chosen for a particular HUC are, for the most part, arbitrary. Given two HUCs, it is often difficult or impossible to determine if water from one HUC flows into the other based on their numbering alone. Because finding the hydrological units upstream and downstream from a given location is a common task, a numbering scheme that allows a computer or user to relate hydrological units, without the need for visual inspection, would be helpful.

4.1.2 Introduction to Pfafstetter labels

The Pfafstetter labeling method described by Verdin and Verdin [90] addresses several disadvantages of the USGS Hydrologic Unit System. As mentioned earlier, the method can automatically divide a terrain into a hierarchy of arbitrarily small hydrological units, each with a unique label. Furthermore, Pfafstetter labels encode the basic topological connectivity of the hydrological units, allowing users to determine if one basin is upstream or downstream of another by examining the labels.

We present a conceptual definition of Pfafstetter labels here and will give a more formal definition in the context of grid DEMs in Section 4.2. Before defining Pfafstetter labels, we define a *river* $\mathcal{R}$ to be a directed path of monotonically non-increasing height over a terrain. The highest and lowest points on the river are the *source* and *mouth* (or *outlet*), respectively. The *basin* of $\mathcal{R}$ consists of the contiguous area of the terrain whose water flows, or *drains*, into the given river at a point between the source and mouth. All water in $\mathcal{R}$ eventually flows through the outlet. Within a basin corresponding to a river $\mathcal{R}$, other rivers exist that flow into $\mathcal{R}$. These rivers are called *tributaries* of $\mathcal{R}$, and the *confluence* of a river $\mathcal{R}$ and a tributary is the mouth of the tributary of $\mathcal{R}$, that is, the point where the tributary joins $\mathcal{R}$. Each tributary has a corresponding basin that is a sub-region of the basin of $\mathcal{R}$.

Given a river $\mathcal{R}$ along with its corresponding basin and tributaries, the Pfafstetter method [90]

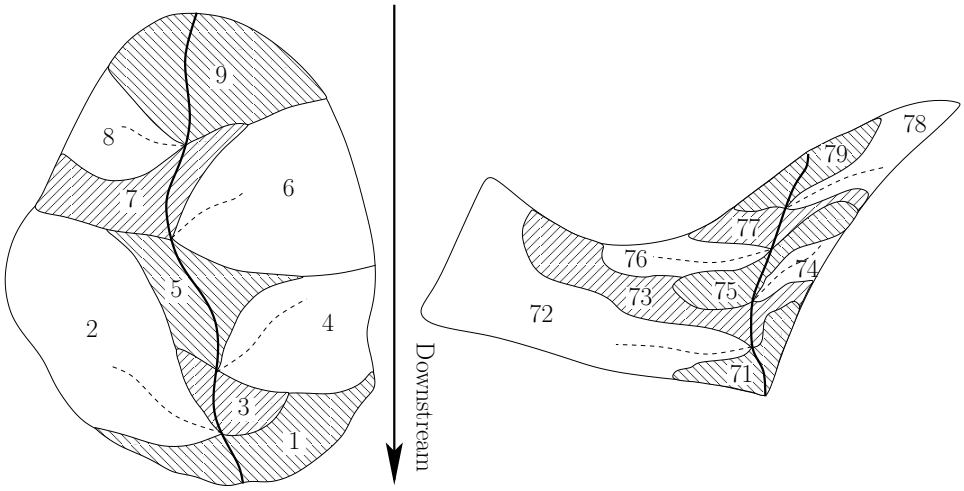


Figure 4.2: A division of a river basin into nine basin/interbasins and recursive subdivision of interbasin 7.

divides the basin into nine disjoint hydrological units: four *basins* and five *interbasins*. As we trace the river $\mathcal{R}$ upstream from the mouth to the source, we encounter multiple confluences with tributaries. Of all the tributaries that flow into $\mathcal{R}$, we identify the four that have the largest tributary basin area and assign the basin labels 2, 4, 6, and 8 to these basins, in the order in which we encounter the tributaries when moving upstream along $\mathcal{R}$. Thus, the furthest downstream tributary basin is assigned the label 2, and the furthest upstream tributary basin is assigned the label 8. Refer to Figure 4.2. The locations of the mouths of these four largest tributary basins divide the main river $\mathcal{R}$ into five distinct segments: the segment between the mouth of $\mathcal{R}$ and the confluence of the first tributary basin, the segments between the first and second, second and third, and third and fourth tributary basins, and the segment above the fourth tributary basin up to the source of $\mathcal{R}$. Regions in the basin of $\mathcal{R}$ that do not drain into one of the four largest tributary basins must drain into one of these five segments along $\mathcal{R}$. Starting from the mouth and proceeding upstream, these regions are assigned the interbasin labels 1, 3, 5, 7, and 9, respectively. In the case where we have $k < 4$ tributaries, we divide the basin into $2k+1$ pieces labeled 1 through $2k+1$, and do not assign labels $2k+2$ through 9.

To get a hierarchy of hydrological units, we apply the above definition recursively to each of the four largest tributaries with their corresponding basins, and to each of the five segments of $\mathcal{R}$ with their corresponding interbasins. In the interbasin case, we reuse the portion of main river that intersects the interbasin and was computed in the previous level of the recursion. The recursive labels of the subdivided regions are appended to the existing label of the original region. Thus, subdivisions further down the hierarchy have longer labels. Refer to Figure 4.2.

4.1.3 Our results

In this chapter we present an I/O-efficient algorithm for computing the Pfafstetter label of each cell of a grid DEM in $O(\text{sort}(T))$ I/Os, where T is the total length of the cell labels. To our knowledge, our algorithm is the first efficient algorithm for the problem. If each Pfafstetter label consist of a constant number of digits, e.g. if we truncate them, our algorithm only uses $O(\text{sort}(N))$ I/Os, where N is the number of grid cells. If the DEM and the labels fit in main memory, the algorithm uses $O(T)$ time. The overall algorithm, as well as a formal definition of Pfafstetter labels in grid DEMs, is given in Section 4.2; details are then given in Section 4.3 and Section 4.4. We have implemented our algorithm, and in Section 4.5 we present the results of a preliminary experimental study using

massive real life terrain data that shows that our algorithm is practically as well as theoretically efficient.

4.2 Pfafstetter labeling of grid DEM

In Section 4.1.2 we discussed the conceptual hydrological definition of Pfafstetter labeling [90], which is independent of the actual terrain representation (DEM format). In this section we define Pfafstetter labels on grid DEMs more formally in terms of the so-called *flow tree*, which can be obtained from a DEM in $O(\text{sort}(N))$ I/Os using existing software tools (algorithms). We then discuss the overall idea in our algorithm for computing the Pfafstetter labels of a given flow tree.

4.2.1 Pfafstetter labeling of flow tree

Several different methods for modeling water flow on grid DEMs have been proposed; refer to [59, 46, 72, 82] for a discussion of the different methods. To model the direction water naturally flows from each cell s in the grid, most of these methods assign one or more *flow directions* from s to one or more of its (at most) eight neighboring cells. In the most common method [59], each cell s is assigned a single flow direction to the lowest of the lower neighboring cells. To model water flow off the terrain, we first identify cells on the boundary of the terrain without any lower neighbors. We then assign the flow direction of these cells to an imaginary cell ρ outside the terrain. We refer to ρ as the global *outside sink*. The cells and flow directions naturally form a graph with a directed edge from cell s to cell t if s is assigned a flow direction to t. Assuming that the DEM does not contain any cells without lower neighbors other than the boundary cells, this graph is indeed a tree $\mathcal{T}$ since it contains $N-1$ edges (each cell except ρ has one downslope edge to a neighbor cell) and does not have cycles (flow directions go to lower cells). If we root $\mathcal{T}$ at ρ, each cell s is connected to ρ through a unique path of cells $s = s_1, s_2, s_3, \ldots, s_k = \rho$, where cell s_i is assigned a flow direction to s_{i+1}. Thus water can flow from s to outside sink ρ through $s_2, s_3, \ldots s_{k-1}$. Water from cells in the subtree rooted in s drains through s on its way to (the outside) ρ. In the notation of Section 4.1.2, such a path in $\mathcal{T}$ corresponds to a *river* $\mathcal{R}$ with *mouth* ρ (and *source* s). If the DEM *does* contain cells without lower neighbors other than the boundary cells, assigning flow directions as above to cells with a lower neighbor leads to a *forest* of trees where water in each tree can flow from a cell through parent cells to the root of a tree [14].

We define Pfafstetter labels of a DEM in terms of a forest of trees. For simplicity, we temporarily consider a single binary *flow tree* $\mathcal{T}$ with root ρ. In Section 4.4 we discuss how our algorithms can easily be extended to the general case of a forest of not necessarily binary trees. Furthermore, we assume that each leaf l in $\mathcal{T}$ is augmented with a *drainage area* $d(l) \geq 1$, and that each internal node v in $\mathcal{T}$ is augmented with a drainage area $d(v)$ that is one plus the sum of the drainage areas of v's children. Note that if $d(l) = 1$ for every leaf l, then $d(v)$ is the size of the subtree rooted in v. In section Section 4.5 we discuss how flow directions can also be assigned to (some) cells without lower neighbors to obtain a flow tree/forest that yield practically realistic watershed hierarchies (Pfafstetter labels).

Pfafstetter labels of a binary flow tree $\mathcal{T}$ augmented with drainage areas are defined as follows. Let the *main river* $\mathcal{R}$ of $\mathcal{T}$ be the root-leaf path obtained by starting at the root ρ of $\mathcal{T}$ and in each node continuing to the child with the largest drainage area. The subtrees obtained if $\mathcal{R}$ is removed from $\mathcal{T}$ are called *tributary basins* and the root of one of these subtrees a *tributary mouth*. First consider the case where at least four tributary mouths are obtained if $\mathcal{R}$ is removed. In this case, let v_2, v_4, v_6, v_8 be the four tributary mouths with largest drainage area, numbered in the order they are met when traversing $\mathcal{R}$ from ρ towards a leaf. Let p_i and s_i denote the parent and the sibling of v_i, respectively; both p_i and s_i are on $\mathcal{R}$. If we remove the eight edges incident to p_2, p_4, p_6 and p_8 (i.e. edges (v_i, p_i) and (s_i, p_i), for $i \in \{2, 4, 6, 8\}$), $\mathcal{T}$ is decomposed into four tributary basins

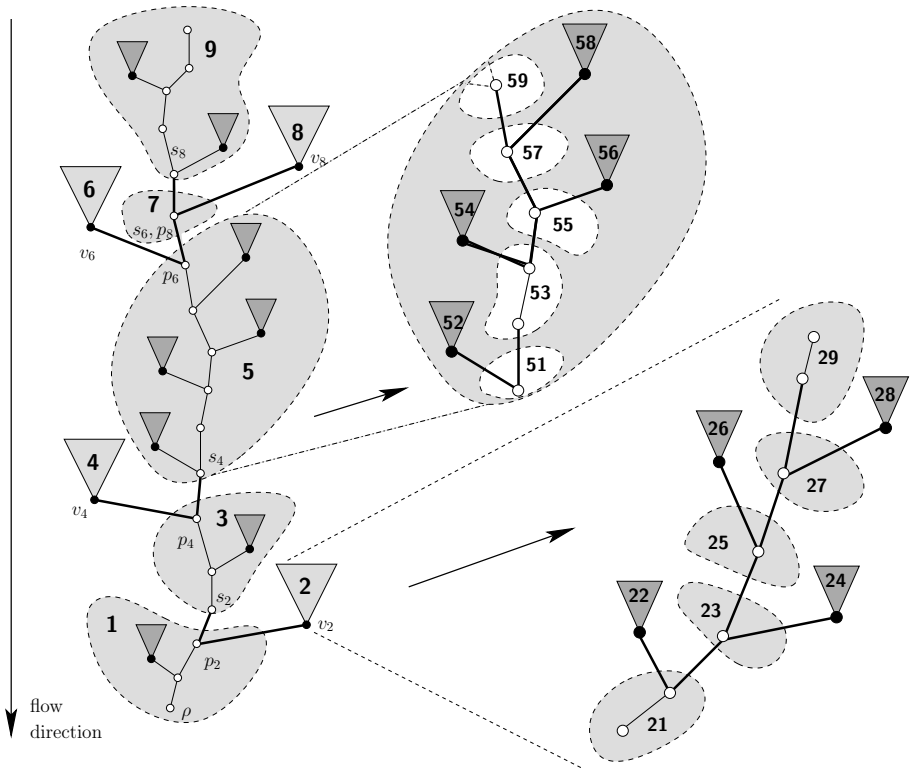


Figure 4.3: *Left figure:* A flow tree $\mathcal{T}$ with the main river shown as white circles and tributary mouths as black circles (circle nodes constitute an augmented river). Removing the eight bold edges decomposes $\mathcal{T}$ into four tributary basins and five interbasins, each with the first digit in their Pfafstetter label shown in bold type. The remaining digits in the Pfafstetter label of the nodes in each basin (subtree) are computed recursively. *Two right figures:* First level of recursion for interbasin labeled 5 and tributary basin labeled 2.

rooted in v_2, v_4, v_6, and v_8, as well as five *interbasins* rooted at $s_0 = \rho$, s_2, s_4, s_6 and s_8. The Pfafstetter label of a node in the tributary basin rooted in v_i is i followed by the label obtained by recursively labeling the basin. The label of the nodes in the interbasin rooted in s_i (which includes nodes on $\mathcal{R}$) is $i+1$ followed by the label obtained by recursively labeling the basin. In the case where $1 \leq k < 4$ tributary mouths are obtained when $\mathcal{R}$ is removed from $\mathcal{T}$, labels 1 through $2k+1$ are assigned as above, while labels $2k+2$ through 9 are not assigned. Finally, no labels are assigned when no tributary mouths are obtained, that is, when all nodes of $\mathcal{T}$ are on $\mathcal{R}$. Refer to Figure 4.3.

4.2.2 Computing Pfafstetter labels of flow tree

The recursive definition of Pfafstetter labels of a binary flow tree $\mathcal{T}$ naturally leads to a recursive algorithm to compute the labels: Compute the main river $\mathcal{R}$ and four largest tributary mouths, break the tree into nine subtrees, and recurse. Unfortunately, because it is difficult to predict the node access pattern and layout the tree efficiently in memory so that nodes are accessed sequentially, it seems hard to make such a direct algorithm I/O-efficient (or time-efficient). Instead our algorithm works by decomposing $\mathcal{T}$ into a set of rivers (augmented with tributary mouths), Pfafstetter labeling them individually, and finally combining the labels of the individual augmented rivers to obtain the Pfafstetter labels for all nodes of $\mathcal{T}$.

Our decomposition of the flow tree $\mathcal{T}$ into augmented rivers is defined by a *tributary tree* $\mathcal{T}^t$,

where each node l in $\mathcal{T}^t$ stores an augmented river $\mathcal{R}_l^t$ and where m is a child of l if and only if the parent of the mouth of $\mathcal{R}_m^t$ is on $\mathcal{R}_l^t$, that is, if $\mathcal{R}_m^t$ flows directly into $\mathcal{R}_l^t$. More precisely, the root r of $\mathcal{T}^t$ contains the path obtained by starting at the root ρ of $\mathcal{T}$ and in each node continue to the child with the largest drainage area; for each node v on the path we also include the (possible) child of v not on the path (called a *tributary mouth node*) in $\mathcal{R}_l^t$. Note that $\mathcal{R}_r^t$ is the main river $\mathcal{R}$ in the above definition of Pfafstetter labels of the flow tree $\mathcal{T}$ augmented with its tributary mouths. The root r has a child for each tributary basin of $\mathcal{R}$, that is, for each subtree of $\mathcal{T}$ obtained if $\mathcal{R}$ is removed from $\mathcal{T}$; the rivers in these children are obtained recursively. Note that this means that each tributary mouth is stored exactly twice, namely in $\mathcal{R}_r^t$ and as the mouth of the main river $\mathcal{R}_l^t$ in a child l of r. Refer to Figure 4.4.

Given a Pfafstetter labeling of each individual augmented river $\mathcal{R}_l^t$ in the tributary tree $\mathcal{T}^t$, we can combine these labels to obtain the Pfafstetter labeling of the whole flow tree $\mathcal{T}$ as follows. Consider the augmented river $\mathcal{R}_r^t$ stored in the root of r. As mentioned, $\mathcal{R}_r^t$ is the main river $\mathcal{R}$ in the definition of Pfafstetter labels of $\mathcal{T}$, augmented with its tributary mouths. Since in the definition of Pfafstetter labels of $\mathcal{T}$, the labeling of $\mathcal{R}$ only depends on the drainage area of its tributary mouths (first digit is determined by the four tributary mouths with largest drainage areas, and the rest recursively determined in each interbasin), the labels of the common nodes in main river $\mathcal{R}$ and the individually labeled augmented river $\mathcal{R}_r^t$ are indeed the same. Furthermore, the labels of the nodes in a tributary basin of $\mathcal{R}$ consists of some prefix determined by the labeling of the nodes on $\mathcal{R}$ (a digit for each recursive labeling step where the tributary basin is part of one of the four interbasins, followed by a digit determined in the recursive call where the tributary mouth has one of the four largest drainage areas), followed by the label obtained by recursively labeling the basin. The prefix is exactly the label assigned to the mouth of the tributary basin in the augmented river $\mathcal{R}_r^t$. Thus we can obtain the Pfafstetter labels for all nodes in $\mathcal{T}$ from a labeling of the augmented rivers in $\mathcal{T}^t$, simply by assigning the nodes in the main river $\mathcal{R}$ the labels of the corresponding nodes in $\mathcal{R}_r^t$ in the root r of $\mathcal{T}^t$, and recursively label the nodes in each subtree of r while prefixing the labels in the subtree rooted in child l with the label of the tributary mouth node in $\mathcal{R}_r^t$ corresponding to the mouth of the main river $\mathcal{R}_l^t$.

Intuitively, computing the tributary tree $\mathcal{T}^t$ from the flow tree $\mathcal{T}$ is easier than computing Pfafstetter labels directly on $\mathcal{T}$. The definition of $\mathcal{T}^t$ suggest a natural algorithm based on a DFS-traversal of $\mathcal{T}$, where in each step the child with largest drainage area is chosen. By modifying the known $O(\text{sort}(N))$ I/O algorithm for DFS-numbering nodes in a tree [34], it seems possible to obtain a $O(\text{sort}(N))$ I/O algorithm for our special DFS-traversal problem. However, while the know general DFS-numbering algorithm is quite complicated (and therefore not of practical interest), the special structure of flow trees allows us to develop a simple and practical $O(\text{sort}(N))$ I/O algorithm. We describe this algorithm in Section 4.4. Similarly, once each individual augmented river in the tributary tree $\mathcal{T}^t$ has been labeled, an algorithm based on DFS-traversal (or a BFS-traversal) can be used to combine the labels from the augmented rivers to obtain the Pfafstetter labels of $\mathcal{T}$ in $O(\text{scan}(T))$ I/Os, where T is the total length of the labels. We also describe such a simple and practical algorithm in Section 4.4. The remaining part of our algorithm, a $O(\text{scan}(T))$ I/O algorithm

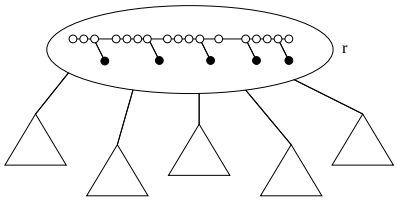


Figure 4.4: The root r of the tributary tree $\mathcal{T}^t$ and five subtrees. The augmented river $\mathcal{R}_r^t$ is stored in the root, and for each tributary mouth node in $\mathcal{R}_r^t$, there is one subtree of r.

for computing Pfafstetter labels on a single augmented river, is described in Section 4.3.

4.3 Labeling single river

In this section we describe a simple and I/O-efficient algorithm for computing the Pfafstetter labels of a single augmented river $\mathcal{R}_l^t$, that is, a simple flow tree consisting of one path (river) where each node possibly has a tributary mouth node child. Our algorithm is described in Section 4.3.3; in Section 4.3.1 and Section 4.3.2 we first discuss a data structure, the Cartesian tree, used in the algorithm.

4.3.1 Cartesian Tree

Let $A = (a_1, a_2, \ldots, a_N)$ be a sequence of N elements, each with an associated weight, and let A_i denote the prefix $(a_1, a_2, \ldots, a_i)$ of A. The Cartesian tree $\mathcal{C}(A)$ of A is a binary tree defined as follows [92]: If A is empty, $\mathcal{C}(A)$ is empty. Otherwise, let a_i be the element with the largest weight in A; if there is more than one occurrence of the largest weight, a_i is the element that appears first in A. $\mathcal{C}(A)$ consists of a root v containing an element with weight $a(v) = a_i$, with a left subtree $\mathcal{C}((a_1, ..., a_{i-1}))$ (a Cartesian tree on the elements before a_i in A) and a right subtree $\mathcal{C}((a_{i+1}, ..., a_N))$ (a Cartesian tree on the elements after a_i in A). Note that the weights of elements on a root-leaf path in $\mathcal{C}(A)$ are nondecreasing.

The Cartesian tree $\mathcal{C}(A)$ of a sequence A can be constructed in $O(N)$ time using an algorithm that iteratively constructs $\mathcal{C}(A_i)$ from $\mathcal{C}(A_{i-1})$ as follows [92]: Let the rightmost path P of $\mathcal{C}(A_{i-1})$ be the path traversed by starting at the root r and repeatedly continue to the right child until a node l without a right child is reached; note that this is not necessarily the path from the root to the rightmost leaf of $\mathcal{C}(A_{i-1})$. We construct $\mathcal{C}(A_i)$ by first traversing P from l towards r, until two adjacent nodes u and v such that $a(u) \geq a_i > a(v)$ are located; if $a(l) \geq a_i$, $u = l$ and v is non-existing, and if $a(r) < a_i$, $v = r$ and u is non-existing. We then construct a new node w containing an element with weight $a(w) = a_i$, and make w the right child of u and v the left child of w. Refer to Figure 4.5. The correctness of the algorithm follows from the fact that the weights of the elements along P are non-decreasing and that w is inserted as a right child without a left child; Refer to [92, 48, 30]. The linear time bound follows from the fact that all nodes on P traversed to find u and v (except u) are removed from P by the insertion of w (that is, they are not on the rightmost path of $\mathcal{C}(A_i)$) and therefore they are not traversed in later iterations; thus we traverse $O(N)$ nodes in total.

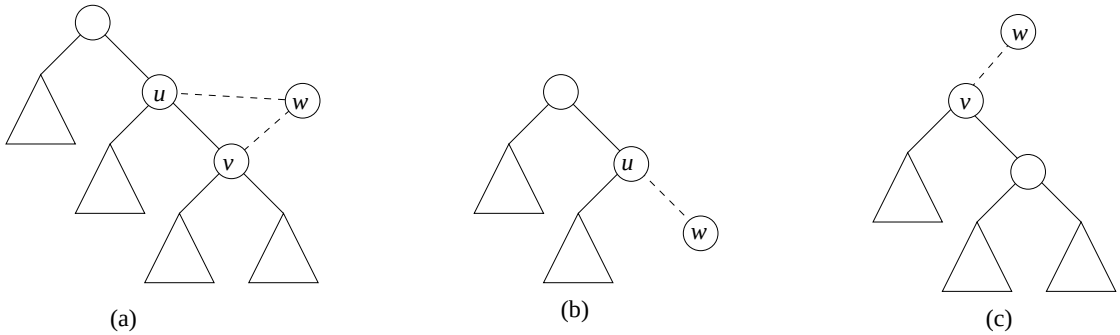


Figure 4.5: Inserting w to obtain $\mathcal{C}(A_i)$ from $\mathcal{C}(A_{i-1})$; dotted lines indicate inserted edges. (a) $a(u) \geq a(w) > a(v)$ (b) $a(l) \geq a(w)$ (c) $a(r) = a(v) < a(w)$.

Given the sequence A stored as a list in external memory, we can implement the above algorithm such that we compute $\mathcal{C}(A)$ and store it as a sorted list C of post-order numbered nodes in external memory using $O(\text{scan}(N))$ I/Os; a post-order numbering of the nodes in $\mathcal{C}(A)$ is the numbering consisting of a recursive numbering of nodes in the left subtree of the root r, followed by a recursive numbering of nodes in the right subtree of r, followed by the numbering of r, and where each node stores the post-order numbers of each of its children. Note that the nodes on the rightmost path of $\mathcal{C}(A)$ have the highest post-order numbers.

To implement the algorithm I/O-efficiently, we maintain the following two invariants for $\mathcal{C}(A_{i-1})$: (1) Except for the nodes on the rightmost path P of $\mathcal{C}(A_{i-1})$, all nodes have been post-order numbered and stored in sorted order in a list C in external memory: (2) Nodes on P are stored on a stack S in the order they appear on P (with the leaf l on top of S), and each node stores the correct number of its left child (stored in C, if existing).

Initially C and S are empty. To compute $\mathcal{C}(A_i)$ from $\mathcal{C}(A_{i-1})$ while maintaining the invariants, we implement the traversal of P from l towards r used to find u and v as follows. Until u is on the top of S (or S is empty), we repeatedly pop a node s from S and insert it after the last element t in C; we number s with the number following the number of t and (except for l) we set its right child number equal to the number of t. Then we set the left child number of the new node w equal to the number of the last element v inserted in C (if existing), and push w on S. After computing $\mathcal{C}(A_N) = \mathcal{C}(A)$, we pop each node s from S in turn and insert it in C, while updating numbers and right child numbers as above.

That the above procedure maintains the first invariant can be seen as follows. Before the procedure, the nodes on the rightmost path of $\mathcal{C}(A_{i-1})$ stored on S have the largest numbers in the post-order numbering of $\mathcal{C}(A_{i-1})$, and by the first invariant the remaining nodes of $\mathcal{C}(A_{i-1})$ are stored in post-order number order in C. Since nodes are popped from S and inserted in C in post-order, the nodes of $\mathcal{C}(A_i)$ in C are also in post-order number order. The left and right child numbers of each node s inserted in C are also correct, since by the second invariant the left child number was already correct before the insertion, and the right child number is explicitly set to the last inserted node t (or left empty in the case of the first inserted node l), which also by the second invariant is the right child of s. That the procedure also maintains the second invariant can be seen as follows. By the second invariant the nodes on P are stored in order on S before the procedure. Since the nodes that are not on P in $\mathcal{C}(A_i)$ are popped from S, and since the only node pushed on S is the new leaf w on P in $\mathcal{C}(A_i)$, the nodes on P are also stored in order on S after the procedure; each node stores the correct left child number, since the left child number of the only new node w is explicitly set to v. After computing $\mathcal{C}(A_N) = \mathcal{C}(A)$, the first invariant implies that all but the nodes on P have been correctly numbered and stored in C. Since by the second invariant, the nodes on P are stored in post-order number order on S, the list C correctly contains all nodes in $\mathcal{C}(A)$ in post-order number order after popping each element from S and inserting it in C.

Overall, the algorithm perform one scan of A and one scan of C, as well as $O(N)$ stack operations. Since a stack can easily be implemented such that each operation takes $O(1/B)$ I/Os (by keeping the top B elements in an internal memory buffer and only reading/writing to disk when the buffer is empty/full), the algorithm uses $O(\text{scan}(N))$ I/Os in total.

Lemma 6 *Given a sequence A of N weights, the Cartesian tree $\mathcal{C}(A)$ can be computed and stored on disk as a sorted list of post-order numbered nodes using $O(\text{scan}(N))$ I/Os.*

4.3.2 Augmented Cartesian Tree

For the assigning Pfafstetter labels, we extend the Cartesian tree to an augmented Cartesian tree $\mathcal{C}_a(A)$ of a sequence $A = (a_1, a_2, \ldots, a_N)$ of N elements. We simply augment each node v of the standard Cartesian tree with copies of the four nodes (post-order number, drainage area, and post-order numbers of children) with the largest weight in the subtree rooted in v. If two nodes have

the same weight, the node with the weight that appear first in A is chosen. Note that one of these largest weight nodes is v itself.

We can easily modify our I/O-efficient Cartesian tree construction algorithm described in Section 4.3.1 to compute an augmented Cartesian tree. During the construction we simply maintain two additional invariants for $\mathcal{C}(A_{i-1})$: (3) Each node t in C stores copies of the four nodes with the largest weights in the subtree of $\mathcal{C}(A_{i-1})$ rooted in t. (4) Each node s on P stored on S store copies of the four nodes with largest weights in the subtree of $\mathcal{C}(A_{i-1})$ rooted in the left child of s.

When we during our algorithm pop a node s from S and insert it after the last element t in C, we know from the invariants that s already contains copies of the four largest weight node in its left subtree and that t contains copies of the four largest weight nodes stored in the subtree rooted in t. Since t is the right child of s, we can easily update s to store copies of the four largest weight nodes in its subtree, that is, fulfill the third invariant, using only the information in s and t. When a new node w is pushed on S, we can easily update it to contain copies of the four largest weight nodes in its left subtree using only the information in the last element t in C, since t is the left child of w. Since the Cartesian tree construction algorithm described in Section 4.3.1 already uses information in the last node t in C when pushing a node on S or popping a node from S, the modification of the algorithm to construct an augmented Cartesian tree can be done without performing any extra I/Os.

Lemma 7 *Given a sequence A of N elements, each with a weight, the augmented Cartesian tree $\mathcal{C}_a(A)$ can be computed and stored as a sorted list of post-order numbered nodes using $O(\text{scan}(N))$ I/Os.*

Observation 1 *The four largest weight nodes stored in the root r of an augmented Cartesian tree $\mathcal{C}_a(A)$ constitute a connected subtree of $\mathcal{C}_a(A)$ rooted in r.*

Proof. The four nodes containing the elements with largest weights trivially include r. Assume that they do not form a connected subtree. Then one of them is a node v, other than r, whose parent u is not one of the four nodes; therefore the weight of u is smaller than the weight of v. This contradicts that the weights of nodes on any root-leaf path in $C_a(A)$ are nondecreasing. □

4.3.3 Labeling a river

We are now ready to describe how to compute the Pfafstetter labels of an augmented river $\mathcal{R}_l^t$ with mouth (root) s_0 and source t. Recall that by the definition of Pfafstetter labels in Section 4.2, the labels of $\mathcal{R}_l^t$ are obtained by first identifying the four tributary mouth nodes v_2, v_4, v_6 and v_8 with largest drainage area, numbered in the order they appear along $\mathcal{R}_l^t$, and labeling them $2, 4, 6, 8$. Then all edges incident to their parents p_2, p_4, p_6 and p_8 are removed, that is, $\mathcal{R}_i^t$ is decomposed into five interbasins rooted in s_0 and the siblings s_2, s_4, s_6 and s_8 of v_2, v_4, v_6 and v_8. Finally, each interbasin is labeled recursively, and the label of each node in the interbasin rooted in s_i is prefixed by $i + 1$. In the case where $\mathcal{R}_l^t$ only has $1 \leq k < 4$ tributary mouth nodes, labels $2k + 2$ through 9 are not assigned; when there are no tributary mouth nodes (when $k = 0$) no label (other than the possible prefix) is assigned.

The augmented Cartesian tree provides us with an easy way of computing the Pfafstetter labels of $\mathcal{R}_l^t$. Consider constructing an augmented Cartesian tree $\mathcal{C}_a(L)$ on the sequence L consisting of the nodes along $\mathcal{R}_l^t$ ordered from mouth to source, where each tributary mouth node v is stored between its parent p and sibling s, and where each river node has weight zero and each tributary mouth node v has weight equal to its drainage area $d(v)$. Refer to Figure 4.6. Note that if $\mathcal{R}_l^t$ has at least one tributary mouth node, then the root r of $\mathcal{C}_a(L)$ corresponds to the tributary mouth node v with largest drainage area. Splitting L at v (while removing v) corresponds to removing the two edges incident to the parent p of v, and results in two sequences $L_l = (s_0, \ldots, p)$ and $L_r = (s, \ldots, t)$

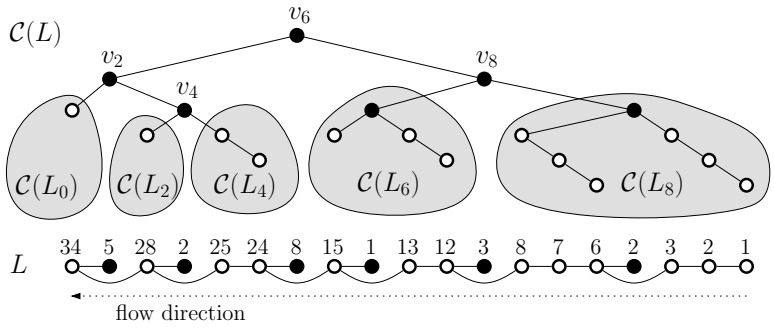


Figure 4.6: *Bottom figure:* An augmented river with drainage areas (as it is stored in L); the weight of river nodes (white circles) is zero and the weight of tributary mouth nodes (black circles) is equal to their drainage area. *Top figure:* Cartesian tree $\mathcal{C}(L)$ with the four tributary mouth nodes v_2, v_4, v_6 and v_8 with largest drainage areas (weight), and the five Cartesian trees $\mathcal{C}(L_0), \mathcal{C}(L_2), \mathcal{C}(L_4), \mathcal{C}(L_6)$ and $\mathcal{C}(L_8)$ for the five interbasins obtained when removing edges incident to their parents p_2, p_4, p_6 and p_8 in L (removing v_2, v_4, v_6 and v_8 from $\mathcal{C}(L)$).

corresponding to two interbasins rooted in s_0 and the sibling s of v. The augmented Cartesian trees rooted in the children of r are exactly $\mathcal{C}_a(L_l)$ and $\mathcal{C}_a(L_r)$. Similarly, if the weights of the four largest weight nodes in L stored in r are all non-zero, they corresponds to the four tributary mouth nodes v_2, v_4, v_6 and v_8 of $\mathcal{R}_l^t$ with largest drainage areas. Splitting L at v_2, v_4, v_6 and v_8 (while removing these nodes) corresponds to removing the edges incident to their parents p_2, p_4, p_6 and p_8, and results in five sequences $L_0 = (s_0, \ldots, p_2), L_2 = (s_2, \ldots, p_4), L_4 = (s_4, \ldots, p_6), L_6 = (s_6, \ldots, p_8)$ and $L_8 = (s_8, \ldots, t)$ corresponding to the five interbasins rooted in siblings s_0, s_2, s_4, s_6 and s_8. By Observation 1, the nodes in $\mathcal{C}_a(L)$ corresponding to v_2, v_4, v_6 and v_8 form a connected subtree rooted in r, and if this subtree is removed, $\mathcal{C}_a(L)$ is decomposed into five subtrees (since it is binary) that are augmented Cartesian trees $\mathcal{C}_a(L_0), \mathcal{C}_a(L_2), \mathcal{C}_a(L_4), \mathcal{C}_a(L_6)$ and $\mathcal{C}_a(L_8)$ for the five interbasins. Thus the Pfafstetter labels of $\mathcal{R}_l^t$ can be obtained by labeling v_2, v_2, v_4 and v_6 with $2, 4, 6$ and 8, respectively, and recursively labeling $\mathcal{C}_a(L_0), \mathcal{C}_a(L_2), \mathcal{C}_a(L_4), \mathcal{C}_a(L_6)$ and $\mathcal{C}_a(L_8)$ while prefixing all labels in $\mathcal{C}_a(L_i)$ with $i + 1$. In the case where only $1 \leq k < 4$ of the weights of the largest weight nodes in L stored in r are non-zero, that is, if $\mathcal{R}_l^t$ only has k tributary mouth nodes $v_2, \ldots, v_{2k}$, removal of the subtree corresponding to $v_2, \ldots, v_{2k}$ decomposes $\mathcal{C}_a(L)$ into $k+1$ augmented Cartesian trees $\mathcal{C}_a(L_0), \ldots, \mathcal{C}_a(L_{2k})$ that can be labeled recursively (that is, labels $2k + 2$ through 9 are not assigned). Finally, if the weights of all nodes stored in r are zero, $\mathcal{R}_l^t$ does not have any tributary mouth nodes and no labels (other than the possible prefix) should be assigned to $\mathcal{C}_a(L)$.

Based on the above observations, we can design an I/O-efficient algorithm for Pfafstetter labeling an augmented river $\mathcal{R}_l^t$. We assume that $\mathcal{R}_l^t$ is given as a list L_l of N nodes along $\mathcal{R}_l^t$ ordered from mouth to source, where each node also store a copy of its (possible) tributary mouth child; each node/child has a unique number and contains a drainage area value. In a single scan of L_l we first produce the sequence L consisting of the nodes along $\mathcal{R}_l^t$ ordered from mouth to source, where each tributary mouth node v is stored between its parent p and sibling s, and where each river node has weight zero and each tributary mouth node v has weight equal to its drainage area $d(v)$. Then we construct an augmented Cartesian tree $\mathcal{C}_a(L)$ on L, stored as a sorted list C of post-order numbered nodes.

Next we label each node in C (storing all labels in a list C_p) using a recursive traversal of $\mathcal{C}_a(L)$ as outlined above, where we always recursively visit the right subtree of a node v before recursively visiting the left subtree of v, and where we explicitly implement the recursion stack S. The stack S can contain two types of elements, namely *label* and *recursion* elements, both consisting of (the number of) a node v of $\mathcal{C}_a(L)$ and a Pfafstetter label (prefix) P. Initially, S contains a recursion element for the root r of $\mathcal{C}_a(L)$ (that is, an element with number N) and an empty label. We

repeatedly pop an element from S and access the corresponding node v in C. If the element is a label element, we simply label v with P and insert it at the end of C_p. If it is a recursion element, we want to label the subtree of $\mathcal{C}_a(L)$ rooted in v, while prefixing all labels with P. To do so, we consider the four largest weight nodes v_2, v_4, v_6 and v_8 stored with v in C. Assume first that their weights are all non-zero. In this case we label v_2, v_4, v_6 and v_8 by pushing a label element for each v_i on S with the label P followed by i; we also recursively label $\mathcal{C}_a(L_0), \mathcal{C}_a(L_2), \mathcal{C}_a(L_4), \mathcal{C}_a(L_6)$ and $\mathcal{C}_a(L_8)$ by pushing a recursion element for each of their roots (obtained from v_2, v_4, v_6 and v_8) with labels P followed by $1, 3, 5, 7$ and 9, respectively, on S. We push the elements in the order they appear in a post-order traversal of the subtree rooted in v, where left subtrees are visited before right subtrees; note that this means that they appear in reverse post-order traversal order on S. In the case where only $1 \leq k < 4$ of the largest weight nodes stored with v in C are non-zero, we only push label elements corresponding to these nodes $v_2, \ldots, v_{2k}$ and recursion elements corresponding to $\mathcal{C}_a(L_0), \ldots, \mathcal{C}_a(L_{2k})$. Finally, if the weights of all the largest weight nodes stored with v in C are zero, we simply label v with P and insert it at the end of C_p, and push two recursion elements with label P on S; first an elements for the left child of v and then an elements for the right child of v (note that this will eventually label the whole subtree rooted in v with P).

That the above algorithm correctly computes the Pfafstetter label of $\mathcal{R}_l^t$ follows from the above discussion. The sequence L is constructed from L_l in a single scan using $O(\text{scan}(N))$ I/Os, and C is also constructed in $O(\text{scan}(N))$ I/Os (Lemma 7). Since we visit the nodes in $\mathcal{C}_a(L)$ in reverse post-order, the N accesses to C correspond to a backwards scan of C, and are therefore performed in $O(\text{scan}(N))$ I/Os. If T is the total size of the computed Pfafstetter labels, the labels are written to C_p in $O(\text{scan}(T))$ I/Os, and the $O(N)$ stack operations can also be performed in $O(\text{scan}(T))$ I/Os (since the combined size of the labels pushed on S is $O(T)$). Thus $\mathcal{R}_l^t$ is labeled in $O(\text{scan}(T))$ I/Os in total.

After computing the labels of the nodes in C (stored in C_p), we can easily label the corresponding nodes in L_l in a single sorting step. However, by essentially reversing the way C was produced from L, we can also easily do so in $O(\text{scan}(T))$ I/Os.

Lemma 8 *Given an augmented river $\mathcal{R}_l^t$ as a ordered list L_l of N numbered nodes along $\mathcal{R}_l^t$, where each node also store its (possible) tributary mouth child and each node/child contains a drainage area value, the Pfafstetter labels of $\mathcal{R}_l^t$ can be computed and stored with the nodes in L_l in $O(\text{scan}(T))$ I/Os, where T is the total size of the labels of all nodes in $\mathcal{R}_l^t$.*

Remarks. (i) Our algorithm can easily be modified to handle augmented rivers where each river node can have more than one tributary mouth child (non-binary flow trees) in the same I/O-bound. (ii) If each Pfafstetter label consists of a constant number of digits (elements), e.g. if we truncate them, our algorithm only uses $O(\text{scan}(N))$ I/Os. (iii) Our algorithm runs in $O(T)$ time.

4.4 Labeling flow tree

In this section, we describe simple and I/O-efficient algorithms for computing the tributary tree $\mathcal{T}^t$ from a binary flow tree $\mathcal{T}$, and for computing the Pfafstetter labels of $\mathcal{T}$ given labels for the individual augmented rivers in the nodes of $\mathcal{T}^t$.

4.4.1 Computing tributary tree

Recall that the root r of the tributary tree $\mathcal{T}^t$ for the flow tree $\mathcal{T}$ contains the augmented river $\mathcal{R}_r^t$ obtained by starting at the root ρ of $\mathcal{T}$ and in each node continue to the child with the largest drainage area, while also including the (possible) tributary mouth child of each node. The root r has a child for each tributary mouth node in $\mathcal{R}_r^t$, which contain recursively defined tributary trees

for each of the tributary basins obtained if the river nodes of $\mathcal{R}_r^t$ are removed from $\mathcal{T}$. In other words, $\mathcal{T}^t$ defines a decomposition of $\mathcal{T}$.

We assume that $\mathcal{T}$ is given as an (unordered) list of nodes numbered from 1 to N, where each node v contains a drainage area value $d(v)$ and the numbers of its (at most two) children. Since each node in $\mathcal{T}^t$ contains a potentially long augmented river, we want to compute a somewhat different *pre-order list representation* of $\mathcal{T}^t$: The augmented river $\mathcal{R}_l^t$ in a node l of $\mathcal{T}^t$ is represented as a list L_l of each river node along $\mathcal{R}_l^t$ ordered from mouth to source, where each node also stores a copy of its (possible) tributary mouth child. The lists L_l for all nodes l in $\mathcal{T}^t$ are stored consecutively in one list L^t in the order corresponding to a pre-order traversal of $\mathcal{T}^t$, where the children of node l are visited in the order that the corresponding river mouths appear along the augmented river $\mathcal{R}_l^t$ from mouth to source.

As discussed in Section 4.2, the definition of $\mathcal{T}^t$ suggests a natural algorithm for computing $\mathcal{T}^t$ from $\mathcal{T}$ based on a DFS-traversal of $\mathcal{T}$, where in each step the child with largest drainage area is chosen. Our algorithm for computing $\mathcal{T}^t$ in $O(\text{sort}(N))$ I/Os and $O(N)$ time is based on this idea, where we also utilize an important property of the flow tree $\mathcal{T}$, namely that the drainage area of nodes on any root-leaf path in $\mathcal{T}$ is strictly increasing (since the drainage area $d(v)$ of a node v in $\mathcal{T}$ is equal to the size of the subtree rooted in v). We do not actually perform a DFS-traversal of $\mathcal{T}$, constructing one augmented river at a time, but traverse the nodes of $\mathcal{T}$ in drainage area order, while at any given time being in the process of constructing several augmented rivers in parallel. We can do so since a visit in drainage area order ensures that when visiting a node w in $\mathcal{T}$, its parent v has already been visited; thus we have already determined if w is the mouth of a new augmented river or the next node on the augmented river through v.

To perform the traversal we first create a list L of the nodes in $\mathcal{T}$ sorted primarily by decreasing drainage area and secondarily by node number. With each node in L we also store a copy of the weights of each of its (at most two) children in $\mathcal{T}$. Note that the root ρ of $\mathcal{T}$ will be the first node in L. Next we assign all nodes in each augmented river $\mathcal{R}_l^t$ in $\mathcal{T}^t$ a unique *river id* number between 1 and the drainage area $d(\rho)$ of ρ. Note that we have enough river id's since the number of augmented rivers in $\mathcal{T}^t$ obviously is smaller than $d(\rho) = N$; in general, if the river id of an augmented river $\mathcal{R}_l^t$ of node l of $\mathcal{T}^t$ is i, then the augmented rivers in the subtree rooted in l have river id's between i and $i-1$ plus the drainage area of the mouth of $\mathcal{R}_l^t$. Conceptually, we want to assign river id's by initially assigning ρ river id $r(\rho) = 1$, and then scan through L and when processing a node v assign river id's to v's children based on the river id $r(v)$ of v. However, a straightforward implementation of this idea where we directly access v's children in L to assign them river id's would lead to scattered accesses and thus an $\Omega(N)$ I/O-algorithm. Instead, we implement the idea by maintaining an I/O-efficient priority queue P with the river id's of nodes that have not yet been processed but has been assigned a river id because their parents were processed. The priority of a node w in P is equal to w's position in L, that is, primarily its drainage area $d(w)$ and secondarily its node number; with w in P we also store the river id to be assigned to the next tributary mouth along the augmented river containing w. Initially, P contains ρ with $r(\rho) = 1$ and and *next river id* $n(\rho) = 2$. This way we know that when processing a node v in the scan through L, the maximal priority element in P is v. Therefore, to process v we simply first extract the maximal element from P to obtain the river id $r(v)$ of v and augment v in L with $r(v)$. Unless v is a leaf, we then assign river id's to the children of v as follows: If v has one child w, it must be the next node on the augmented river containing v. Therefore we insert an element in P for w with river id $r(w) = r(v)$ and next river id $n(w) = n(v)$; the priority of w (drainage area $d(w)$ and node number of w) is obtained from the child information stored with v in L. If v has two children, the child w with the largest drainage area $d(w)$ is by definition the next node on the augmented river containing v, and the other child u is the mouth of another augmented river in one of the tributary basins of the river containing v. To start numbering the augmented river with mouth u, we insert an element for u in P with river id $r(u) = n(v)$ and next id $n(u) = n(v) + 1$. We also insert an element in P for w with river id $r(w) = r(v)$ and next river id $n(w) = n(v) + d(u)$. For both new elements in P, the priority

(drainage area and node number) is obtained from the child information stored with v in L. After this, we continue with the next node in L.

After having processed all nodes in L, all nodes in each augmented river have the same river id, since a node on a augmented river is always assigned the same river id as its parent river-node (if existing). To prove that all augmented rivers have unique river id's, we show that our algorithm assign id's such that for each node v (except the source) on augmented river $\mathcal{R}_l^t$, the augmented rivers in tributary basins from v and upstream on $\mathcal{R}_l^t$ have unique river id's in the range from $n(v)$ to $n(v)+d(v)-2$; this obviously means that all rivers have unique river id's in the range from $r(\rho)=1$ to $n(\rho)+d(\rho)-2=2+N-2=N$. We do so by induction on drainage area values. The statement obviously holds for the base case of a node v with $d(v)=2$, since the single child of v is assigned river id $r(v)$ (as v). Consider then a node v on $\mathcal{R}_l^t$ and assume that our algorithm correctly assigns river id's to nodes with drainage area less than $d(v)$. If v has one child w, it is assigned river id $r(w)=r(v)$ and $n(w)=n(v)$, and by induction tributary basins from w and upstream along $\mathcal{R}_l^t$ are assigned unique river id's in the range between $n(w)=n(v)$ and $n(w)+d(w)-2=n(v)+d(v)-3$; thus augmented rivers in tributary basins upstream from v also have river id's in the range between $n(v)$ and $n(v)+d(v)-3<n(v)+d(v)-2$ as required. If v has two children, we by induction assign the augmented rivers in the tributary basin with mouth in the lowest drainage area child u unique river id's in the range between $r(u)=n(v)$ and $n(u)+d(u)-2=n(v)+d(u)-1$ (by inserting u in P with $r(u)=n(v)$ and $n(u)=n(v)+1$); by induction, we assign the augmented rivers in tributary basins from the other child w and upstream along $\mathcal{R}_l^t$ unique river id's in the range between $n(w)=n(v)+d(u)$ and $n(w)+d(w)-2=n(v)+d(u)+d(w)-2$ (by inserting w in P with $n(w)=n(v)+d(u)$). In total, augmented rivers in tributary basins from v and upstream in $\mathcal{T}_l^t$ have unique river id's in the range between $n(v)$ and $n(v)+d(u)+d(w)-2=n(v)+d(v)-3<n(v)+d(v)-2$ as required, since $d(v)=1+d(u)+d(w)$.

After assigning unique river id's to augmented rivers, the list L_l for a node l in $\mathcal{T}^t$ (consisting of an ordered list of river node along $\mathcal{R}_l^t$, where each node also store a copy of its (possible) tributary mouth child), simply consists of the nodes in L with the corresponding augmented river id sorted by drainage area (where we for each node v have removed the copy of the river child of v). Furthermore, since we assign river id's such that the id of a river $\mathcal{R}_l^t$ is lower than the id's of all rivers corresponding to nodes in the subtree of $\mathcal{T}^t$ rooted at l, and such that the id's of all rivers in a tributary basin of l whose mouth is closer to the mouth of $\mathcal{R}_l^t$ (further downstream) are smaller than the id's of all rivers in a tributary basin of l whose mouth is further away (upstream) from the mouth of $\mathcal{R}_l^t$, the lists L_l can be ordered according to the required pre-order traversal of $\mathcal{T}^t$ simply by sorting them by river id. Thus we can produce list L^t consisting of the lists L_l for all nodes l in $\mathcal{T}^t$ corresponding to a pre-order traversal of $\mathcal{T}^t$, simply by sorting L by river id and secondarily by drainage area.

All that remain is to analyze how many I/Os the above algorithm use. The list L can easily be constructed in $O(\text{sort}(N))$ I/Os using a constant number of $O(\text{sort}(N))$ sorting and $O(\text{scan}(N))$ scanning steps on the input list I of nodes in $\mathcal{T}$: We simply first make a copy L of I and sort it by the number of the first child of each node. Then we sort I, and scan L and I simultaneously to add a copy if the first child of each node v in L (if existing) to v. Then we add a copy of second children (if existing) in a similar way using a sort and a scan step. Finally, we sort L primarily by drainage area and secondarily by node numbers. In the assignment of river labels we scan L and perform $O(N)$ priority queue operations. Using an I/O-efficient priority queue, all the priority queue operations can be performed in $O(\text{sort}(N))$ I/Os [10, 31]. Finally, we use $O(\text{sort}(N))$ I/Os to sort L to obtain $\mathcal{T}^t$.

Lemma 9 *Given flow tree $\mathcal{T}$ as an unordered list of nodes (with drainage area and child numbers), a pre-order list representation of the tributary tree $\mathcal{T}^t$ for $\mathcal{T}$ can constructed in $O(\text{sort}(N))$ I/Os.*

4.4.2 Labeling flow tree using tributary tree labels

Next we describe how to compute the Pfafstetter labels of a flow tree $\mathcal{T}$, represented as an (unordered) list of labeled nodes, given Pfafstetter labels of the individual augmented rivers in the tributary tree $\mathcal{T}^t$ of $\mathcal{T}$. As above, let the augmented river $\mathcal{R}_l^t$ in a node l of $\mathcal{T}^t$ be represented as a list L_l of each river node along $\mathcal{R}_l^t$ ordered from mouth to source, where each node also stores a copy of its (possible) tributary mouth child (tributary mouths). Assume that $\mathcal{R}_l^t$ has been labeled, that is, that each node in L_l, as well as its (possible) tributary mouth child, has been assigned a Pfafstetter label. Assume that we are given the lists L_l for all nodes l in $\mathcal{T}^t$ stored consecutively in one list L^t in the order corresponding to a pre-order traversal of $\mathcal{T}^t$, where the children of node l are visited in the order the corresponding tributary mouths appear along the augmented river $\mathcal{R}_l^t$ from mouth to source.

As discussed in Section 4.2, we can obtain the Pfafstetter labels of all nodes in the flow tree $\mathcal{T}$ from a labeling of the augmented rivers in the tributary tree $\mathcal{T}^t$, simply by keeping the labels of the river-nodes on the augmented river $\mathcal{R}_r^t$ (the main river) in the root r of $\mathcal{T}^t$, and recursively labeling the nodes in each subtree of r, while prefixing the labels in the subtree rooted in child or r with the label of the corresponding tributary mouth. Since we are given $\mathcal{T}^t$ in pre-order list representation in L^t, we can easily implement this procedure I/O-efficiently using a stack S of Pfafstetter label prefixes. Initially, S contains an empty prefix. We scan through L^t, labeling the river-nodes in each augmented river $\mathcal{R}_l^t$ when scanning L_l. To process L_l, we first pop a prefix from S; then we scan through L_l and add this prefix to the Pfafstetter labels of all nodes (river-nodes as well as copies of tributary mouth children). During the scan we also push the Pfafstetter label of each tributary mouth child (that is, tributary mouth stored with river-nodes in L_l) on an auxiliary stack S' as they are meet in L_l. Finally, we pop each element from S' in turn and push it on S; note that this means that the tributary mouth labels are pushed on S in the reverse order the tributaries appear on $\mathcal{R}_l^t$, from source to mouth. This means the tributary mouth label of the tributary closest to the mouth of l is on the top of S. After finishing the scan of L^t, all river-nodes have been labeled, and in another scan of L^t we can produce a list of labeled nodes in $\mathcal{T}$ (without all the extra information stored with each node in L^t).

To prove the correctness of the above algorithm, we show that it maintains the following invariant: Just before processing L_l, that is, node l in $\mathcal{T}^t$, the correct prefix to be added to all Pfafstetter labels in the subtree rooted in l is on top of S, and except for this prefix S is maintained during the labeling of the subtree. The invariant obviously is true before labeling of the root r of $\mathcal{T}^t$, that is, scan of L_r. Consider the processing of node l. To label L_l the algorithm pops a prefix from S and adds it to all labels in L_l; it also pushes the labels of the tributary mouths on S in the order they appear on $\mathcal{R}_l^t$ from mouth to source. By the invariant, this means that all river-nodes in $\mathcal{R}_l^t$ are now labeled correctly. It also means that the prefixes that need to be added to the labels of each of the subtrees rooted in the children of l are stored on top of S in the order the corresponding tributary mouths appear on $\mathcal{R}_l^t$ from mouth to source. This is exactly the order the children are visited in the pre-order traversal of $\mathcal{T}^t$. Thus in particular the top prefix on S is the correct prefix for the next node m to be visited as required by the invariant. The invariant also implies that except for this top prefix, S is intact after processing m and its ancestors. Thus at that time the correct prefix for the next child of l is on the top of S as required. This process continues until all the children of l have been processed. After that we have correctly labeled the subtree rooted in l, and except for the top prefix, the stack S contains the same prefixes as before we started labeling l as required.

If the total size of the Pfafstetter labels of all nodes in $\mathcal{T}$ is T, the two scans of L^t takes $O(\text{scan}(T))$ I/Os. Apart from the scans of L^t, we push and pop the Pfafstetter label of each of the tributary mouths of the $O(N)$ augmented rivers in $\mathcal{T}^t$ a constant number of times on S. Since a stack can easily be implemented such that each operation takes $O(1/B)$ I/Os, we use $O(\text{scan}(T))$ I/Os in total on the stack operations. Thus the algorithm use $O(\text{scan}(T))$ I/Os in total.

Lemma 10 *Given a pre-order list representation of the tributary tree $\mathcal{T}^t$ for a flow tree $\mathcal{T}$, where each river has been assigned a Pfafstetter label individually, the Pfafstetter labels of $\mathcal{T}$, represented as an (unordered) list of nodes, can be constructed in $O(\text{scan}(T))$ I/Os. Here T is the total size of the labels of all nodes in $\mathcal{T}$.*

Altogether we have now shown how to compute Pfafstetter labels of a flow tree $\mathcal{T}$: We first compute a pre-order list representation of the tributary tree $\mathcal{T}^t$ for $\mathcal{T}$ in $O(\text{sort}(N))$ I/Os (Lemma 9). Then we compute the labels for each of the augmented rivers $\mathcal{R}_l^t$ in $\mathcal{T}^t$. The number of I/Os needed to compute the labels for an augmented river $\mathcal{R}_l^t$ is $O(T_l/B))$, where T_l is the total size of the labels of $\mathcal{R}_l^t$ (Lemma 8). Note that if $T_l < B$, the computation can be performed without any I/Os other than the ones needed to load L_l into main memory. Thus in total the labels for all augmented rivers are computed in $O(\text{scan}(N)) + O(\text{scan}(T)) = O(\text{scan}(T))$ I/Os. Finally, we compute the labels of $\mathcal{T}$ from the labeling of the individual augmented rivers in $\mathcal{T}^t$ using $O(\text{scan}(T))$ I/Os (Lemma 10).

Theorem 1 *The Pfafstetter labels of a flow tree $\mathcal{T}$ can be constructed in $O(\text{sort}(N) + \text{scan}(T))$ I/Os, where T is the total size of the labels of all nodes in $\mathcal{T}$.*

Remarks. (i) Our algorithm can easily be modified to handle non-binary flow trees in the same I/O-bound. (ii) Our algorithm can also easily be modified to handle forests rather than trees in the same I/O-bound. (iii) If each Pfafstetter label consists of a constant number of digits (elements), e.g. if we truncate them, our algorithm only uses $O(\text{sort}(N))$ I/Os. (iiii) If $\mathcal{T}$, $\mathcal{T}^t$ and all labels fit in memory, we can easily design a Pfafstetter labeling algorithm that uses $O(T)$ time, since the sorting and priority queue steps can then easily be avoided.

4.5 Implementation and experimental results

In this section, we present the results of an experimental study of our Pfafstetter labeling algorithm. We first in Section 4.5.1 discuss how we implemented our algorithm to be able to handle general grid DEMs (as opposed to the simplified case considered in the previous sections). In Section 4.5.2 and Section 4.5.3 we then discuss the data used and experimental results, respectively.

4.5.1 Implementation

In Section 4.2 we discussed how we can obtain a flow tree $\mathcal{T}$ from a grid DEM that (other than the boundary cells) *does not* contain any cells without a lower neighbor, simply by assigning each cell a flow direction to the lowest of its lower neighbors and from each boundary cell without a lower neighbor to a special cell ρ (the outside sink). Given the grid DEM with N cells in row (or column) major order, we can easily in $O(\text{scan}(N))$ I/Os construct a representation of $\mathcal{T}$ consisting of an unordered list of numbered nodes, where each node contains the numbers of its children, simply by scanning through the grid three rows at a time, while for each cell looking at cells in a 3×3 neighborhood.

In the common case where the initial grid DEM *does* contain cells other than boundary cells without lower neighbors, often parts of sinks due to noise or plateaus of *flat cells*, the above procedure leads to a forest of trees, since each cell without a lower neighbor becomes the root of a separate flow tree. Simply computing Pfafstetter labels for such a forest does not lead to very realistic watersheds, because treating each flat cell as a sink does not model global water flow very well. We can remove noise and route flow on flat cells using the methods and algorithms described in Chapter 3. In particular we can use TerraFlow [14] to remove noise and produce drainage area and flow direction grids from noisy input DEMs. After that we can compute Pfafstetter labels in

Dataset	Ten 1	Ten 2	Ten 3	Ten 4	Neuse
Input size (MB)	17	116	150	713	5,819
Size (mln cells)	2.7	21.7	30.8	147.0	396.5
Running time	0m30	6m51	10m29	58m10	187m43

Table 4.1: Size and Pfafstetter labeling time for the five DEMs.

$O(\text{sort}(T))$ I/Os using our algorithm described in the previous sections, modified to work on a flow forest rather than a flow tree and to handle flow trees that are not binary.

To obtain the most realistic watershed hierarchy (Pfafstetter labels), we implemented our algorithm to work on hydrologically conditioned grid DEM models, where all cells, including flat cells on plateaus, have already been assigned a flow direction, as well as had their drainage area computed. Our implementation takes two input grids corresponding to a flooded DEM, namely the corresponding flow directions and the corresponding drainage areas. From these inputs, we obtain the unordered list representation of $\mathcal{T}$ used in our Pfafstetter algorithm by a simple simultaneous scan of the two grids using $O(\text{scan}(N))$ I/Os; in the same scan we also augment each node with the grid position of the corresponding cell. After that our implementation follows the algorithm described in the previous sections (modified to handle a non-binary flow tree), and after computing Pfafstetter labels of all nodes, we sort the nodes by grid position using $O(\text{sort}(T))$ I/Os to obtain an output Pfafstetter label grid. (Optionally, we allow the user to truncate labels to a maximum length, so that each label fits in $O(1)\log N$-bit words and the sorting of labels can be done in $O(\text{sort}(N))$ I/Os). We implemented our algorithm in C++ using TPIE [11], a library that provides support for implementing I/O-efficient algorithms and data structures. The implementation work was greatly simplified by the fact that all main primitives of our algorithm—scanning, sorting, stacks and priority queues—are already implemented I/O-efficiently in TPIE.

4.5.2 Datasets

To investigate the practical performance of our algorithms, as well as the realism of the computed watersheds, we conducted a set of experiments with five grid DEMs of varying size. The largest DEM covered the Neuse river basin in North Carolina at a resolution of 20 feet. It contained 396.5 million cells (such that the flow directions and drainage areas occupied 5.8Gbytes), and is publicly available from ncfloodmaps.com. The other four DEMs covered sub-basins of the upper Tennessee river basin at a resolution of one arc second (approximately 100 feet) and contained 2.7, 21.7, 30.8 and 147 million cells, respectively; these datasets are from the National Elevation Dataset (NED) from the United States Geological Survey, publicly available at seamless.usgs.gov.

4.5.3 Experimental results

For each of the five input DEMs we used TERRAFLOW to compute hydrologically conditioned DEMs which removes local minima from the terrain using flooding. With TERRAFLOW, we also compute flow directions and drainage area, and then we used our implementation to compute Pfafstetter labels, truncated to nine digits. The experiments were run on a Dell Precision Server 370 (Pentium 4 3.40 GHz processor) with hyperthreading enabled and running Linux 2.6.11. The machine had 1 GB of physical memory, but we made sure that our implementation never used more than 256 MB by setting a kernel flag to limit memory to 256 MB and instructing TPIE to abort if more memory than this limit was allocated. All data was stored on a single 400 GB SATA disk drive.

Table 4.1 shows the time used to label each of the five input DEMs, not counting the time used by TERRAFLOW. In all cases, the time taken by TERRAFLOW was more than five times the time taken by the Pfafstetter labeling routine. Table 4.2 shows how much time is spent in the various

Dataset	Ten 1	Ten 2	Ten 3	Ten 4	Neuse
Constructing $\mathcal{T}$	16%	9%	8%	7%	16%
Sorting nodes in $\mathcal{T}$	12%	16%	16%	15%	13%
Decomp. $\mathcal{T}$ into aug. rivers	43%	30%	31%	34%	30%
Sorting to obtain $\mathcal{T}^t$	9%	19%	19%	20%	19%
Labeling $\mathcal{T}^t$ and $\mathcal{T}$	5%	8%	7%	6%	6%
Sorting labeled cells	8%	13%	14%	13%	12%
Exporting data	6%	4%	5%	4%	5%

Table 4.2: Breakdown of labeling time for each of the five DEMs.

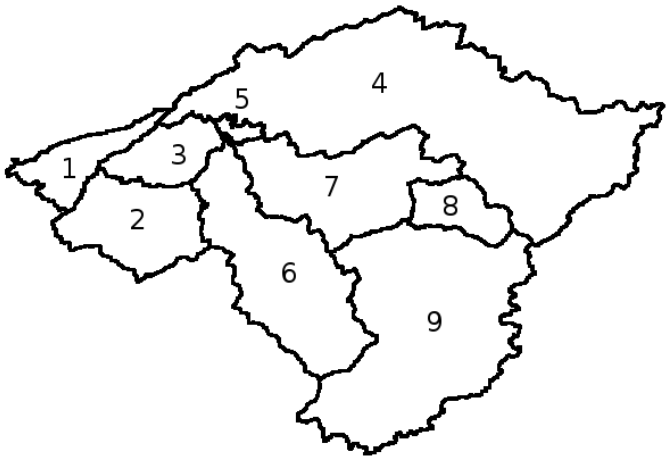
phases of the algorithm, as a percentage of total time. Decomposing $\mathcal{T}$ into a set of augmented rivers is the most time consuming phase of the algorithm. This is not unexpected, since this phase uses a priority queue and performs $O(\text{sort}(N))$ I/Os. The other sorting phases (also using $O(\text{sort}(N))$ I/Os) consume significant portions of the overall time as well. Interestingly, labeling $\mathcal{T}$ and $\mathcal{T}^t$ (using the augmented Cartesian tree) is a small fraction of the total time (this is somewhat expected, since $O(\text{scan}(N)) < O(\text{sort}(N))$). It is also interesting to note that reading and importing the initial grids (constructing $\mathcal{T}$) and exporting the final results is not an insignificant portion of the total time. Overall, we conclude that our algorithm is practically, as well as theoretically, efficient.

To investigate how Pfafstetter label watersheds computed using our algorithm align with the published digital USGS 8-digit HUCs, we compared the two for a portion of the French Broad–Holston river basin (Ten 3 in the Tables and USGS HUC 060101). USGS 8-digit HUCs are freely available online as part of the USGS National Elevation Data (NED) set. Both the USGS boundaries and the Pfafstetter boundaries were derived from 30 meter NED grids, the the exact method of producing the USGS boundaries is not known. As can be seen on Figure 4.7, the watershed boundaries agree well. The Pfafstetter method always divides the basin into nine sub-basins, whereas the USGS HUC only has four sub-basins in the area. Pfafstetter basins are defined by the location and size of the four largest tributaries, and these basins can vary widely in size. Note in Figure 4.7 that interbasin 5 is rather small because basins 4 and 6 are close to eachother along the main river. In situations where this variation in size is undesirable, Pfafstetter basins can easily be combined to form basins that are of approximately the same extent as the USGS basins (e.g., Pfafstetter basins 7,8, and 9 can be combined to approximate USGS sub-basin 05). This is easy in the Pfafstetter case, because the labels encode the topology and basins can be merged and split in an intellegent and automatic way to construct basins of similar size. It would be much more difficult to split or combine USGS HUCs automatically as there are no fixed rules regarding the numbering of USGS HUCs. A close inspection of the overlay of the two watershed decompositions show minor discrepancies between Pfafstetter and USGS HUC watersheds, but our Pfafstetter labels are consistent with the underlying elevation, flow direction and flow accumulation data. This consistency across multiple data layers is desirable in many GIS applications and avoids the need to rely on multiple heterogeneous data sets.

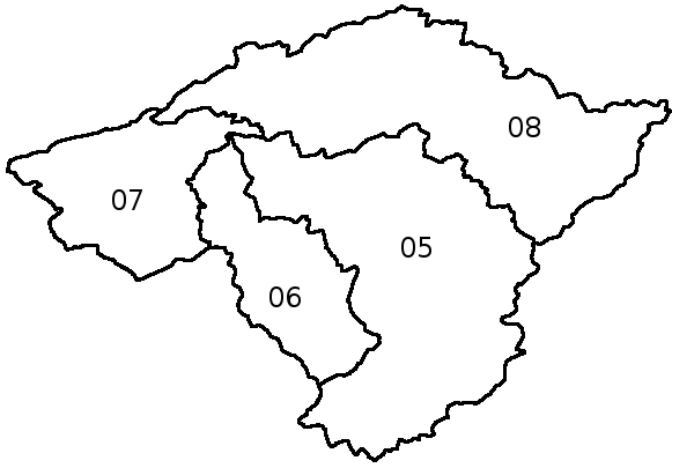
4.6 Conclusion

In this chapter we presented an I/O-efficient algorithm for computing the Pfafstetter label of each cell of a grid terrain model. We also presented the results of a preliminary experimental study that showed that our algorithm is practically as well as theoretically efficient. This algorithm can be used for automated computation of watershed hierarchies and has the added advantage that the Pfafstetter labels encode upstream and downstream relationships of hydrological units. For example, one can tell that basin 92 is upstream of interbasin 35 because basin 9 is upstream of

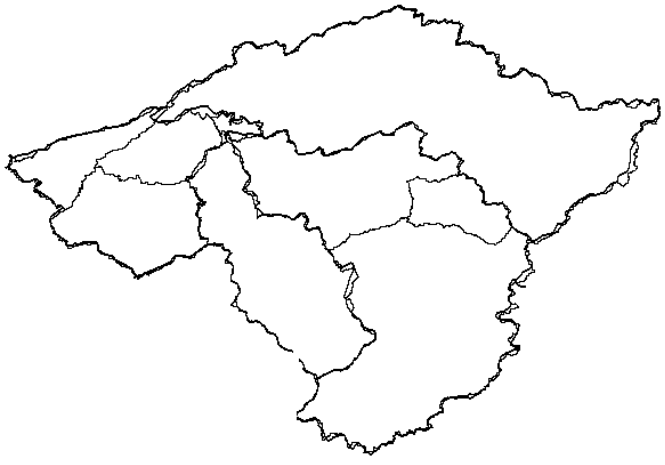
interbasin 3 by definition, 92 is a sub basin of basin 9, and 35 is an interbasin inside basin 3 that is along the main river of basin 3 into which basin 9 flows. Similar logic can be applied to no water from basin 93 flows into basin 36. The automated method, topological encoding, and scalability to large data sets are three major advantages of our approach over existing watershed decomposition methods.



(a) Pfafstetter



(b) USGS



(c) Overlay

Figure 4.7: Comparison of Pfafstetter label watersheds to USGS HUCs in the French Broad–Holston river basin (HUC 060101). Common boundaries are generally in good agreement.

Chapter 5

Planar Point Location

5.1 Problem Definition

The planar point location problem is the problem of storing a planar subdivision defined by N line segments such that the region containing a query point p can be computed efficiently. Planar point location has many applications in, e.g., Geographic Information Systems (GIS), spatial databases, and graphics. In many of these applications, the datasets are larger than the size of physical memory and must reside on disk. Therefore, we are interested in planar point location structures that minimize the number of I/Os needed to answer a query.

While several theoretically I/O-efficient planar point location structures have been developed, e.g., [51, 1, 25], they are all relatively complicated and consequently none of them have been implemented. Based on a bucket approach, Vahrenhold and Hinrichs developed a simple and practically efficient, but theoretically non-optimal, heuristic structure [86]. In this chapter, we show that a point location structure based on a persistent B-tree is efficient both in theory and practice; the structure obtains the theoretical optimal bounds and our experimental investigation shows that, for a wide range of real-world GIS data, it uses a similar number of I/Os to answer a query as the structure of Vahrenhold and Hinrichs. For a synthetically generated worst-case dataset the structure uses significantly less I/Os to answer a query.

5.1.1 Previous Results

In the RAM model, several linear space planar point location structures that can answer a query in optimal $O(\log_2 N)$ time have been developed, e.g., [43, 76, 60]. One of these structures, due to Sarnak and Tarjan [76], is based on a persistent search tree. A persistent data structure maintains a history of all updates performed on it, such that queries can be answered on any of the previous versions of the structure, while updates can be performed on the most recent version thus creating a new version.[1] See the recent survey by Snoeyink [79] for a full list of RAM model results.

In the I/O model, Goodrich et al. [51] developed an optimal static point location structure using linear space and answering a query in $O(\log_B N)$ I/Os. Agarwal et al. [1] and Arge and Vahrenhold [25] developed dynamic structures. Several structures for answering a batch of queries have also been developed [51, 26, 36, 86]. Refer to [9] for a survey.

While these structures are all theoretically I/O-efficient, they are all relatively complicated and consequently none of them have been implemented. Based on an internal memory bucket approach [41], Vahrenhold and Hinrichs therefore developed a simple but non-optimal heuristic structure, which performs well in practice [86]. The main idea in this structure is to impose a grid on the segments that defines a subdivision and store each segment in a "bucket" corresponding to each grid cell it intersects. The grid is constructed such that for certain kinds of "nice data", each segment is stored in $O(1)$ buckets such that the structure requires linear space, and each bucket contains $O(B)$ segments such that a query can be answered in $O(1)$ I/Os. In the worst case however, each segment may be stored in $\Theta(\sqrt{N/B})$ buckets and consequently the structure may use $\Theta(\frac{N}{B}\sqrt{N/B})$ space. In this and some other cases, there may be a buckets containing $O(N)$ segments such that a query takes $O(N/B)$ I/Os.

[1] The type of persistence we describe here is often called *partial persistence* as opposed to *full persistence* where updates can be performed on any previous version.

Most of the structures in the above results actually solve a slightly generalized version of the planar point location problem, namely the *vertical ray-shooting problem*: Given a set of N non-intersecting segments in the plane, the problem is to construct a data structure such that the segment directly above a query point p can be found efficiently. This is also the problem we consider in this chapter.

Using a general technique by Driscoll et al. [40], persistent versions of the B-tree have also been developed [29, 88]. A persistent B-tree uses $O(N/B)$ space, where N is the number of updates performed, and updates and range queries can be performed in $O(\log_B N)$ and $O(\log_B N + T/B)$ I/Os, respectively [29, 88]; note that the structure requires that all elements stored in it during its entire lifespan are comparable, that is, that the elements are totally ordered. In Sarnak and Targan's application of persistence, not all elements (segments) stored in the structure over its lifespan are comparable; thus a similar I/O-efficient structure cannot directly be obtained using a persistent B-tree.

5.1.2 Our Results

Our main result [17] is an external data structure for vertical ray-shooting (and thus planar point location) that is I/O-efficient both in theory and practice. The structure, described in Section 5.2, uses linear space and answers a query in optimal $O(\log_B N)$ I/Os. It is based on the persistent search tree idea of Sarnak and Tarjan [76]. As a theoretical contribution of independent interest, we show how to modify the known persistent B-tree such that only elements present in the same version of the structure need to be comparable, that is, so no total order is needed. In Section 5.3, we then present an extensive experimental evaluation of our structure's practical performance compared to the heuristic grid structure of Vahrenhold and Hinrichs using both real-world and artificial (worst-case) datasets. In their original experimental evaluation, Vahrenhold and Hinrichs [86] used hydrology and road feature data extracted from the U.S. Geological Survey Digital Line Graph dataset [85]. On similar "nicely" distributed sets of relatively short segments, our structure answers queries in a similar number of I/Os as the grid structure but requires about twice as much space. On less "nice" data, our structure performs significantly better than the grid structure; we present one example where our structure answers queries using 90% fewer I/Os and requires 94% less space.

5.2 Ray-shooting using persistent B-trees

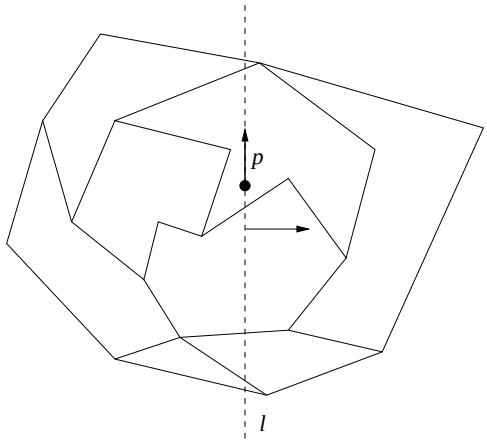


Figure 5.1: Vertical ray-shooting using sweep and persistent search tree.

Our structure for answering vertical ray-shooting queries among a set of non-intersecting segments in the plane is based on the persistent search tree idea of Sarnak and Tarjan [76]. This idea utilizes the fact that any vertical line l in the plane naturally introduces an "above-below" order on the segments it intersects. This means that if we conceptually sweep the plane from left to right ($-\infty$ to ∞) with a vertical line, inserting a segment in a persistent search tree when its left endpoint is encountered and deleting it when its right endpoint is encountered, we can answer a ray-shooting query $p = (x, y)$ by searching for the segment directly above y in the version of the search tree we had when l was at x. Refer to Figure 5.1. Note that two segments that cannot be intersected with the same vertical line are not "above-below" comparable. This means that not all elements (segments) stored in the persistent structure over its lifespan are comparable and thus an I/O-efficient structure cannot directly be obtained using a persistent B-tree. To make the structure I/O-efficient, we need a persistent B-tree that only requires elements present in the same version of the structure to be comparable. In Section 5.2.1, we first describe the persistent B-tree of [29, 88] and then in Section 5.2.2 we describe the modifications needed to use the tree in a vertical ray-shooting structure.

5.2.1 Persistent B-tree

A B-tree, or more generally an (a, b)-tree [56], is a balanced search tree with all leaves on the same level, and with all internal nodes except possibly the root having $\Theta(B)$ children (typically between $B/2$ and B). Normally, elements are stored in the leaves and the internal nodes contain "routing elements" used to guide searches (sometimes called a B$^+$-tree). Since a node or leaf contains $\Theta(B)$ elements it can be stored in $O(1)$ blocks, which in turn means that the tree uses linear space ($O(N/B)$ disk blocks). Since the tree has height $O(\log_B N)$, a range search can be performed in $O(\log_B N + T/B)$ I/Os. Insertions and deletions can be performed in $O(\log_B N)$ I/Os using *split*, *merge*, and *share* operations on the nodes on a root-leaf path [56, 28, 35].

A persistent (or *multiversion*) B-tree as described in [29, 88] is a directed acyclic graph (DAG) with elements in the sinks (leaves) and "routing elements" in internal nodes. Each element is augmented with an insert and a delete *version* (or *time*), defining the *existence interval* of the element. An element is *alive* at time t (version t) if t is in the element's existence interval and *dead* otherwise. Similarly, an existence interval is associated with each node, and it is required that the nodes and elements alive at any time t (version t) form a B-tree with fanout between αB and B for some constant $0 < \alpha < 1/2$. Given the appropriate root (in-degree 0 node), we can thus perform a range search in any version of the structure in $O(\log_B N + T/B)$ I/Os. To be able to find the appropriate root at time t in $O(\log_B N)$ I/Os, the roots are stored in a standard B-tree, called the *root B-tree*.

An update in the current version of a persistent B-tree (and thus the creation of a new version) may require structural changes and creation of new nodes. To control these changes and obtain linear space use, an additional invariant is imposed on the structure; whenever a new node is created, it must contain between $(\alpha + \gamma)B$ and $(1 - \gamma)B$ alive elements (and no dead elements). For reasons that will become clear shortly, we require that $\gamma > 2/B$, $\alpha - \gamma \geq 1/B$, and $2\alpha + 3\gamma \leq 1 - 3/B$. Note that the second constraint implies that $\alpha > \gamma$, and that the last implies that $\lceil B/2 \rceil < (1 - \gamma)B$, $\lfloor \frac{1}{2}(1 - \gamma)B \rfloor > (\alpha + \gamma)B$, and $\lceil \frac{1}{2}((\alpha + \gamma)B + B) \rceil < (1 - \gamma)B$.

To insert a new element x in the current version of a persistent B-tree we first perform a search for the relevant leaf l using $O(\log_B N)$ I/Os. Then we insert x in l. If l now contains more than B elements, we have a *block overflow*. In this case we perform a *version-split*; we copy all, say k, alive elements from l and mark l as deleted, i.e, we update its existence interval to end at the current time. Depending on the number, k, of alive elements, we create one or two new leaves and recursively update the parent of l. If $(\alpha + \gamma)B \leq k \leq (1 - \gamma)B$, we create one new leaf with the k elements and recursively update *parent*(l) by persistently deleting the reference to l and inserting a reference to the new node. If on the other hand $k < (\alpha + \gamma)B$ or $k > (1 - \gamma)B$, we have a *strong*

underflow or *strong overflow*, respectively, violating our additional invariant. The strong overflow case is handled using a *split*; we simply create two new nodes with approximately half of the k elements each, and update $parent(l)$ recursively in the appropriate way. The two new nodes both contains at most $\lceil \frac{B}{2} \rceil < (1-\gamma)B$ elements and at least $\lfloor \frac{1}{2}(1-\gamma)B \rfloor > (\alpha+\gamma)B$ elements. Like a strong overflow is handled with a split similar to a normal B-tree, the strong underflow case is handled with operations similar to merge and share rebalancing operations on normal B-trees; we perform a version-split on a sibling l' of l to obtain k' ($\alpha B \leq k' \leq B$) other alive elements. Since $k + k' \geq 2\alpha B$ and $\alpha > \gamma$ we have more than $(\alpha+\gamma)B$ elements. If $k + k' \leq (1-\gamma)B$, we simply create a new leaf with the $k+k'$ elements. If on the other hand $k+k' > (1-\gamma)B$, we perform a split in order to create two new leaves. The two new nodes both contain at least $\lfloor \frac{1}{2}(1-\gamma)B \rfloor > (\alpha+\gamma)B$ elements and at most $\lceil \frac{1}{2}((\alpha+\gamma)B+B) \rceil < (1-\gamma)B$ elements. The first case corresponds to a *merge* and the second to a *share*. Finally, we recursively update $parent(l)$ appropriately.

A deletion is handled similarly to an insertion; first the relevant element x in a leaf l is found and marked as deleted. This may result in l containing $k < \alpha B$ alive elements, violating the invariant that the live elements form a B-tree with fanout between αB and B. We call this condition a *block underflow*. Note that a node with a block underflow also has a strong underflow since $k < (\alpha+\gamma)B$. To reestablish the invariants, we perform a merge or share as previously; we first perform a version-split on a sibling node to obtain a total of $k + k' \geq 2\alpha B - 1$ elements, which is at least $(\alpha+\gamma)B$ by the requirement $\alpha - \gamma \geq 1/B$. We then either create a new leaf with the obtained elements, or split them and create two new leaves precisely as previously. Once we have performed the merge or share, we recursively update the parent of l. Figure 5.2 illustrates the "rebalance operations" needed as a result of an insertion or a deletion.

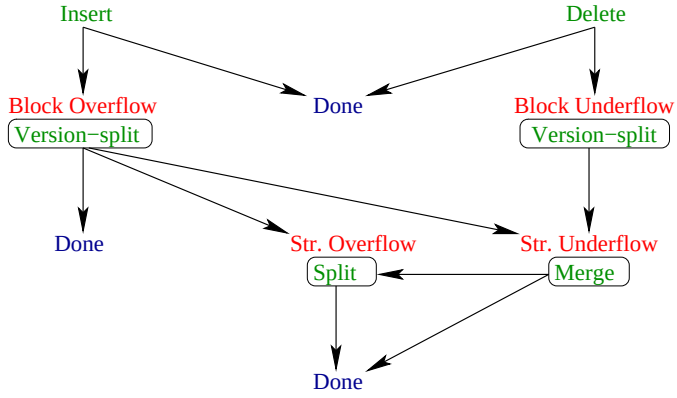


Figure 5.2: Illustration of "rebalancing operations" needed when updating a persistent B-tree.

Both insertions and deletions are performed in $O(\log_B N)$ I/Os, since the changes (rebalancing) needed on any level of the tree can be performed in $O(1)$ I/Os, and since the rebalancing at most propagates from l to the root of the current version. To see that a persistent B-tree uses $O(N/B)$ blocks after N updates, first note that a rebalance operation on a leaf creates at most two new leaves. Once a new leaf l is created, at least γB updates are performed on l before a rebalance operation is needed on it again. Thus at most $2\frac{N}{\gamma B} = O(N/B)$ leaves are created during N updates. Each time a leaf is created or marked dead, a corresponding insertion or deletion is performed recursively one level up the tree. Since at most $4\frac{N}{\gamma B}$ leaves are created or deleted during the N operations, it follows by the same argument as above that the number of nodes created one level up the tree is bounded by $2^2\frac{N}{(\gamma B)^2}$. By induction, the number of nodes created h levels up the tree is bounded by $2^{h+1}\frac{N}{(\gamma B)^{h+1}}$. The total number of nodes (blocks used) over N updates is therefore bounded by $\frac{2N}{\gamma B}\sum_{h=0}^{\log_B N}(2/\gamma B)^h$, which is $O(\frac{N}{B})$ since $\gamma B > 2$.

Theorem 2 *([29, 88]) After N insertions and deletions of elements with a total order into an initially empty persistent B-tree, the structure uses $O(N/B)$ space and supports range queries in any version in $O(\log_B N + T/B)$ I/Os. An update can be performed on the newest version in $O(\log_B N)$ I/Os.*

5.2.2 Modified Persistent B-tree

In the persistent B-tree as described above, elements were stored in the leaves only. To search efficiently, internal nodes also contain elements ("routing elements"). In our discussion of the persistent B-tree so far we did not discuss precisely how these elements are obtained, that is, we did not discuss what element is inserted in $parent(v)$ when a new node (or leaf) v is created and a new reference is inserted in $parent(v)$. As in standard, non-persistent B-trees, the persistent B-trees described above use a *copy* of the maximal element in v as a routing element in $parent(v)$ when v is created. Therefore, when an element is deleted, that is, marked as dead in a leaf, live copies of the element can still exist as routing elements in the internal nodes. Thus, even though when searching for a given element e at time t we only compare e with elements considered alive at time t, we may compare e to an internal routing copy of an element long after the original leaf element is marked dead. If not all elements stored in the persistent B-tree during its entire lifespan are comparable, that is, if they are not totally ordered, we cannot evaluate the comparisons needed to perform the search. In the vertical ray shooting problem, segments are only partially ordered (only segments intersecting a give vertical line can be compared) and therefore the standard persistent B-tree cannot be used to design an I/O-efficient vertical ray shooting structure.

To obtain an I/O-efficient vertical ray-shooting structure, we construct a persistent B-tree structure that only requires elements present in the same version to be comparable. To do so, we modify the existing persistent B-tree to store actual data elements in both leaf nodes and internal nodes, and impose the new invariant that the alive elements at any time t form a B-tree with data elements in internal as well as leaf nodes. Thus, the main difference between our modified structure and the previous structure is that at any given time t, at most one live copy of an element exists in the structure; we do not create multiple live copies of the same element to use as routing elements. Except for slight modifications to the version-split, split, merge, and share operations, the insert algorithm remains unchanged. In the delete algorithm we need to be careful when deleting an element x in an internal node u. Since x is the only live copy, we must mark it as dead, but x is also associated with a reference to a child u_c of u that is still alive. Therefore, instead of immediately marking x dead, we first find x's predecessor y in a leaf below u and persistently delete it. Then we delete x from u, insert a live copy of y with a reference to the child u_c, and perform the relevant rebalancing. What remains is to describe the modifications to the rebalance operations.

Version-split

Recall that a version-split (not leading to a strong underflow or overflow) consists of copying all alive elements in a node u, using them to create a new node v, deleting the reference to u, and recursively inserting a reference to v in $parent(u)$. Since the reference to u has an element x associated with it, we cannot simply mark it deleted by updating its existence interval. However, since we are also inserting a reference to the new node v, and since the elements in v are a subset of the elements in u, we can use x as the element associated with the reference to v. Thus we can perform the version-split almost exactly as before, while maintaining the new invariant, by simply using x as the element associated with the reference to v as shown in Figure 5.3.

Split

When a strong overflow occurs after a version-split of u, a split is needed; two new nodes v and v' are created and two references inserted in $parent(u)$. As in the version-split case, the element

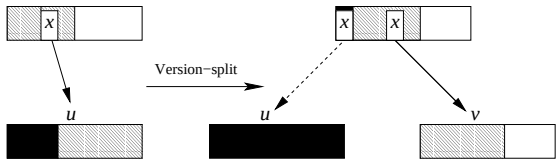


Figure 5.3: Illustration of a version-split. Partially shaded regions correspond to alive elements, black to dead elements, and white regions to unused space. (Individual elements use this same shading convention to indicate their status.)

x associated with u in $parent(u)$ can be used as the element associated with the reference to v'. To maintain the new invariant we then "promote" the maximal element y in v to be used as the element associated with the reference to v in $parent(u)$, that is, instead of storing y in v, we store it in $parent(u)$. Refer to Figure 5.4. Note that, because of the promotion of y, the new node v has one less element than it would have had using the split procedure described in the previous section. However, the previous space arguments still applies since $\Omega(\gamma B)$ updates are still required on v before further structural changes are needed. Otherwise a split remains unchanged.

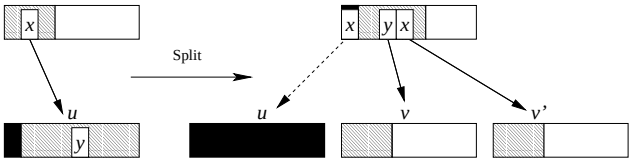


Figure 5.4: Illustration of a split.

Merge

When a strong underflow occurs after a version-split of u, we perform a version-split of u's sibling u' and create a new node v with the obtained elements. We then delete the references to u and u' in $parent(u)$ by marking the two elements x and y associated with the references to u and u' as deleted. We can reuse the maximal of these elements, say y, as the reference to the new node v. To maintain the new invariant (preserve all elements) we then "demote" x and store it in the new node v. Otherwise a merge remains unchanged. Refer to Figure 5.5. The demotion of x leaves the new node v with one more element than it would have had using the merge procedure described in the previous section. However, as in the previous case, $\Omega(\gamma B)$ updates are still needed on v before further structural changes are needed.

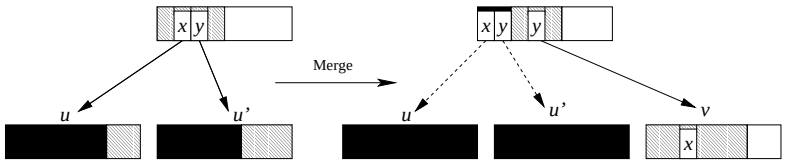


Figure 5.5: Illustration of a merge.

Share

When a merge would result in a new node with a strong overflow, we instead perform a share; we first perform a version-split on the two sibling nodes u and u' and create two new nodes v and v'

with the obtained elements. As in the merge case, we then delete the references to u and u' in $parent(u)$ by marking two elements x and y associated with u and u' as deleted. We can reuse the maximal element y as the reference to v' but x cannot be used as a reference to v. Instead, we demote x to v and promote the maximal element z in v to $parent(u)$. Refer to Figure 5.6. Since we have both a demotion and promotion in this case, the number of elements in the new node v is identical to the number of elements we would have if we used the share procedure described in the previous section.

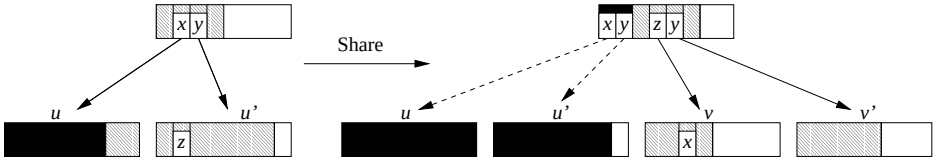


Figure 5.6: Illustration of a share.

Theorem 3 *After N updates on an initially empty modified persistent B-tree, the structure uses $O(N/B)$ space and supports range queries in any version in $O(\log_B N + T/B)$ I/Os. An update can be performed on the newest version in $O(\log_B N)$ I/Os.*

Corollary 1 *A set of N non-intersecting segments in the plane can be pre-processed into a data structure of size $O(N/B)$ in $O(N \log_B N)$ I/Os such that a vertical ray-shooting query can be answered in $O(\log_B N)$ I/Os.*

While trivially performing a sequence of N given updates on a (modified as well as unmodified) persistent B-tree takes $O(N \log_B N)$ I/Os, it has been shown how the N updates can be performed in $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os (the sorting bound) on a normal (unmodified) persistent B-tree [87, 10, 19]. In the modified B-tree case, the lack of a total order seems to prevent us from performing the updates in this much smaller number of I/Os. It remains a challenging open problem to construct the ray-shooting structure (the modified persistent B-tree) in $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os. Such a fast algorithm would immediately lead to a semi-dynamic (insert-efficient) vertical ray-shooting structure using the external version of the logarithmic method [25].

5.3 Experimental results

In this section we describe the results of an extensive set of experiments designed to evaluate the performance of the (modified) persistent B-tree when used to answer vertical ray-shooting queries, compared to the performance of the grid structure of Vahrenhold and Hinrichs [86].

5.3.1 Implementations

Both the persistent B-tree [16, 17] and the grid structure [86] were implemented using TPIE [11, 22]. Below we discuss the two implementations separately.

Persistent B-tree Implementation

When using the persistent B-tree to answer vertical ray-shooting queries the elements (segments) already implicitly contain their existence interval (the x-coordinates of the endpoints). Thus we

implemented the structure without explicitly storing existence intervals.[2] This way each element occupied 28 bytes. To implement the root B-tree we used the standard B-tree implementation in the TPIE distribution [22]. In this implementation each element occupies 16 bytes.

Two parameters α and γ are used in the definition of the persistent B-tree. Since we are working with large datasets, we choose these parameters to optimize for space. Space is minimized when γ (and thus the number of updates needed between creation of nodes) is maximized, and the constraints on α and γ require that we choose $\gamma \leq \min\{\alpha - 1/B, 1/3 - 2/3\alpha - 1/B\}$. The maximal value of γ is when $\alpha = \frac{1}{5}$, so we chose $\alpha = \frac{1}{5}$ and $\gamma = \frac{1}{5} - \frac{1}{B}$ accordingly. If we wanted to optimize for query performance, we should choose a larger value of α—leading to a smaller tree height—but a few initial experiments indicated that increasing α had relatively little impact on query performance.

Grid structure Implementation

The idea in the structure of Vahrenhold and Hinrichs [86] is to impose a grid on (the minimal bounding box of) the segments and store each segment in a bucket corresponding to each grid cell it intersects. The grid is designed to have N/B cells and ideally each bucket is of size B. To adapt to the distribution of segments, a slightly different grid than the natural $\sqrt{N/B} \times \sqrt{N/B}$ grid is used; for a bounding box of width x_d and height y_d, two parameters, $F_x = \frac{1}{N \cdot x_d} \sum_{i=1}^{N} |x_{i2} - x_{i1}|$ and $F_y = \frac{1}{N \cdot y_d} \sum_{i=1}^{N} |y_{i2} - y_{i1}|$, where the i'th segment is given by $((x_{i1}, y_{i1}), (x_{i2}, y_{i2}))$, are used to estimate the amount of overlap of the x-projections and y-projections of the segments, respectively. Then the number of rows and columns in the grid is calculated as $N_x = \alpha_x \sqrt{N/B}$ and $N_y = \alpha_y \sqrt{N/B}$, where $\alpha_x/\alpha_y = F_y/F_x$ and $\alpha_x \alpha_y = 1$.

In the TPIE implementation of the grid structure [86], the segments are first scanned to compute N_x and N_y. Then they are scanned again and a copy of each segment is made for each cell it crosses. Each segment copy is also augmented with a bucket ID, such that each copy occupies 32 bytes. Finally, the new set of segments is sorted by bucket ID using TPIE's I/O-optimal merge sort algorithm [7, 11]. The buckets are stored sequentially on disk and a *bucket index* is constructed in order to be able to locate the position of the segments in a given bucket efficiently. The index is simply a $N_x \times N_y$ array with entry (i, j) containing the position on disk of the first segment in the bucket corresponding to cell (i, j) (as well as the number of segments in the bucket). Each index entry is 16 bytes.

If L is the the number of segment copies produced during the grid structure construction, $O(\frac{L}{B} \log_{M/B} \frac{L}{B})$ I/Os is the number of I/Os used by the algorithm. Ideally, this would be $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os, but in the worst case each segment can cross $\sqrt{N/B}$ buckets and the algorithm requires $O(\frac{N}{B} \sqrt{N/B} \log_{M/B} \frac{N}{B})$ I/Os. Refer to Figure 5.7 for an example of such a dataset.

Answering a query using the grid method simply involves looking up the position of the relevant bucket (the cell containing the query point) using the bucket index and then scanning the segments in the bucket to answer the query. In the ideal case each bucket contains $O(B)$ segments, and the query can be performed in $O(1)$ I/Os. However, in the worst case, a query takes $O(N/B)$ I/Os.

5.3.2 Data

To investigate the efficiency of our vertical ray-shooting data structure, we used road data from the US Census TIGER/Line dataset containing all roads in the United States [83]. In this dataset, one curved road is represented as a series of short linear segments. Roads (segments) are also broken at intersections, such that no two segments intersect other than at endpoints. Because of various errors, the dataset actually contains a few intersection segments, which we removed in a preprocessing step.

[2] We have also implemented a general persistent B-tree with existence intervals. Both implementations will be made available in TPIE.

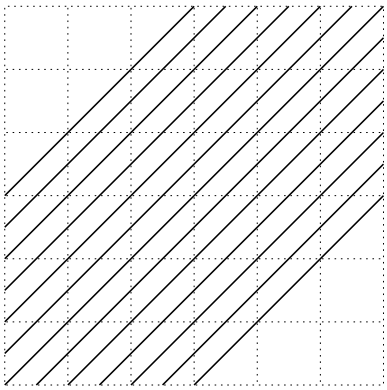


Figure 5.7: Worst-case dataset for the grid method.

Our first 6 datasets, containing between 16 million segments (374 MB) and 80 million segments (1852 MB), consist of the roads in connected parts of the US (corresponding to the six CD's on which the data is distributed). A summary of the number of segments in each dataset is given in Table 5.1, and images of the datasets appear in Figure 5.8. In addition to these large datasets, we also used four smaller datasets with disjoint data regions; dataset CL1 consists of the four US states Washington, Minnesota, Maine and Florida, and CL2 excludes Minnesota. The bounding boxes of CL1 and CL2 are relatively empty with the exception of three or four "hot spots" where the states are located. These datasets were included to investigate the effect of non-uniform data distributions. The dataset DST spans a sparsely populated region of the United States and was included to investigate the effect of longer but more uniformly distributed segments (we expect that the segments are longer and more uniformly distributed in the desert than in metropolitan areas). The dataset ATL, on the other hand, was chosen as a dense, but maybe less uniform, dataset. Finally, in addition to the real world data, we also generated a dataset LONG, corresponding to Figure 5.7, designed to illustrate the worst-case behavior of the grid method.

For query point sets, we generated a list of 100000 randomly sampled points from inside each of the datasets; for each query point p, we made sure that segments are hit by vertical rays emanating from p in both the positive and negative y-directions.

5.3.3 Experimental Setup

We ran our experiments on an Intel PIII - 500MHz machine running FreeBSD 4.5. All code and data resided on a local 10000 RPM 36GB SCSI disk. The disk block size was 8KB, so each leaf of the persistent B-tree could store 290 elements and each internal node (also containing references) 225 elements. In the root B-tree, each leaf could hold 510 elements and each internal node 681 elements. The grid method bucket index had $N/256$ entries of 16 bytes each.

We were interested in investigating the I/O performance when the data sets are much larger than the available internal memory. Because extremely large data sets are not readily available and would prohibit extensive testing, we limited the RAM of the machine to 128 MB and, since the OS was observed to use 60 MB, we limited the amount of memory available to TPIE to 12 MB. Constraining the memory also provides insight into how the structures would perform if the system had more total memory but was under heavy load.

TPIE is flexible with regards to how I/Os are actually performed; for implementations where scanning and reading large contiguous portions of the data are common, it is often preferable to use the UNIX `read()` and `write()` system calls to take advantage of OS features such as prefetching. For data structure implementations that perform random I/O, memory-mapped I/O is often

preferable. Therefore we implemented the grid method using streams and `read()`/`write()` system calls exclusively, while we implemented the persistent B-tree using memory-mapped I/O (except that `read()`/`write()` system calls were used to process sweep events during construction).

To increase realism in our experiments, we used TPIE's built-in (8-way set associative LRU-style) cache mechanism to cache parts of the data structures. Since a query on the persistent B-tree always begins with a search in the relatively small root B-tree, we used a 72 block cache (8 internal nodes and 64 leaf nodes) for the root B-tree—enough to cache the entire structure in all our experiments. We also used a separate 16 block cache for the internal persistent B-tree nodes and a 32 block cache for the leaf nodes. Separate caches were used for internal nodes and leaves to ensure that accesses to the many leaves did not result in eviction of the few internal nodes from the cache. In total, the caches used for the entire persistent structure were of size 120 blocks or 960KB. For the grid structure, we (in analogy with the root B-tree in the persistent case) cached the entire bucket index—of size 2.41 MB for the largest dataset (D1-6).

5.3.4 Experimental Results

Structure Size

Figure 5.9 shows the size of the grid and persistent B-tree data structures constructed on the 11 datasets. For the real life (TIGER) data, the grid method uses about 1.4 times the space of the raw data, whereas the persistent B-tree sometimes uses almost 3 times the raw data size. The low overhead of the grid method is a result of relatively low segment duplication (around 1.02 average copies per segment) due to the very short segments. The rest of the space is used to store the bucket index. The larger overhead of the persistent B-tree is mainly due to structural changes (rebalance operations) resulting in the creation of multiple copies of each segment; we found that there are roughly 2.4 copies of each segment in a persistent B-tree structure.

Analyzing the numbers for the real datasets in more detail reveals that for the first six datasets and ATL the space utilization is quite uniform. For datasets DST, CL1, and CL2 the grid structure uses slightly less space while the persistent B-tree uses more space. The persistent B-tree method uses more space because the relatively small datasets are sparsely populated with segments; at any given time during the construction sweep, the sweep line intersects relatively few segments. As a result, many transitions are made between a height one (one leaf) and height two (with a low-degree root) tree, resulting in relatively low block utilization.

For the artificially generated dataset LONG, the space usage of both structures increases dramatically. As expected, the cause of the enormous space use of the grid structure is a high number of segment copies (93 per segment on the average). The persistent B-tree also has a significant increase in space, but not nearly as much as the grid structure—the structure is 94% smaller than the grid structure. The reason for increased space usage in the persistent B-tree is that all the segments are long and thus they stay in the persistent structure for a long time (in many versions), resulting in a high tree. Furthermore, most of the structural changes in the beginning of the construction algorithm are splits, leading to many copies of alive elements. Similarly, most of the structural changes at the end of the execution are merges, again leading to a large redundancy.

Construction Efficiency

Figure 5.10 shows construction results in terms of I/O and physical time. The numerical data used for the graphs can be found in Table 5.2 in the Appendix. For all the real-world datasets the persistent B-tree structure uses around 1.5 times more I/Os than the grid structure. This is rather surprising since the theoretical construction bound for the persistent B-tree is $O(N \log_B N)$ I/Os, compared to the (good case) $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ bound for the grid structure. Theory thus predicts that the tree construction algorithm should perform about B times as many I/Os as the grid method.

The reason for this discrepancy between theory and practice is that during the construction sweep the average number of segments intersecting the sweep line is relatively small. For the D1-6 dataset, it is less than 2500 segments (Refer to Figure 5.11). Thus the size of the persistent B-tree accessed during each update is small and as a result the caches can store most of the nodes in the tree.

While it only takes about 50% more I/Os to construct the persistent B-tree structure than to construct the grid structure, it takes nearly 17 times more physical time. One reason for this is that most of the I/Os performed by the grid construction algorithm are sequential, while the I/Os performed by the persistent B-tree algorithm are more random. Thus the grid algorithm takes advantage of the optimization of the disk controller and the OS file system for sequential I/O. Another reason is that construction of the persistent B-tree structure is more computationally intensive than construction of the grid. Our trace logs show that over 95% of the construction time for the persistent B-tree is spent in internal memory compared to only 50% for the grid method.

While the grid construction algorithm outperforms the persistent B-tree algorithm on the real-life datasets the worst-case dataset, LONG, causes it significant problems. For this dataset, the grid construction takes 48 minutes compared to 53 minutes for the 80 times bigger dataset D1-6, and compared to 10 minutes for the persistent B-tree. The reason is that the large size of the structure results in a high I/O construction cost. The persistent B-tree construction also takes relatively more I/Os (and time), mainly due to the high average number of segments intersecting the sweep-line (500000 at the peak).

Query Efficiency

Our query experiments illustrate the advantage of the persistent B-tree structure over the grid structure; Figure 5.12 shows that a query is consistently answered in less than two I/Os on the average, while the grid structure uses between approximately 2.6 and 28 I/Os on the average for the real-world datasets, and 126 I/Os for the LONG dataset. The numerical data used for the graphs can be found in Table 5.3 in the Appendix.

Analyzing the grid structure results for the real-world datasets in more detail reveals that the performance is mostly a function of the distribution of segments within their bounding box. The bounding boxes become more full as we move from D1 to D1-6, and as a result the average number of I/Os per query drops from 3.76 to 2.65. The dataset ATL overlaps with a significant portion of D1, but because of the better distribution of segments within the bounding box the I/O performance is better for ATL. Datasets CL1 and CL2 exacerbate the problem with non-uniform distributions; for these datasets, most grid cells, more than 90%, are completely empty. As a result, a query within a non-empty cell is very expensive. Finally, as expected the LONG data set shows the grid structures vulnerability to long segments; on average a query takes 126 I/Os.

Analyzing the persistent B-tree results in detail reveals that its performance is mostly a function of the average number of segments intersecting the sweepline; datasets D1 through D1-6 and ATL have higher average I/O cost than DST, CL1 and CL2. Similarly, dataset D1 has a lower average cost than ATL, since the region of D1 not in ATL has a smaller number of sweepline-segment intersections. This is the opposite of the behavior of the grid method whose query performance is worse for D1 than ATL. Finally, even though the average number of sweepline-segment intersections for the LONG dataset is more than half a million, the height of the tree is no more than three at any time (version), and as a result the query efficiency is maintained.

Further Experiments

In our experiments with the TIGER data we noticed that non-empty buckets in the grid structure often occupied three or more disk blocks of segments. Therefore we expected that by reducing the grid spacing in both the x and y-direction by a factor of two—creating four times as many buckets—each bucket would likely occupy only one disk block, leading to an improved query performance.

Such an improvement is of course highly data dependent. It would also come at the cost of space, since we must index four times as many buckets, and the denser grid may result in more segment duplication. To investigate this we ran our tests again using such a modified grid; we found that for the real-life datasets the number of segment copies did not increase significantly (from 1.02 to 1.04 copies per segment). Thus the construction performance was maintained. In terms of query performance, we found that the four-fold increase in the number of buckets leads to a factor of two to three improvement in the query performance. Refer to Figure 5.13. In these experiments we cached the entire bucket index, which is four times larger than in the regular grid method. For D1 the index is 2 MB, and for D1-6 the index is 10 MB, ten times larger than the cache used in the persistent B-tree. For the LONG dataset, the modified grid uses twice the space of the standard grid and still has poor query performance compared to the persistent B-tree.

Finally, to investigate the influence of caches we ran a series of experiments without caching. In these experiments we found that one additional I/O is used per query in the grid structure in order to access the bucket index. In the persistent B-tree structure one or two extra I/Os are used depending on the height of the root B-tree. This can immediately be reduced by one I/O per query by just caching the block containing the root of the root B-tree. Thus we conclude that the query performance does not depend critically on the existence of caches.

5.4 Summary

In this chapter, we have presented an external point location data structure based on a persistent B-tree that is efficient both in theory and practice. One major open problem is to construct the structure in $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os, as compared to the (trivial) $O(N \log_B N)$ algorithm discussed here.

Data Set	Segments (in Millions)	Size (MB)
D1	16.36	374
D1-2	31.22	714
D1-3	41.78	956
D1-4	57.33	1312
D1-5	69.82	1598
D1-6	80.91	1852
CL1	6.69	153
CL2	5.09	116
ATL	10.84	248
DST	6.40	146
LONG	1.00	23

Table 5.1: Number of segments and raw dataset size (assuming 24 bytes per segment) of the experimental datasets.

Data Set	Relative Size		I/Os (Thousands)		Time (Minutes)	
	Grid	Tree	Grid	Tree	Grid	Tree
D1	1.377	2.669	334	513	10	174
D1-2	1.370	2.668	637	977	20	341
D1-3	1.371	2.660	854	1309	27	436
D1-4	1.374	2.673	1174	1798	38	607
D1-5	1.374	2.662	1430	2187	46	727
D1-6	1.374	2.657	1657	2533	53	892
CL1	1.354	2.816	129	203	3.8	63
CL2	1.352	2.834	103	162	3.0	51
ATL	1.377	2.684	223	331	6.7	116
DST	1.368	2.902	136	214	4.2	68
LONG	124.013	7.168	1816	58	48	10

Table 5.2: Data structure size, construction I/Os, and construction time for grid and persistent B-tree. See also Figures 5.9 and 5.10.

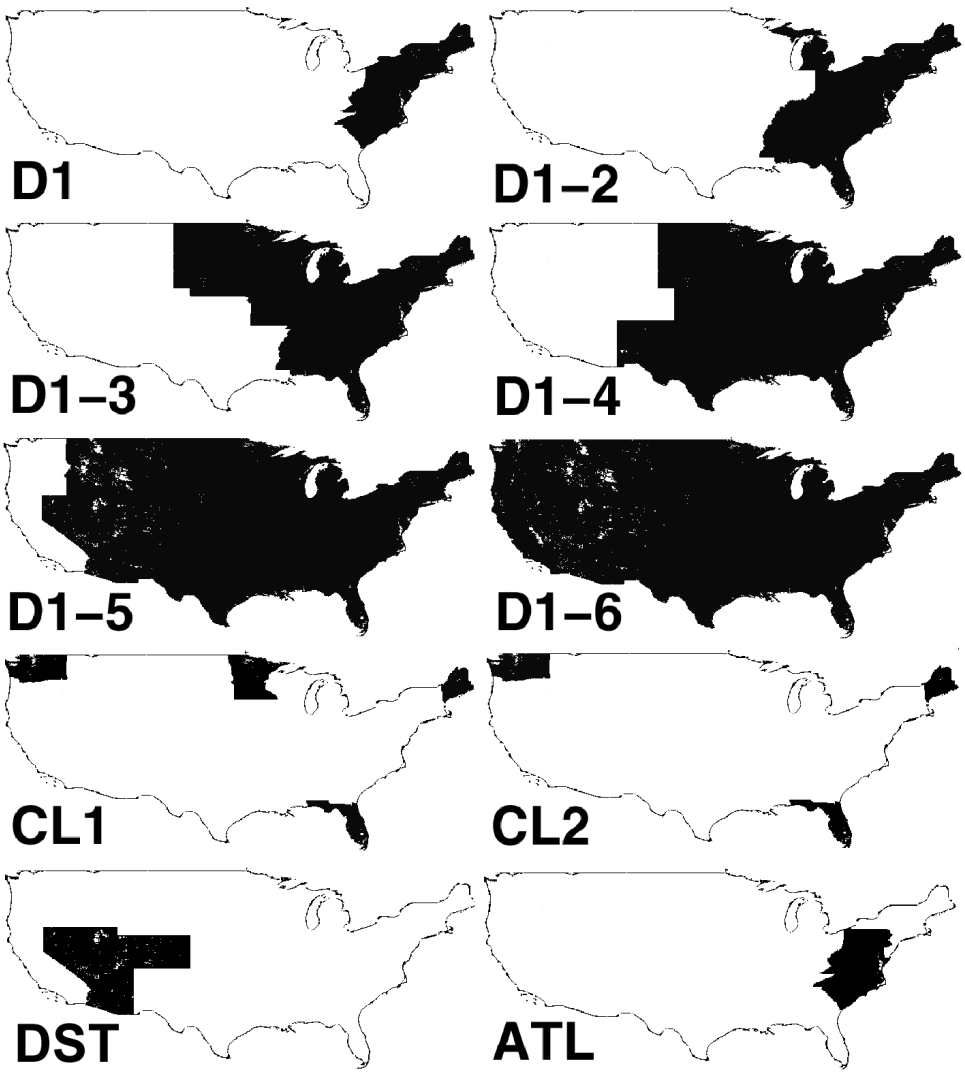


Figure 5.8: Illustration of the real-world datasets used in our experiments. Black regions indicate areas in the dataset. The outline of the continental US serves as a visualization guide and is not part of the actual data set.

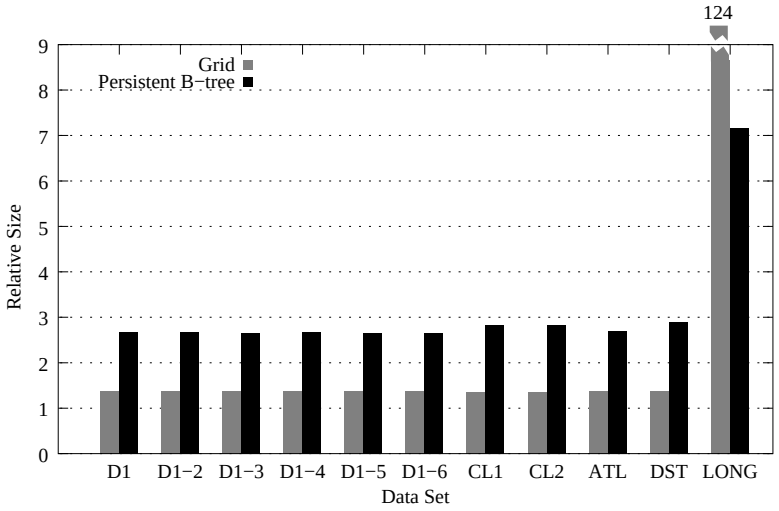


Figure 5.9: Space utilization of grid and persistent B-tree based structures. Size is relative to the raw dataset size.

Data Set	I/Os per Query			Time per Query (ms)		
	Grid	1/2 Grid	Tree	Grid	1/2 Grid	Tree
D1	3.76	1.70	1.86	7.22	5.60	5.07
D1-2	3.26	1.58	1.94	6.61	5.20	5.08
D1-3	3.39	1.62	1.96	8.41	6.93	5.86
D1-4	2.84	1.47	1.97	8.40	7.49	7.14
D1-5	2.70	1.44	1.96	8.25	7.43	7.17
D1-6	2.65	1.43	1.98	8.62	7.76	7.72
CL1	18.55	5.50	1.51	18.31	7.24	3.00
CL2	28.40	8.16	1.41	24.81	9.75	3.40
ATL	2.74	1.45	1.91	5.78	4.81	3.44
DST	2.65	1.42	1.58	4.32	3.20	3.77
LONG	126.00	64.22	1.74	1067.11	506.11	0.77

Table 5.3: Query performance of grid, half-grid, and persistent B-tree structures. See also Figures 5.12 and 5.13.

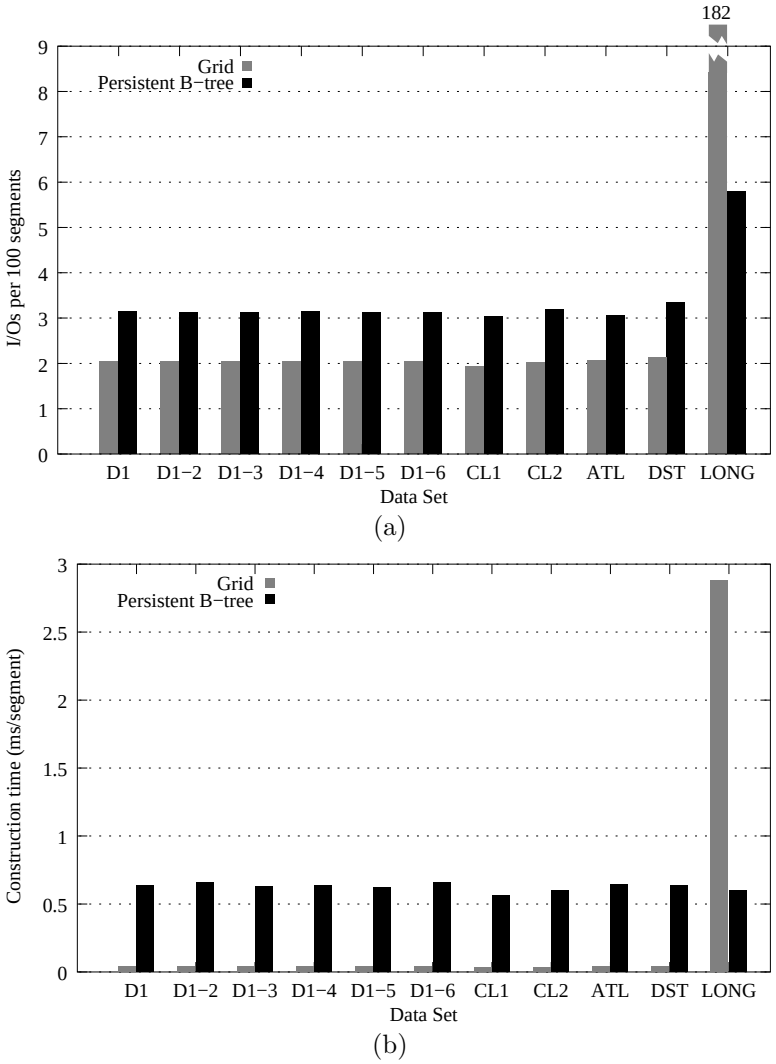


Figure 5.10: Construction performance: (a) Number of I/Os per 100 segments. (b) Construction time per segment.

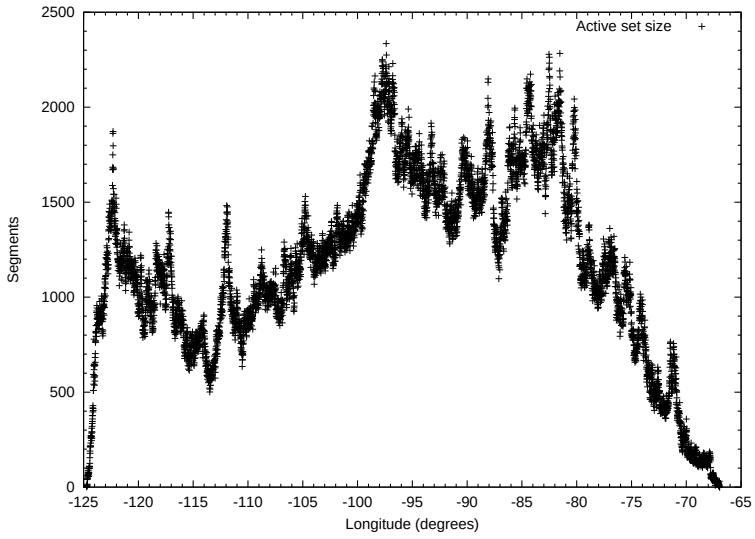
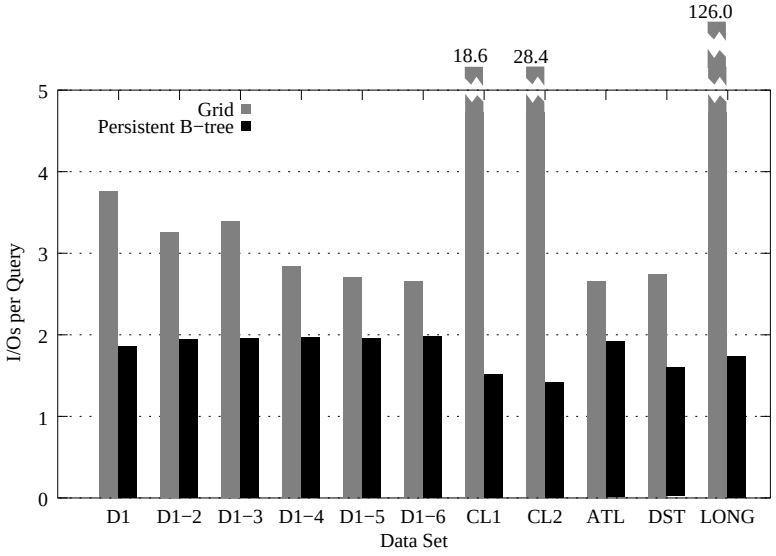
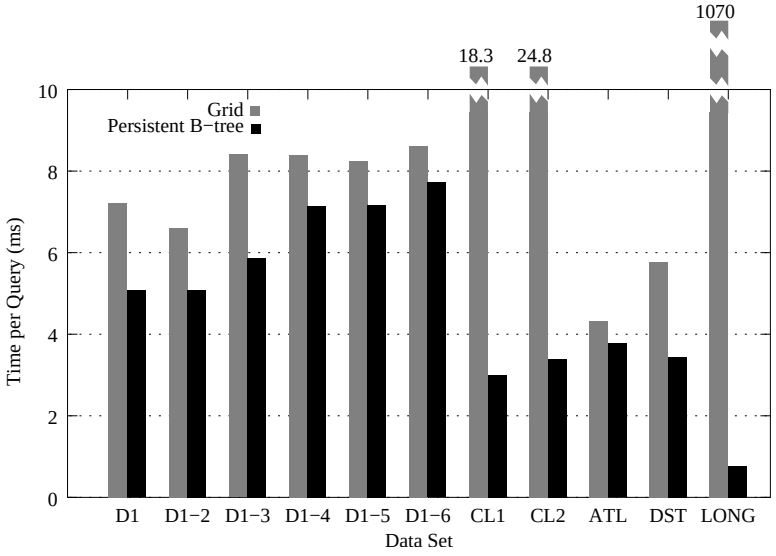


Figure 5.11: The number of segments intersecting the sweep line as a function of sweep line position during the construction of the persistent B-tree for the D1-6 dataset.

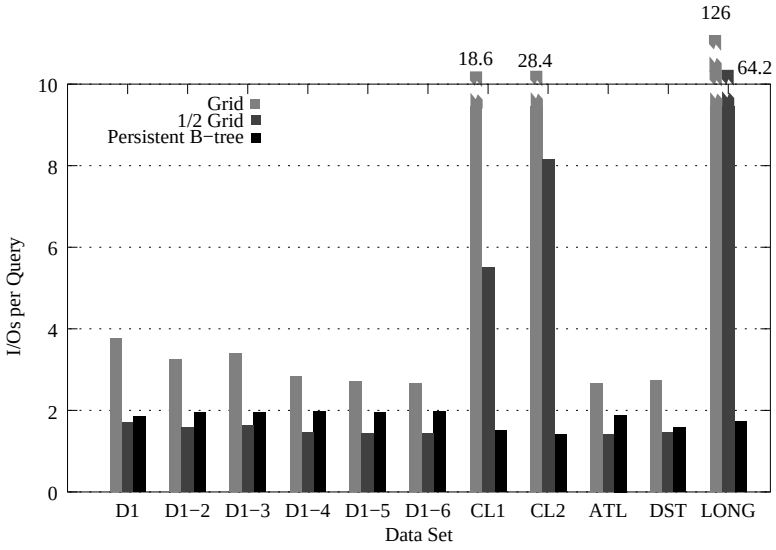


(a)

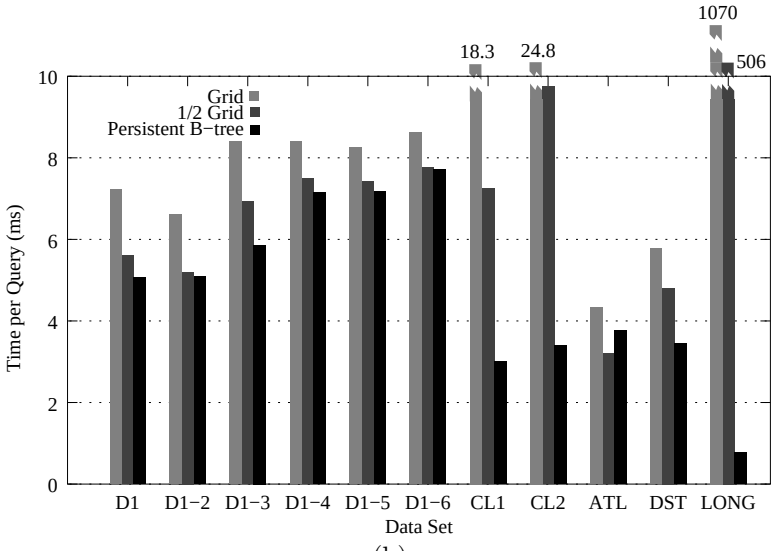


(b)

Figure 5.12: Query performance: (a) Number of I/Os per query. (b) Time per query in milliseconds.



(a)



(b)

Figure 5.13: Query performance of the grid method using four times as many buckets (1/2 Grid): (a) Number of I/Os per query. (b) Time per query in milliseconds.

Chapter 6

A Geoprocessing Pipeline

6.1 Introduction

The previous chapters discussed several algorithmic challenges in GIS for processing large data sets and presented scalable solutions to the problems described. While our algorithms were designed to solve individual tasks, our solutions can be combined to form a geo-processing pipeline that takes as initial input a set of remotely sensed elevation points and produces as a final output a watershed hierarchy. This pipeline consists of four major stages shown in Figure 6.1. In the first stage, we construct a grid DEM from a set of elevation points using the scalable quad-tree construction method [2] we developed in Chapter 2. In the second stage, we remove sink from our constructed DEM using previously-known methods for topological persistence [45, 44] and flooding [14] as described in Chapter 3. The third stage extracts river networks from the terrain using the flow routing and upslope contributing area algorithms described in Sections 3.4 and 3.5. The fourth and final stage computes the Pfafstetter watershed hierarchy using the methods developed in Chapter 4.

This pipeline provides an extremely scalable and powerful tool for GIS users to analyze very large terrain data sets. Our approach has a number of key advantages over other algorithms. First, because out solutions scale to massive data sets, users do not need to worry about the size of their input data sets and do not need pre-process their data or break the data into tiles. In many pipeline stages, tiling is not even possible because we are computing a global function. A second advantage of our pipeline approach is that it is scalable from end to end. There is no bottleneck in the middle of the pipeline that limits overall scalability. A scalable watershed algorithm would be of limited use if earlier stages in the pipeline, such as grid construction and flow routing, were not scalable. A final advantage of our pipeline approach is that each stage seamlessly integrates with the other pipeline stages, while still remaining loosely coupled to other stages. The seamless integration means output from a single stage in the pipeline can be immediately used as input for the next pipeline stage without manual pre- or post-processing of intermediated data sets. Furthermore, each stage is loosely coupled in the sense that no stage depends on a particular implementation of another stage. For example, in the grid-construction stage, we could use a variety of point interpolation methods or even a segmentation scheme that uses a kd-B-tree instead of a quad-tree. None of the other stages rely on a particular interpolation method to work properly and individual pipeline stages can be

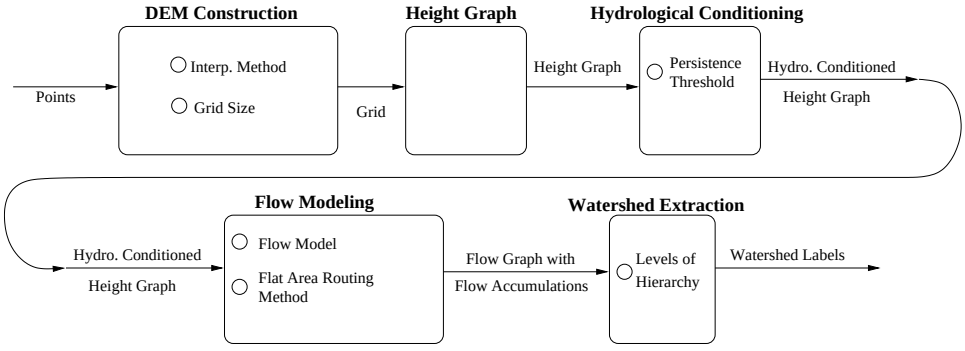


Figure 6.1: Algorithms in this thesis form a geo-processing pipeline from points to watersheds.

tuned for a particular user's needs.

In addition to being scalable our pipeline features a modular design in which additional components and models can be added easily. We expose a number of tuning parameters to the user so they can choose between a number of models and option, e.g., grid resolution, flow model, or persistence threshold. Because our approach is scalable, users can run the pipeline multiple times while changing parameters to find optimal settings for their data and their needs. In the hands of the GIS community these tools can be an extremely valuable aid in modeling and analyzing large hi-resolution terrain data sets acquired by modern mapping methods. Ultimately, we hope to distribute many of the methods described, implemented, and tested in this thesis to the GIS community for the benefit of others.

In this chapter, we demonstrate the power of our geo-processing pipeline by showing the results of a case study on the Neuse river basin in North Carolina. We describe our experimental setup, including which data, software, and hardware we used, in Section 6.2. To verify the scalability of our algorithms, we present experimental results that show our algorithms can process grids with over 1,590 million data cells that were derived from over 415 million lidar input points in Section 6.3. While the emphasis of the thesis work was scalability, we also show how we can tune various parameters in various pipeline stages. Tuning parameters in one pipeline stage can influence results in later pipeline stages. We show how tuning grid construction parameters offers a trade-off between construction computation time and DEM quality in Section 6.5. In Section 6.6, we conduct experiments on grids with different resolutions and show how different grid resolutions can result in different watershed boundaries. Finally, we show in Section 6.7 how changing the persistence threshold during the sink removal stage changes the river network and the watershed boundaries.

6.2 Experimental Setup

For our experiments, we chose the Neuse river basin of North Carolina as our case study. We chose this particular data set because it included a large set of publicly available lidar points that covered a large regional watershed. The Neuse data set is a collection of 477 million lidar points (over 20GB of raw data) publicly available for download from the North Carolina flood mapping project [69]. The data cover an area of roughly 6200 square miles with an average point spacing of approximately 20 feet. However, the point spacing is rather heterogeneous, ranging from nine feet in open areas to more than 50 feet in densely vegetated regions. Because lidar pulses are absorbed over water, there are few data points over large bodies of open water. The data have been pre-processed by the data providers to remove large amount of vegetation and many buildings from the terrain. However, many man-made features still exist, including bridges, as shown in Figure 6.2. Because this lidar data was collected for the purposes of flood mapping, some bridges like the one shown in Figure 6.3 have been cut during pre-processing to allow the flow of water under the bridge. While many bridges across major waterways have been cut, Figure 6.2 shows that this is not always the case.

6.2.1 Software and Hardware

We implemented our algorithms using the C++ programming language and the Linux operating system. We built our code on a number of external software libraries, primarily TPIE, GRASS, and GDAL. As mentioned in the introduction, TPIE [11], is a templated, portable, I/O environment written in C++ that provides support for implementing I/O-efficient algorithms and data structures. Our implementation work was greatly simplified by the fact that all main primitives of our algorithms—scanning, sorting, stacks and priority queues—are already implemented I/O-efficiently in TPIE. For data visualization and basic data manipulation, we used the open-source GIS GRASS [52], written primarily in C. In particular, for our grid DEM construction algorithm, we used the regularized spline with tension interpolation code that exists in the GRASS module

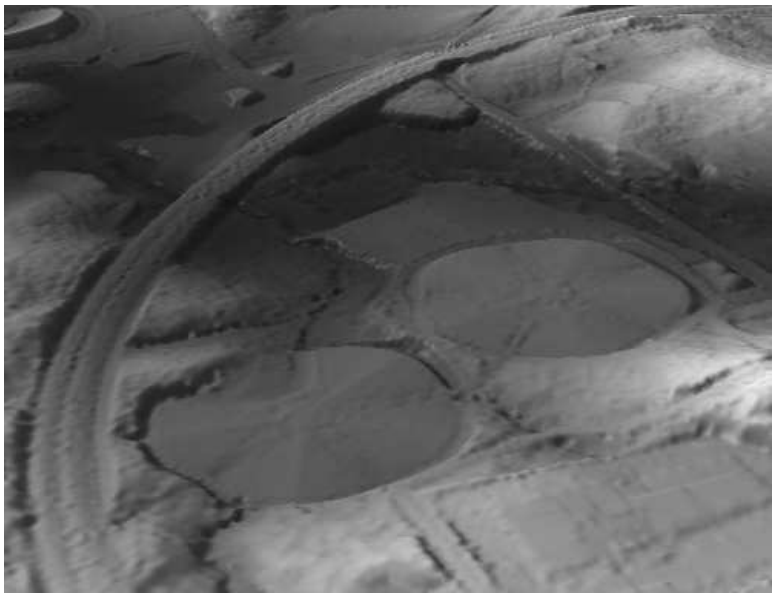


Figure 6.2: Sample bare Earth lidar data still shows some man-made features including the bridge in the center left and eight baseball fields grouped into the two circles shown in the foreground.

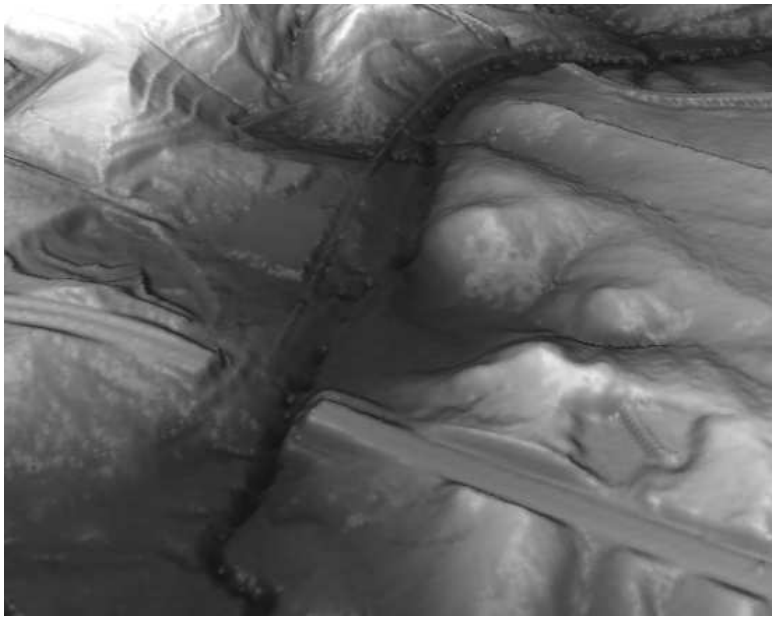


Figure 6.3: In this lidar example, the bridges have been cut by the data providers to allow water to flow through. Many bridges across major waterways have been cut, but many bridges across smaller streams have not.

Resolution (feet)	40	20	10
# of quad-tree points (millions)	205	340	415
# of grid cells (millions)	99	397	1590
Pipeline stage			
DEM Construction	14h 19m	19h 56m	27h 12m
Hydrological conditioning	16m	1h 24m	7h 55m
Flow Modeling			
Flow Routing	15m	1h 26m	6h 34m
Flow Accumulation	19m	1h 40m	7h 35m
Watershed extraction	46m	2h 28m	14h 39m
Total	15h 55m	25h 54m	63h 34m

Table 6.1: Running times for various pipeline stages on the Neuse river basin data set.

`s.surf.rst`. Because of scalability problems with the new vector engine in GRASS 6.2, we used GRASS 5.4 for our grid construction algorithm, and GRASS 6.2 for everything else. Finally we used the Geospatial Data Abstraction Library (GDAL) [50], to read and write a number of raster and vector formats.

While developing our own software, we also contributed several bug fixes and improvements to both the TPIE and GRASS code, including writing an improved sorting algorithm for TPIE, updating and maintaining the TPIE code in general, submitting patches to GRASS to support large raster files, and fixing a few bugs in the GRASS modules `s.surf.rst` and `r.terraflow`.

All of our experiments in this chapter were run on a Dell Precision Server 370 (Pentium 4 3.40 GHz processor) running the Linux 2.6.11 kernel. The machine had 1 GB of physical memory. All test data were stored on a single 400 GB SATA disk drive. We set the memory limit of TPIE to 640MB and instructed TPIE to alert the user if the memory limit was exceeded. Setting the memory limit lower than the amount of physical memory allows the operating system to use the remaining memory for other system processes without competing with our application for memory.

6.3 Scalability

For the scalability results, we constructed grids of 40, 20, and 10 foot resolution, removed sinks, computed the river network, and extracted the watershed boundaries. For these experiments, we used the regularized spline with tension approximation method for the grid construction. The following approximation parameters described in Section 2.2 and Section 2.4 were used: The maximum number of points in a quad-tree leaf was $k_{\max} = 15$. The maximum number of points used in the interpolation of any quad-tree leaf was $n_{\max} = 135 = 9k_{\max}$. The tension parameter φ was set to 40 and the smoothing parameter w_0/w_j was set to be a constant 0.1 for all points. The minimum distance ε between any two points in a quad tree segment was set to be half of the grid cell resolution. The persistent threshold was set to 55ft. These parameters were determined to produce good DEMs after experimenting with a number of parameters. In particular we found that for all three grid resolutions, only 15 minima remained after hydrological conditioning with a persistence over 50 feet. All but two of these minima were quarries, while the last two were small watershed blocked by bridges. A persistence threshold of 55 feet preserved the quarries while connecting the areas blocked by bridges to the rest of the Neuse river network. A visual overview of the output is shown in Figure 6.4 and Figure 6.5 for the 20ft grid case. Data at other resolutions look similar. Running times of our pipeline stages are shown in Table 6.1.

Looking at the results of Table 6.1 in more detail, we first consider the grid construction stage at multiple resolutions. Because the point thinning parameter ε is one half the grid resolution by

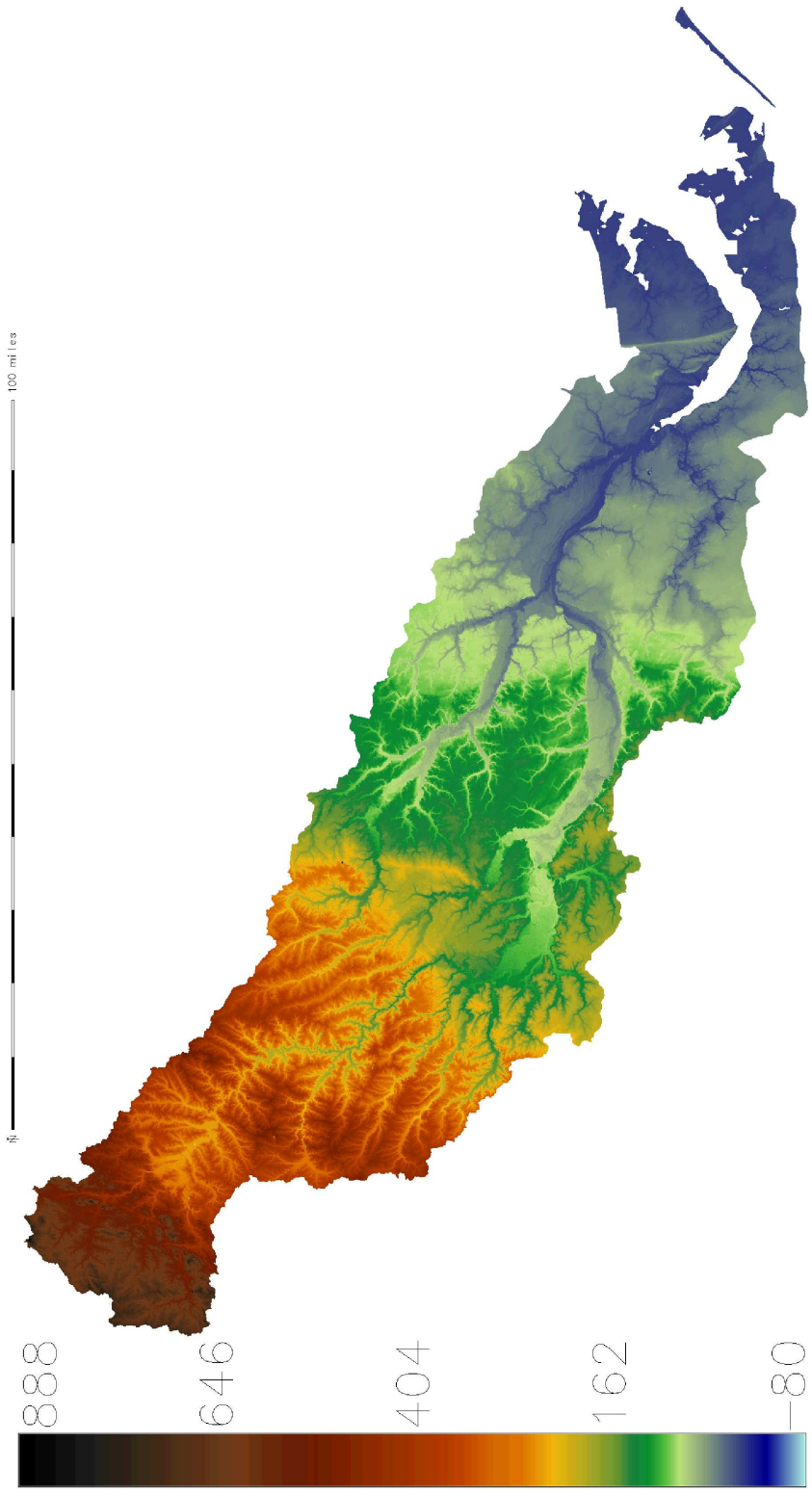
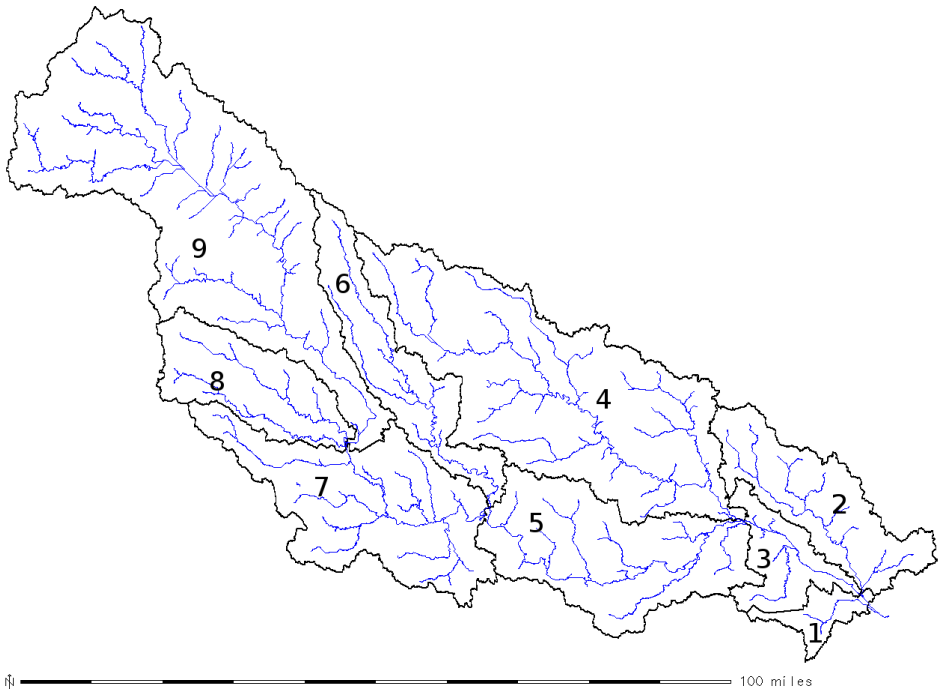
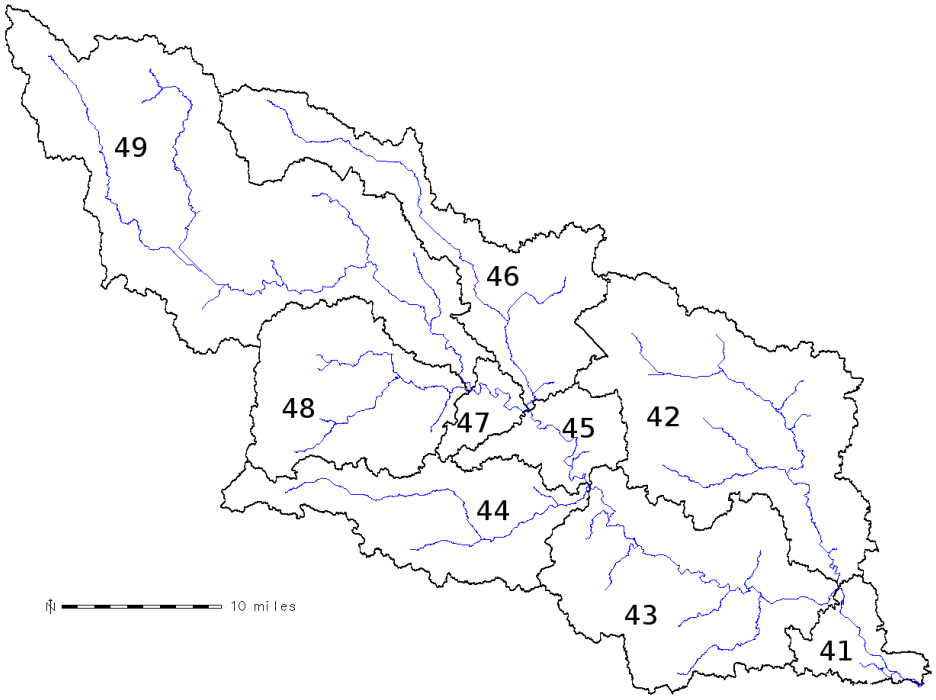


Figure 6.4: DEM of Neuse river basin derived from lidar points



(a)



(b)

Figure 6.5: (a) First level of Pfafstetter watershed labels for largest basin in Neuse. (b) Recursive decomposition of basin four.

default, we note that there are fewer points in the quad tree for the 40 foot resolution grid than for the 20 foot or 10 foot grid. In changing the resolution from 40ft to 20ft, the number of points in the quad tree increases 65% while the 10ft DEM contains only 22% more points than the 20ft case. The number of grid cells in the output DEM is inversely proportional to the grid resolution. Note that many cells in the bounding box containing the Neuse data set have no value assigned to them because they are outside of the study area. In Table 6.1, we do not count these so called *no-data* or null value cells in our cell count. At 20ft resolution, the number of quad-tree points and grid cells are similar indicating that with roughly one lidar point per 20×20 sq.ft, the average lidar point spacing is close to 20ft. Note for the 10ft DEM the number of grid cells is almost four times the amount of lidar points in the grid. One of the benefits of using an interpolation method over a simple binning approach is that we can reconstruct a terrain surface given a sparse point sample. Thus, even though the we cannot have a point sample for every 10ft grid cell, interpolation can compute the elevation of grid cells with no data points using nearby points, and can construct grids with more cells than the number of original points.

The most expensive of any of the pipeline phases is the DEM construction, consuming over 40% of the the total run time in the 10ft case and over 95% of the total run time in the 40ft case. As noted in Chapter 2 this is primarily due to the interpolation code, which is CPU bound and not I/O-bound. While still a time intensive stage, we improved the performance of the construction over previous experiments [2] by tuning the interpolation parameters. We found that a $k_{\max}$ of 15 and npmin of 135 led to improved run time performance. We also verified that the resulting river networks and watershed boundaries were similar to a grid computed using a much higher $k_{\max}$, so we could optimize $k_{\max}$ for faster speed. Whereas the interpolation phase of grid construction represented over 80% of the total run time in the experiments describe previously in Chapter 2, interpolation represented slightly over 50% of the run time when using a smaller $k_{\max}$.

The memory usage of our algorithms is typically very low, while we set the TPIE limit to 640MB, we never needed more than 20MB of memory for the grid construction, hydrological conditioning, flow accumulation, or watershed extraction. The extra memory was used only occasionally for sorting steps or for maintaining a portion of an I/O-efficient priority queue in memory. Flow routing on flat surfaces was the only part of the pipeline that required large amounts of memory and only in extreme circumstances. In our current implementation, we load each flat area separately into memory and then compute flow directions across each flat area. While running on the 10ft grid, we detected one large flat area containing 9.4 million grid cells, representing a total area of 21,600 acres (8,740 hectares) or 33.7 square miles (87.25 km^2). This is the area containing Falls Lake, a man-made reservoir between Raleigh and Durham, North Carolina, and depicted in Figure 6.6. A dam 30ft above the water surface creates a sink with a persistence of 30ft behind the dam. This sink is flooded when the persistence threshold is 55ft and this flooding creates the a very large flat area. For this experiment, we temporarily needed over 571MB of memory to process this single flat area. This is only slightly below the 640MB memory limit we used for TPIE on our 1GB machine. All other flat areas were less than 100MB in size and easily fit within our 640MB memory limit. Using our current implementation, we would be unable to process the flat area at 5 foot resolution using only 1GB of memory. Given 2GB of memory, we could handle this flat area even at 5ft resolution, but we may need to consider implementing an I/O efficient method for computing shortest paths [23] that can route flow across very large flat areas.

The final stage of our pipeline, watershed hierarchy extraction, takes roughly twice the amount of time as the hydrological conditioning, flow routing, and flow accumulation stages, and is still considerably faster than the grid construction stage. The primary reason for the longer run-time is that the Pfafstetter algorithm performs four $O(\text{sort}(N))$ I/O steps and two scanning steps, with both the flow routing and flow accumulation perform half as many sorting steps in addition to two scans. Since $O(\text{sort}(N))$ is approximately three scans for most data set sizes, we *expect* Pfafstetter to be roughly twice as slow as routing and accumulation. The slowest stage of our pipeline—DEM construction—is still quite I/O-efficient, but is simply CPU bound by a number of computationally

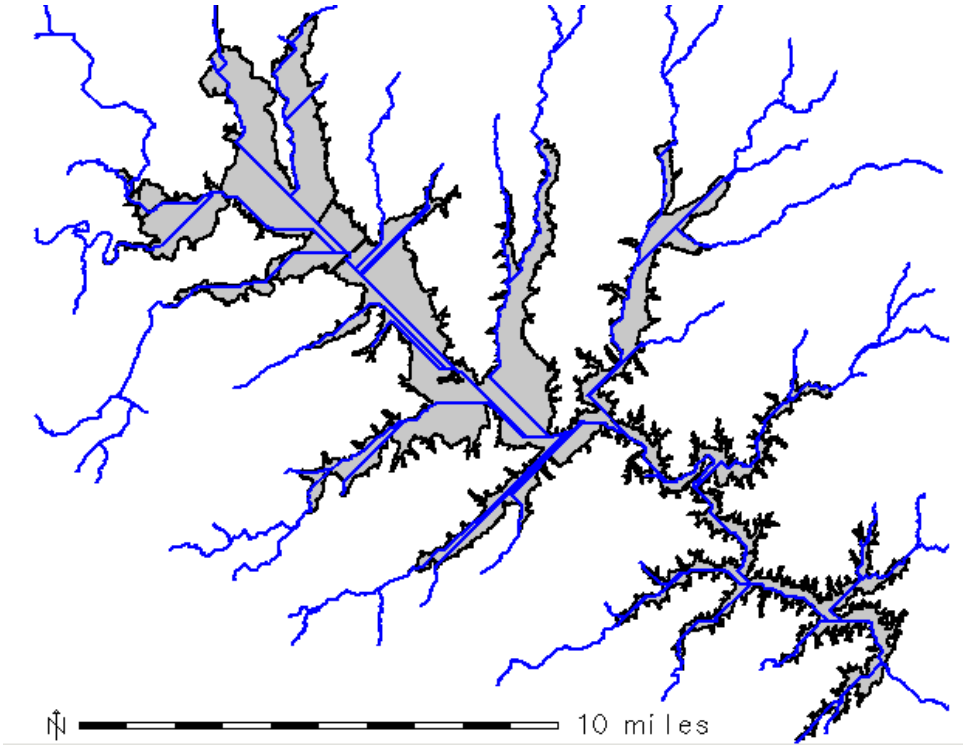


Figure 6.6: Falls lake, with a dam located near the Southeast corner of the figure. The boundary of the Falls lake flat is outlined in black while blue lines show rivers entering the reservoir and routed across the flat area.

intensive calculations. None of our other algorithms execute code internally that has a run time worse than $O(n \lg n)$, while interpolation is using an $O(n^3)$ interpolation routine.

Overall, we consider our approach scalable to massive data sets. In direct comparisons to other interpolation methods or flow routing methods, out approach is faster and more scalable. As noted in previous chapters, many previous methods simply crash on extremely large data sets. One example of an I/O-efficient algorithm for terrain modeling that is designed to scale large data sets is TerraFlow, but we also outperform this algorithm.

6.4 Comparison to TerraFlow

Much of our initial work was inspired by TerraFlow [14], the only other I/O-efficient algorithm for hydrological modeling on grid DEMs. Since the TerraFlow program only worked on grids, it could not benefit from new lidar point data sets without a scalable grid construction algorithm. Our implementation improves on TerraFlow in a number of ways. First, we provide a I/O-efficient grid DEM construction algorithm. Second, we add a new hydrological conditioning algorithm based on topological persistence. Whereas TerraFlow removes all sinks except for a single global minimum, we can preserve multiple sinks above a given persistence threshold. We also implemented a simple way for detecting flat areas under the realistic assumption that a constant number of grid rows fit in memory. Furthermore, we extended the geo-processing beyond flow modeling to include computation of watershed hierarchies. Finally, much of the work in this thesis was combined with recent work on TINs DEMs by Ke Yi [95] into a common framework that was simply not possible with TerraFlow. Because our new pipeline is significantly different from TerraFlow, we found it easier to implement our new modeling methods from scratch rather than modifying the existing TerraFlow code.

Because our new implementation constructs many of the same gridded outputs that TerraFlow constructs, it is natural to compare the performance of the two implementations. We compared our new implementation to the TerraFlow implementation for the 20ft grid case. Due to bugs in the TerraFlow code related to handling rasters with over 2 billion grid cells (including no-data values), we were unable to run TerraFlow on the 10ft Neuse data set. While flow routing times were almost equal for both implementations as shown in Table 6.2, our approach is considerably faster in the other two stages and overall run time. TerraFlow also pre-processes the terrain to distinguish between *interior* no-data completely surrounded by real data values and *boundary* no-data cells that have a path of no-data cells that is connected to the boundary of the grid. In TerraFlow, no-data values in the interior are treated as infinitely high walls and flow is routed around these no-data values while boundary no-data values are considered infinitely deep sinks and flow is routed into these boundary no-data cells. We do not make this distinction because we can construct grid DEMs that interpolate data values in regions where no point sample exists. Most modern terrain data sets either have filled in internal no-data values, or a GIS module can fill in these no-data values as a pre-processing step.

Even ignoring the pre-processing of no-data step, TerraFlow still takes considerably longer than our new approach on the same machine. There are two primary reasons for our speedup over TerraFlow. First, TerraFlow uses a different algorithm that, while still has a $O(\text{sort}(N))$ I/O bound, performs more scanning and sorting steps to hydrologically condition the terrain. Second, in the case of flow routing and accumulation, we use a compact edge-based representation for flow directions. In the single flow direction model, we only store information about a single downslope neighbor cell for each cell in the grid DEM. On the other hand, TerraFlow keeps a copy of all eight neighbors with each grid cell, effectively multiplying the original input size by eight. While our compact representation is only a constant factor reduction in the size of the flow direction files, this constant factor can have a large impact on the real running time of the algorithm.

Method	Ours	TerraFlow
Pipeline stage		
Pre-processing	NA	1h 33m
Noise removal	1h 24m	6h 10m
Flow routing	1h 26m	1h 22m
Flow accumulation	1h 40m	3h 12m
Total	4h 30m	12h 17m

Table 6.2: Running times for our pipeline stages that overlap with TerraFlow.

6.5 Sensitivity to Construction Parameters

In our implementation, we expose a number of interpolation parameters to the user, including $k_{\max}$, ε, and a constant smoothing parameter. Tuning these parameters can significantly improve the run-time of the grid construction while still generating high quality grid DEMs suitable for hydrologic modeling. We conducted a number of experiment, starting with $k_{\max}$ that examined the influence of these three parameters. For each of the experiments in this section, we used a subset of 16.6 million points, covering approximately 163 square miles (11.4 million grid cells), extracted from a small portion of the Neuse river basin. We set the grid resolution to 20ft, and set the tension to the default of 40 and the persistence threshold to 30ft.

The maximum number of points per quad-tree leaf, $k_{\max}$, is a very influential parameter in the grid construction algorithm of Chapter 2. As $k_{\max}$ increases, fewer quad-tree leaves are created, but the quad-tree leaves contain more grid points and there are more points in neighboring leaves. Fewer quad-tree leaves leads to a faster construction quad-tree construction time, but more points increases the cost of surface approximation in each leaf. Decreasing $k_{\max}$ increases the quad-tree construction cost but decreases the approximation cost. A very small $k_{\max}$ also does not provide a sufficient sample size to accurately construct a plausible representation of the surface. We varied the value of $k_{\max}$ in the range of 1 to 64. For these experiments we set ε=10ft and the smoothing parameter to be 0.1. The results in Table 6.3 summarizes our findings. From the second row in the table, we see that the time to construct the quad tree and find points in neighbor quad tree leaves decreases as $k_{\max}$ increases. This is expected because as $k_{\max}$ increases, the quad-tree has fewer leaves. For example, with $k_{\max} = 1$, the quad-tree has 10.8 million leaves, while for $k_{\max} = 64$ the tree has only 453 thousand leaves. While increasing $k_{\max}$ decreases the quad-tree construction time, it increases the interpolation time dramatically. Over our range of $k_{\max}$ values, the interpolation time increased by a factor of 210, while the construction time decreased only by a factor of 3 over the same range. We note that in this case study, the interpolation time and run-time are roughly the same for $k_{\max} = 8$.

Changing $k_{\max}$ also influences the number of no-data cells in the output grid, the number of sinks, and the root-mean-square (RMS) deviations of the terrain. For small $k_{\max}$ values, quad-tree leaves containing no points are more common. If there are no points in either a given quad-tree leaf or its neighbors, we cannot interpolate grid cells in this region. In this case we write a no-data value for these cells. In our experiments, we found that smallest $k_{\max}$ value for which we had zero no-data values was $k_{\max} = 16$. We expect $k_{\max}$=1 to have the most sinks as it is closest to a simple binning approach that computes the elevation of a grid cell by a simple average of all points falling into the grid cell. However, in our interpolation method, we only need one point in a quad-tree leaf or its neighbors to interpolated all grid cells in that leaf. Indeed, a simple binning approach results in over 1.00 million no-data cells while for $k_{\max} = 1$ there are only 4225 such cells.

As $k_{\max}$ increases, the number of sinks in the grid DEM also increases. For small values of $k_{\max}$, the interpolation procedure uses only a small number of points for each quad-tree leaf and the interpolated surface has less complexity than a quad-tree leaf interpolated on more points. Thus

k	1	2	4	8	16	32	64
build time (min)	37.9	39.1	29.2	19.9	15.8	13.5	13.4
interpolation (min)	3.5	5.5	7.2	13.7	31.3	77.6	736.9
total time (min)	41.4	44.6	36.4	33.6	47.1	91.1	749.4
# no-data cells	4225	1299	245	60	0	0	0
# sinks (thousands)	117.9	137.3	157.3	178.0	200.7	217.8	247.1
RMS deviation (ft)	0.444	0.372	0.376	0.345	0.267	.256	N/A

Table 6.3: Impact of $k_{\max}$ on construction time, number of sinks, and RMS deviation.

larger $k_{\max}$ values result in more sinks. Finally, we compared the difference between each output grid and some base-grid and computed the RMS deviation. Because we have no "ground-truth" data set to compare against, we arbitrarily chose one of the grids ($k_{\max}$=64) to compare with. We see that as $k_{\max}$ increases, the RMS deviation decreases. However, the overall decrease is less than 0.2 feet (2.5 inches).

By visually inspecting the data sets, we could not find any major discrepancies across data sets. There were some slight deviations in areas with high slope or rapid topographic change. Because the deviations were only slight in this test area, we detected no major changes in either the river network or watersheds extracted from grids constructed using different $k_{\max}$ values. The value of $k_{\max}$ primarily effects computation time and the number of no-data cells. For this reason, we choose a value of $k_{\max}$ that reduces the number of no-data cells and constructs a grid quickly. A value of $k_{\max}$ between 8 and 16 yields good results in our case study.

In addition to tuning $k_{\max}$, we can also tune the thinning parameter ε. Recall that if any two points in a quad-tree leaf are a distance ε apart, the most recently inserted point is discarded. By default, ε is one-half the grid cell size. We conducted a few experiments in which we varied ε and examined the effects on run time, number of sinks, and RMS deviations. For these experiments, we set $k_{\max}$ to 8, and the smoothing parameter to 0.1. Our results are summarized in Table 6.4. As expected, decreasing ε increases the number of points used in interpolation and increases the interpolation time. For ε = 5ft, less than 1000 points are discarded of 16.6 million total points. Again, with more points, the interpolated surface complexity increases and the number of sinks increases as well. We computed the RMS deviation, using the ε = 10ft grid as a base, because 10ft is the default value. While both deviations are small, the smallest value occurs for ε =5ft.

After constructing the grid, we computed the river networks and watershed boundaries. We observed no major differences in the networks or watershed boundaries. We did observe a few (fewer than 5) minor differences in the river network. An example of such a minor discrepancy is shown in Figure 6.7. For ε=10ft or 5ft, we find the extracted river (white) follows the actual river course visible in the grid DEM. In the 20ft case however, the extracted river (black) follows a different course. Because the default ε=10ft results in a more accurate grid DEM than the 20ft case, we chose the default ε of one-half the grid cell size for our other tests. We found that decreasing ε further only increased the run time of the construction without significantly improving the quality of the DEM.

For our final set of experiments on grid construction parameters, we varied the smoothing param-

ε	20	10	5
points interpolated (million)	8.58	14.0	16.6
interpolation time (min)	9.0	13.7	18.0
# sinks (thousands)	151.2	178.0	184.9
RMS deviation (ft)	0.311	N/A	0.197

Table 6.4: Impact of ε on construction time, number of sinks, and RMS deviation.

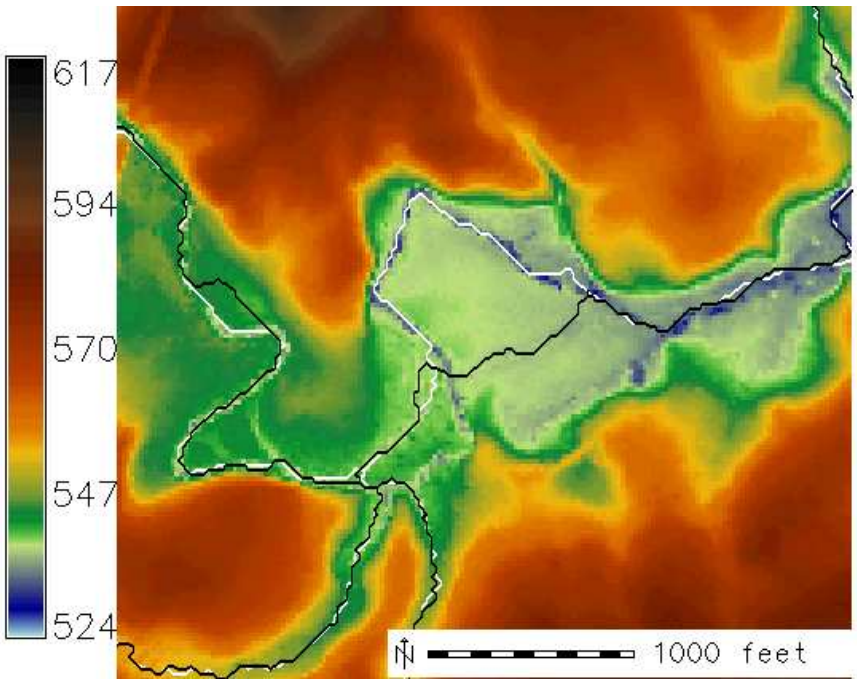


Figure 6.7: Rivers extracted using ε =10 ft (white) and 20 ft (black).

eter while using $k_{\max} = 8$ and ε =10ft. A smoothing parameter of 0 results in a interpolated surface that passes exactly through the input points. For a non-zero smoothing parameter, the algorithm constructs an approximation surface in which the input points can deviate from the constructed surface. The default smoothing parameter is 0.1. Smoothing only effects the interpolation routine, and not the quad-tree construction. By increasing the smoothing parameter, we can decrease the number of sinks in the constructed terrain. Our results are summarized in Table 6.5. We compute the RMS deviation by comparing the grids to the base grid with the default smoothing of 0.1. For a smoothing parameter of 5, the RMS deviation increased significantly. We also observed some strange blocky edges in the terrain with this high smoothing parameter that suggested that such high smoothing values should be avoided. While increasing smoothing can decrease the number of sinks somewhat, many sinks still remain even after significant smoothing. By looking at the persistence of the sinks created, we noted that increasing smoothing typically eliminates sinks with very small persistence while occasionally reducing the persistence of other sinks by a foot or less. Since smoothing did not remove larger sinks, smoothing had little effect on the hydrologically conditioned DEM, the river network, or watershed boundaries as the small differences in smoothing where negligible compared to the extensive flooding performed by the hydrological conditioning stage. Thus, we found that we could just use the default smoothing. For other terrain applications, such as topographic analysis, Mitasova et al. [68] describe the benefits of tuning the smoothing parameter.

smoothing	0	0.1	1	5
# sinks (thousands)	186.4	178.0	129.7	67.9
RMS deviation (ft)	0.0131	N/A	0.089	0.300

Table 6.5: Impact of smoothing parameter on number of sinks, and RMS deviation.

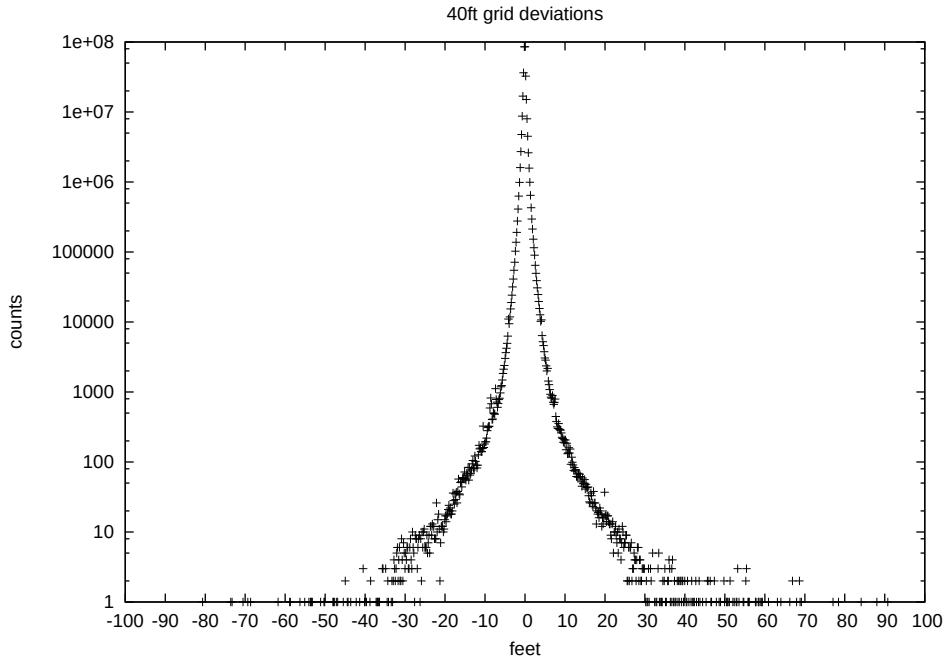
6.6 Sensitivity to Grid Resolution

The run time of all stages of our geo-processing pipeline depend heavily on the number of grid cells in the terrain and therefore the grid resolution. While our pipeline can handle very large input sizes, it may not be necessary to create a very high resolution grid with a small cell size for a particular study. For example, a 10ft DEM may be necessary for a very local detailed study, but a statewide study of watersheds may only need a coarse 40ft DEM. In this section, we consider the effects of grid resolution on derived DEM products. We constructed DEMs of 10, 20, and 40 foot resolution for the entire Neuse basin and looked at the deviations in the resulting DEMs, the number of minima created, the persistence of the minima, and the resulting river network and watershed boundaries. For each of these experiments we used $k_{\max} = 16$, a smoothing parameter of 0.1, and a persistence threshold of 55ft. We used the default ε of half the grid resolution.

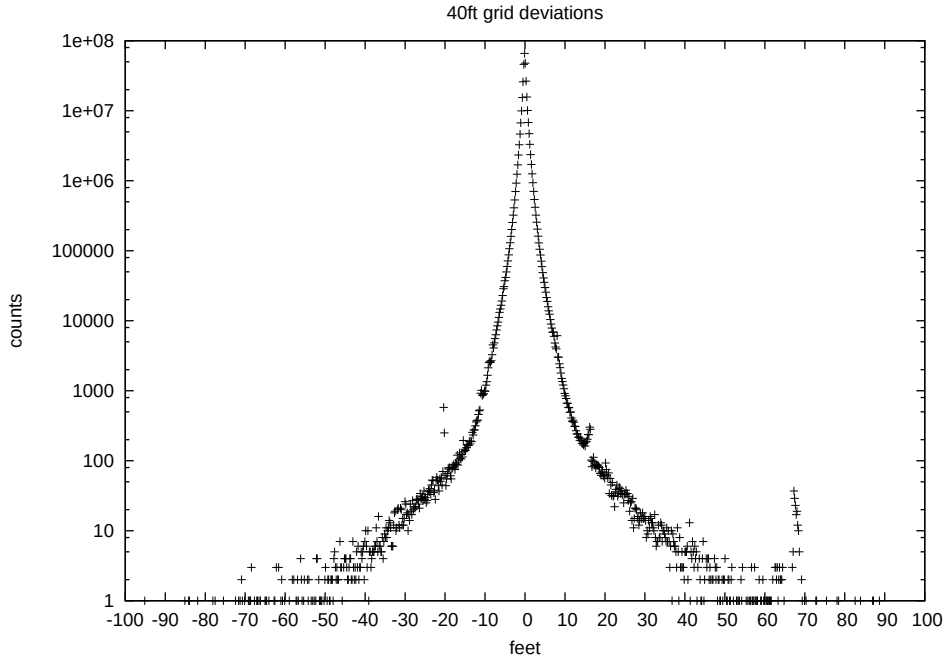
Because we had no "ground truth" elevation measurements, we used the 20ft grid as our baseline for computed the RMS deviation of the other grids. A histogram of the deviations for each resolution is shown in Figure 6.8(a) and Figure 6.8(b), respectively. Note that the vertical scale is logarithmic and the histograms are centered around a deviation of 0ft. For the 10ft grid, deviations have a magnitude of less than 1ft for 94.6% of the grid cells and a magnitude less than 3ft for 99.8% of all grid cells. For the 40ft grid, 83.7% and 99.0% of cells have a deviation with magnitude less than 1ft and 3ft, respectively. In the 40ft grid case, a few spikes appear in the deviation histogram around 15 and 70 feet. In each of these cases, the deviations occurred at the bottom of quarries that had a sparse point sampling. In the grid construction phase, cells in these areas were interpolated using very few points subject to large deviations depending on which far away points were used to interpolate in these areas. Many of these quarries are over 100ft deep, so it is reasonable to suspect that in the 40ft grid, we might interpolate of group of cells in the DEM construction using one point high on the walls of the quarry, while the 20ft grid might sample one point closer to bottom of the quarry. Each spike is an instance of sparse point sampling in small isolated areas, and overall, the 40ft grid agrees well with the 20ft grid.

If we look closer at the spatial distribution of areas with high deviations, we can verify that quarries and other sharp drop-offs indeed are the source of most high deviation between grid resolutions. In Figure 6.9, we see an example of a quarry that is over 100ft (30m) deep. The first figure shows the constructed 20ft grid. Some block-like artifacts are apparent at the bottom of the quarry. This is indicative of sparse point sampling in the grid construction. Note that these artifacts only appear in the bottom of this quarry and in general the interpolated DEM is smooth where there were sufficient sample points during grid construction. The middle and bottom figures display a number of white dots where the deviation between the 10ft grid (middle) and 40ft grid (bottom) exceed 5ft. Note these areas of high deviation are clustered around areas of rapid topographic change. Because the 40ft grid cannot resolve details as well as the 20ft or 10ft grid, there are more places in the 40ft grid where the deviation is higher. Note that in addition to the quarry, there are some high deviations along the river banks, another area of rapid topographic change. Visual inspection of other areas with high deviations showed that the highest deviations occurred in areas with steep slopes or sharp boundaries. This is to be expected as the cell size will effect the ability to represent these features accurately.

Next, we examined the effect of different grid resolutions on the number of sinks created and the persistence of these sinks. Our findings are summarized in Table 6.6. The total number of sinks increases as we reduce the grid cell size. However, the percentage of grid cells classified as sinks decreases with higher resolution. For example, the 10ft grid has 27.33 million sinks representing 1.7% of all grid cells, while over 3.5% of the 40ft grid's 99 million grid cells are classified as sinks. While the 10ft grid has many more sinks, Table 6.6 shows that most of these sinks have a very low persistence with only 47.5% of all sinks having a persistence greater than 0.1ft (3cm). It is interesting to note that the number of sinks above some persistence threshold systematically decreases with decreasing cell size. One possible reason for this is that the higher resolution grids will make both

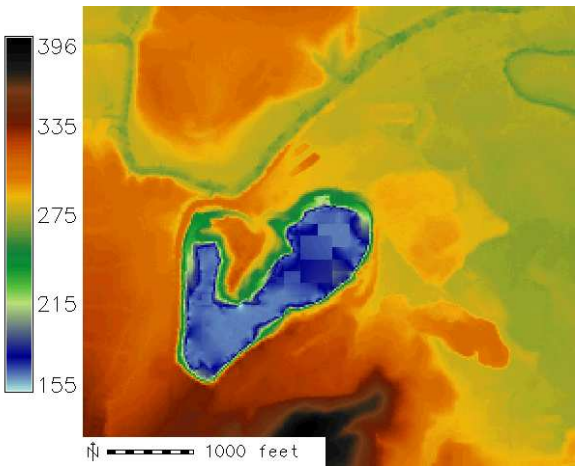


(a)

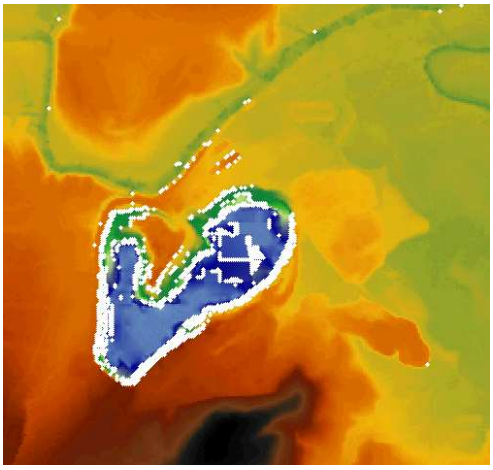


(b)

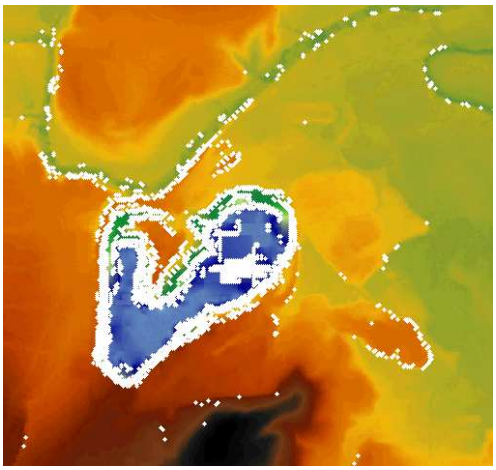
Figure 6.8: Distribution of deviations from 20ft grid elevations for (a) 10ft grid (b) 40ft grid. Vertical scale is logarithmic.



(a)



(b)



(c)

Figure 6.9: Spatial distribution of deviations from (a) 20ft grid elevations for (b) 10ft grid (c) 40ft grid. White points indicate spots where vertical elevation deviation exceeds 5ft.

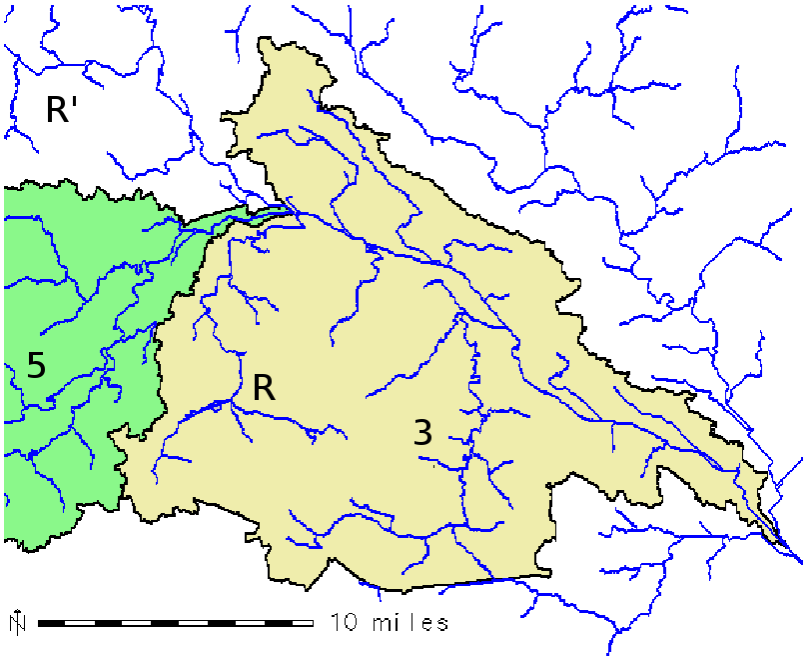
Resolution (feet)	10	20	40
# of sinks (millions)	27.33	12.45	3.54
# of sinks with persistence			
greater than 0.1ft	12,970,927	7,775,221	2,629,032
greater than 1ft	1,230,822	861,123	443,693
greater than 5ft	29,882	22,447	16,259
greater than 10ft	3,122	2,572,	1,860
greater than 20ft	440	358	280
greater than 30ft	121	96	75
greater than 40ft	15	15	14
greater than 100ft	12	12	12

Table 6.6: Number of sinks in grid DEMs of various resolutions. Persistence values listed in feet. The 10ft grid has many more smaller sinks, but all grids agree on the the number of sinks with persistence greater than 100ft.

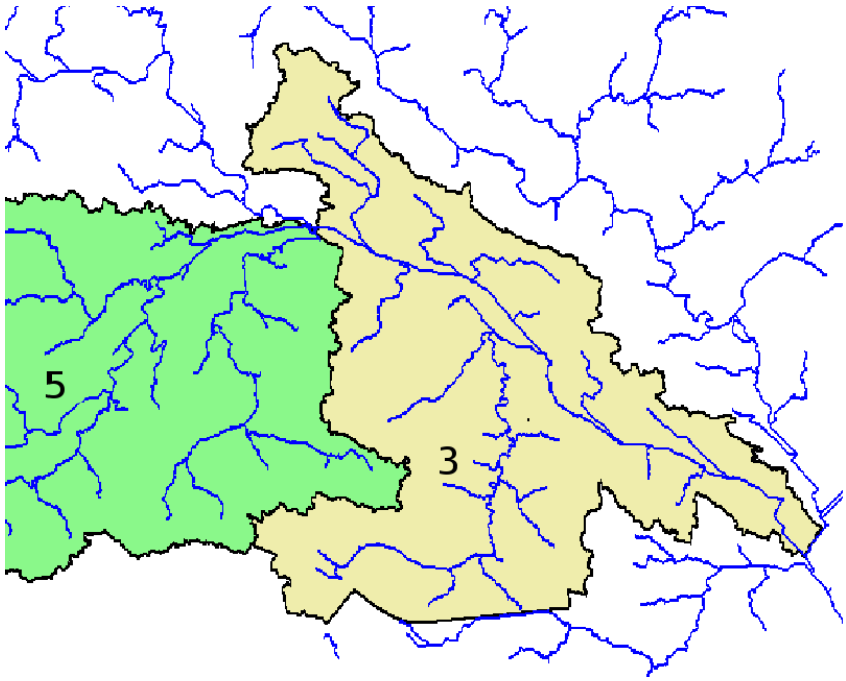
minima and maxima more pronounced. Imagine trying to find the exact location of a minimum using one measurement near the minimum in the 40ft case versus using sixteen measurements near the minimum in the 10ft case. The 10ft grid has a higher probability of evaluating a cell close to the true minimum, while if the 40ft grid cell is slightly off from the true location of a minimum, it will overestimate the height of the minimum. This overestimation of minimum height for the 40ft case would lead to a systematic decrease in persistence of sinks for the larger grid cell sizes. The persistence value depends on the height of the saddle as well and a coarse grid resolution could either overestimate or underestimate the true height of the saddle, so while it may be possible given a large systematic overestimation of saddle heights that coarse grids would have systematically higher persistence values, but experimental evidence seems to indicate this is not the case.

As a final experiment regarding the effects of grid resolution on other pipeline stages, we looked at the watershed hierarchies derived from 10ft, 20ft, and 40ft grids. Most watershed boundaries were in good agreement, however, there are a few exceptions. We show an example of different watershed boundaries in Figure 6.10. Both figures show two watersheds with level 1 Pfafstetter labels 5 and 3. Water flows West from the hydrological unit, or basin, labeled 5 into 3. The top figure shows the basin boundaries for the 10ft grid and the 20ft grid looks similar to the 10ft grid at this scale. Note that the river labeled R in the top figure is in basin 3. In the bottom figure generated on a 40ft grid, the river labeled R is contained in basin 5. This is the only noticeable difference between the level 1 Pfafstetter basins extracted from the 10, 20, and 40ft grids.

Upon zooming to the outlet of basin 5 as shown in Figure 6.11, we begin to see the reason for the different delineations. Figure 6.11(a) shows the underlying terrain at 10ft resolution before hydrological conditioning. An overlay of basin boundaries (black) and the the rivers (white) extracted from the hydrologically conditioned 10ft DEM is also shown. Two large rivers, labeled R and R', join the main river very close to each other. In the 10ft case, shown in Figure 6.11(b), the mouths of R and R' are separated by about a 1/4-mile (400m), and R' is upstream of R. In the 20ft case, shown in Figure 6.11(c), the main river contained in basin 5 migrates to the southeast and the rivers R and R' are barely separated with R' still slightly upstream. In the 40ft case, shown in Figure 6.11(d), R joins the main branch upstream of R' and is therefore labeled as being part of basin 5. The Pfafstetter labels are consistent for each of the three resolutions, but in areas where two tributaries join a main river close to each other, the relative upstream ordering of tributaries can change and thus the watershed boundaries change. While it is good that the Pfafstetter labeling algorithm can generate appropriate labels as river connectivity changes, only one of these possible labelings is correct in reality. A visual inspection of Figure 6.11(a) seems to indicate that



(a)



(b)

Figure 6.10: Two watershed regions labeled by Pfafstetter algorithm have quite different boundaries in the (a) 10ft grid and (b) 40ft grid.

R likely joins the main river bed upstream of R' and the National Hydrography Dataset available online from the USGS also indicates that R (Mosley Creek) flows into the Neuse River upstream of R' (Contentnea Creek) about 10 miles East-northeast of the city of Kinston, NC. Thus, the lower resolution 40ft grid has the correct order of river connectivity, though at all three resolutions, river R joins the main river about 1 mile East of the actual mouth of Mosley Creek.

While large noticeable changes in watershed boundaries are rare in the higher levels of the watershed hierarchy, slight differences in watershed boundaries are more common in smaller watersheds less than an few thousand acres, where there are a number of small streams of roughly the same flow accumulation flowing into a main channel. In areas where the hydrological conditioning stage has flooded much of the terrain, the flat routing method can create some artificial river networks that do not agree well with reality and can result in inaccurate Pfafstetter basins. Because the Pfafstetter labeling method encodes watershed connectivity properties, it is relatively straightforward to split or merge the boundaries of two Pfafstetter boundaries if the user knows the correct connectivity from auxiliary data sets. Overall, we find that the optimal choice of grid resolution depends on the application. For a regional study covering the entire Neuse basin, a 40ft grid produces both a detailed river network and watershed hierarchy that only differs from the 10ft or 20ft grid in a few spots. For more local studies, a 20ft or 10ft grid would be more suitable. The increased detail of a high resolution grid is quite noticeable when zoomed into a very small area and the lidar data certainly support grids with resolutions as high as 10ft.

6.7 Sensitivity to Persistence Thresholds

One of the advantages of our new pipeline implementation over previous algorithms is the ability to set a persistence threshold to remove only the subset of sinks from a grid DEM that have a persistence value below the threshold. In this section we consider how the persistence threshold effects the river network and thus the watershed hierarchy. We extracted watershed boundaries from a 20ft grid DEM after removing sinks below three persistence thresholds; 50ft, 40ft, and 30ft. None of these thresholds removed all sinks; the number of sinks remaining after hydrological conditioning were 15, 28, and 96, for the 50ft, 40ft, and 30ft grids, respectively. The highest persistence of any minima in the basin was 253 feet inside a quarry Northeast of Raleigh-Durham International Airport. Figure 6.14 shows the location of sinks and the boundaries of their drainage areas for the three persistence thresholds. In the 50ft case (Figure 6.14), we could easily visually inspect each sink individually. Of the fifteen total sinks, thirteen of these sinks were genuine quarries with no drainage outlet and were correctly preserved by the hydrological conditioning algorithm. An example of such a quarry is shown in Figure 6.12. The area draining into the quarry is relatively small at 600 acres (240 hectares) and this excluding this drainage area from the river network does not significantly alter the watershed boundaries on a regional scale. This is to be expected as quarries do not typically drain large areas.

An example of a sink that is preserved with a threshold of 50ft but should have been removed is shown in Figure 6.13. The drainage area of this sink is over 10 times larger at 7300 acres than the previous case. The river network is blocked by a bridge and a road in the Southeast corner of the image where the watershed boundary is almost linear. Most of the water in this watershed should flow out under the bridge which is about 25 feet above the river surface. However, the quarry just upstream of the bridge results in the hydrological conditioning algorithm assigning a high persistence value (170+ feet) to this quarry and directing all water blocked upstream by the bridge into the quarry. The correct modification to the terrain would be to remove the bridge crossing the river just East of the quarry. The quarry would still have a high persistence, but water not within the quarry boundaries would travel around the quarry and under the bridge. With the exception of this example and one other small watershed, a persistence threshold of 50ft in this case study removed almost all sinks that were not quarries.

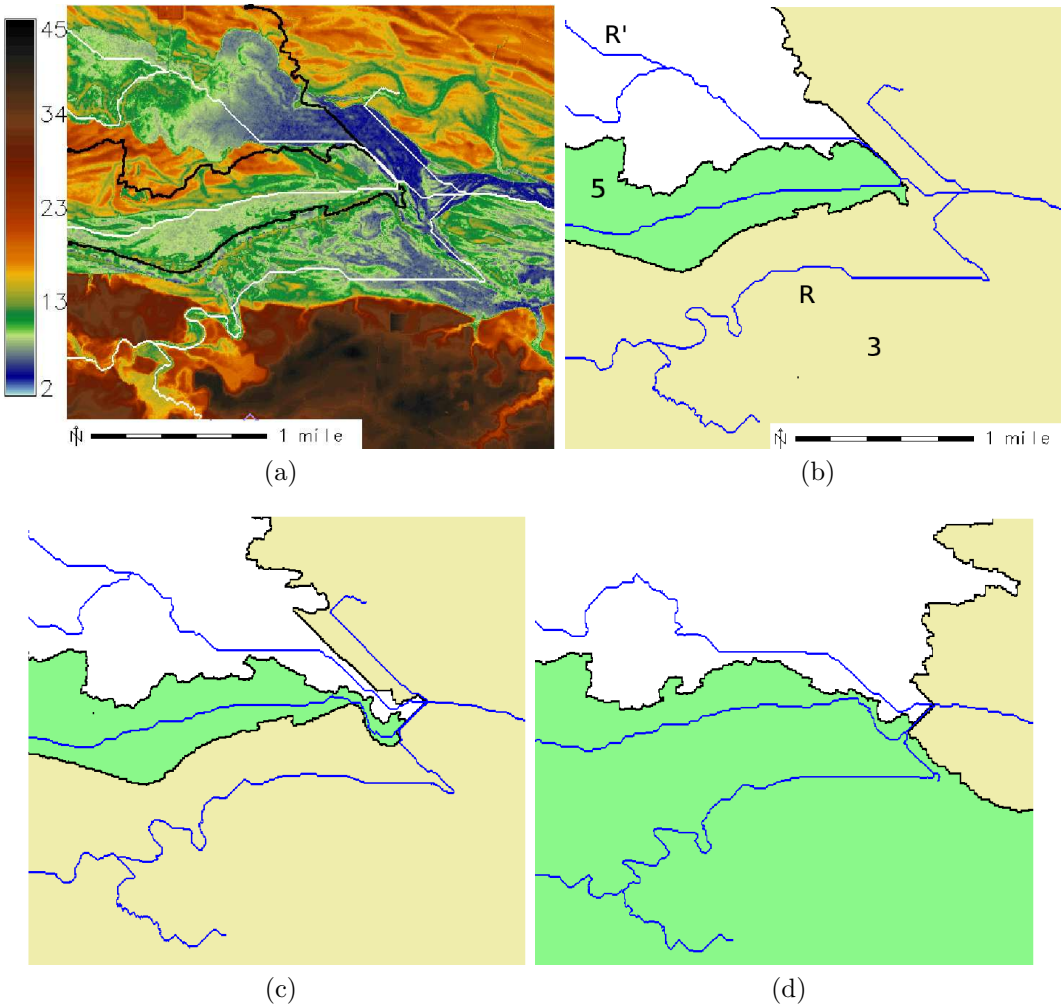


Figure 6.11: A detailed view of Figure 6.10. (a) Base terrain shown at 10ft resolution. Watershed boundaries at (b) 10ft, (c) 20ft, and (d) 40ft grid resolutions

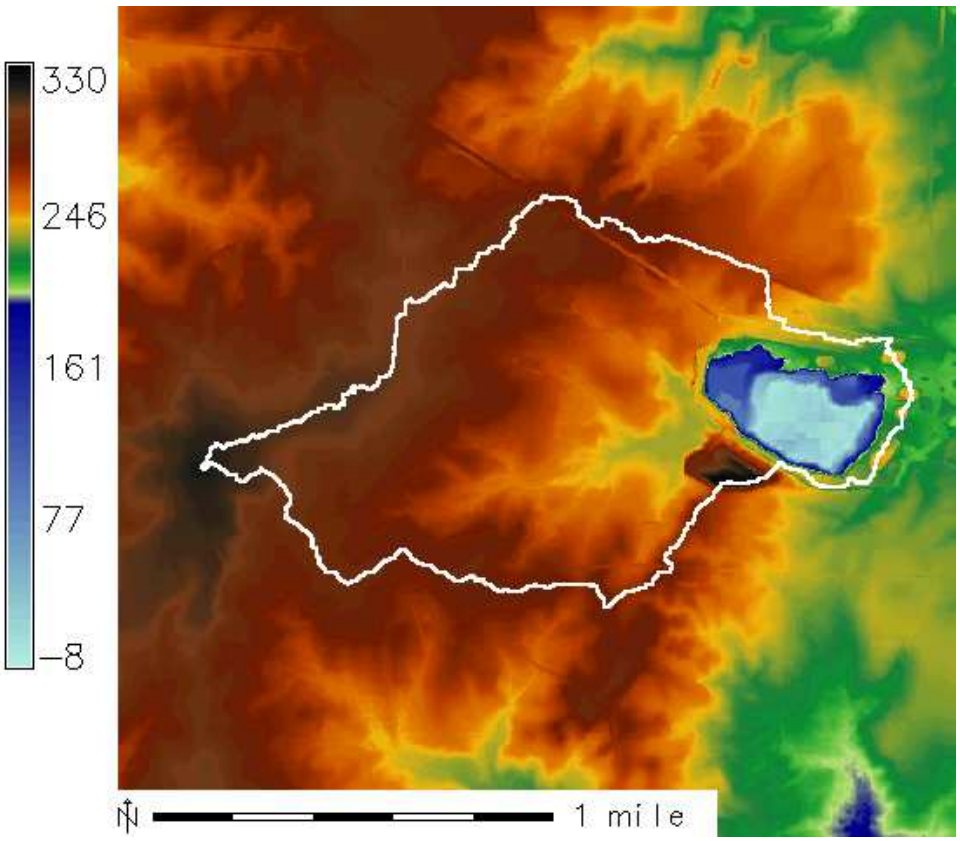


Figure 6.12: A quarry and its 600 acre watershed is preserved with a persistence threshold of 220 feet or less.

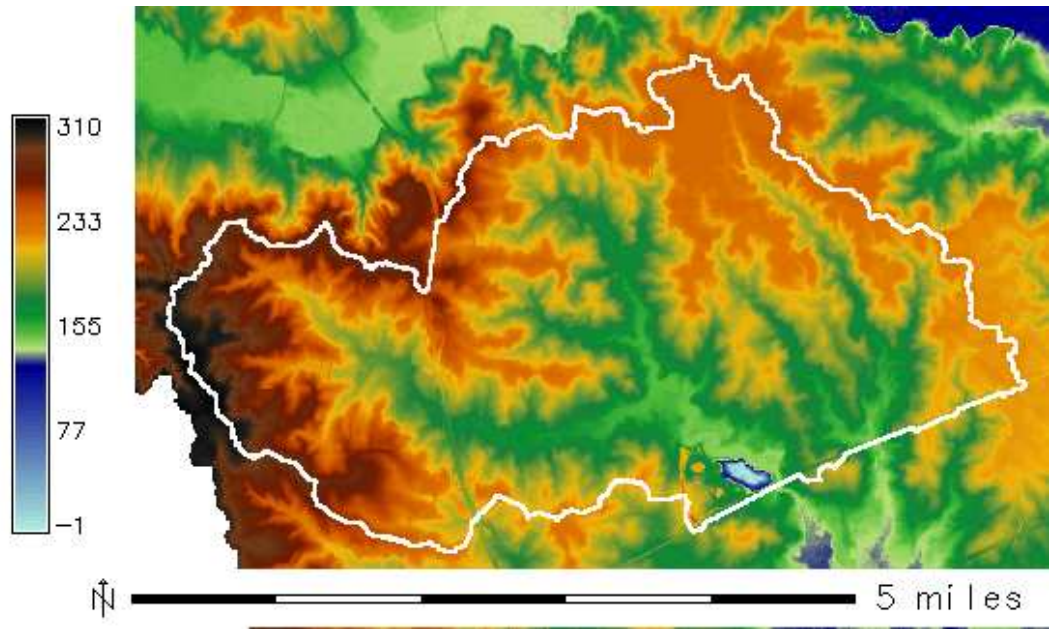
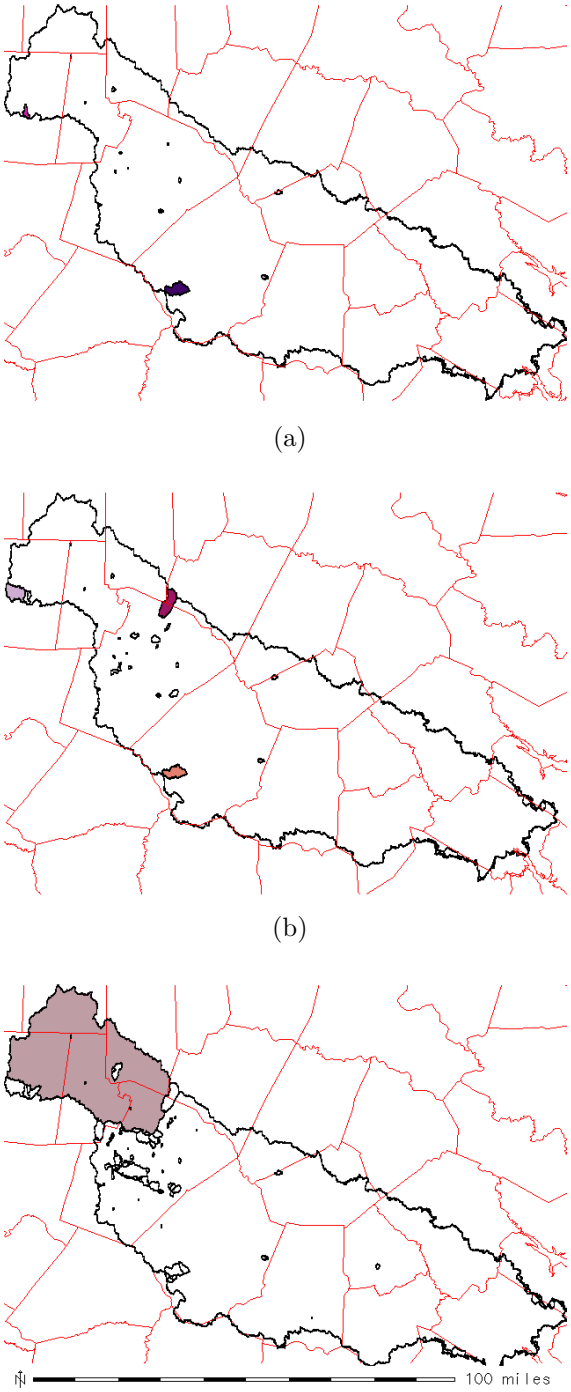


Figure 6.13: The largest (7300 acres) incorrectly computed closed basin, shown in white, for a persistence threshold of 50ft. A bridge in the southeast blocks flow.

As we lower the persistence threshold, more sinks are kept and fewer are removed. At a persistence threshold of 40ft, 13 additional sinks appear. The drainage area boundaries are shown in Figure 6.14(b). These additional sinks are all examples of small streams being blocked by bridges, but the drainage area of the additional sinks is small and does not dramatically effect the watershed boundaries or the river network. In particular, most of the Neuse river basin drains to a single outlet along the coast in the southeast corner of the figure. However, if we lower the persistence threshold to 30ft, we see dramatic changes in the number and drainage area of the preserved sinks as illustrated in Figure 6.14(c). The most obvious observation is that water upstream of the Falls lake dam, shown in pink shading, is disconnected from the rest of the basin. Also, many more minima appear, especially in urban areas such as Wake county, just South of the disconnected Falls lake basin. A brief inspection of a number of these sinks in Wake county revealed 62 total sinks, 47 of which were caused by bridges blocking rivers, 10 of which were around quarries and five whose source was not obvious.

Because a large portion of the Neuse river basin is detached from the main basin with a persistence threshold of 30ft, we expect the watershed boundaries to be significantly different for persistence thresholds of 50ft and 30ft. In Figure 6.15, we see that while basin 9 loses a significant fraction of its drainage area when lowering the persistence to 30ft, it is still larger than basin 8 and the ordering of the Pfafstetter basins (indicated by color) is unchanged. If a further downstream area lost a significant fraction of its total area by lowering the persistence threshold, re-ordering and re-labeling of basins would be much more likely.

If we look at the component that was cut off from the Neuse river basin in Figure 6.15, we can see by the watershed labels in Figure 6.16, that flow has been routed in an unrealistic way. Note that under the Pfafstetter label method, water in even numbered basins and basin 9 flow into lower numbered odd basins. The figure shows a region surrounded by even number basins plus basin nine. Thus, this region has no outlet to any other region. This is consistent with the observation that the region is disconnected from the main Neuse river basin in the terrain model. The odd numbered basins are tightly clustered in the center where a sink collects all of the water. If we looked closely



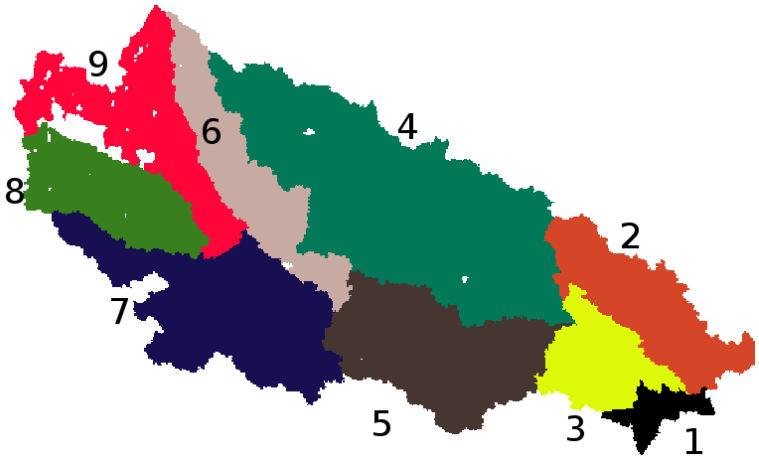
(a)

(b)

(c)

Figure 6.14: Drainage area boundaries of sinks shown in black with overlay of North Carolina county boundaries. A persistence threshold of (a) 50ft removes almost all sinks caused by bridges and creates one large primary basin. A threshold of (b) 40ft results in 28 remaining sinks, but the primary basin is intact. For a threshold of (c) 30 ft, the Neuse river basin becomes disconnected at the Falls lake dam (Northwest/shaded), and 96 sinks remain, most of which are due to bridges.

(a)



(b)

Figure 6.15: Pfafstetter basins for (a) persistence threshold of 50ft and (b) 30ft. Even though the headwaters are disconnected in the 30ft case, the ordering of the basin remains unchanged.

at the flow directions in this basin, we would see flow from basin 4 being directed Northwest towards the center, when in the real terrain water flows Southwest towards the Falls lake dam. Thus lowering the persistence below 30ft will not yield a good hydrologically conditioned DEM.

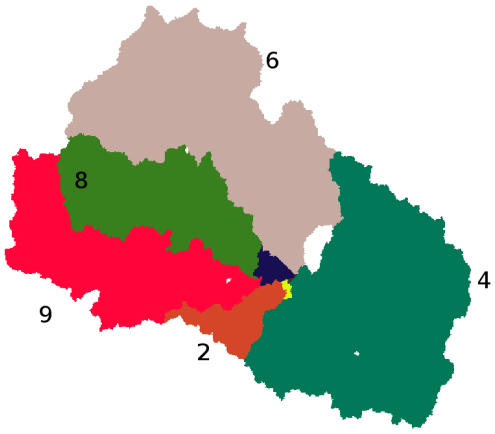


Figure 6.16: Watershed of Falls Lake area when persistence threshold is 20ft. Rivers computed in the southeast region eventually drain to a sink in the center of the image, instead of flowing under the dam which is to the the southeast

In the three persistence values tested in this section, we found that persistence can indeed be used to preserve real terrain features such as quarries, but that many bridges and an occasional dam create sinks with a moderately high persistence that should be removed. The gap in persistence values between the bridge with the highest persistence and the quarry with the lowest persistence is over 20ft. Thus, a persistence threshold of 55ft in this case study preserves all the quarries while routing flow across bridges. This new method of scoring and removing sinks below a threshold score could prove to be a valuable tool for many hydrological studies.

6.8 Conclusions

In this Chapter, we demonstrated that the algorithms presented in this thesis form a scalable and flexible pipeline that efficiently process massive amounts of data derived from modern remote sensing methods. Our primary emphasis in this thesis was on scalable algorithms, but we have seen that our tunable design allows us to explore interesting modeling issues as well. While lidar provides many potential benefits to the GIS community, our experiments highlighted the need for additional work in some areas. Bridges are particularly problematic for hydrological flow routing. We have seen in this Chapter that the topological persistence of most sinks blocked by bridges have a high persistence value, but this value is much lower than features such as quarries. We believe further improvements in th GIS modeling using topological persistence can help identify bridges effectively and lead to improved hydrological conditioning models that can automatically make small local cuts through bridges. This will significantly reduce the extent of terrain modification via flooding and dramatically reduce the size of flat areas.

Bridge removal is also important for accurate watershed extraction. As discussed in Section 6.6, subtle changes in the order of river mouths joining a main channel can significantly change watershed boundaries computed using the Pfafstetter method. Often times, odd flow routing paths are the result of poor flat routing models on areas that have been hydrologically conditioned by flooding sinks. Flooding sinks caused by bridges in not an ideal approach, but future work in bridge removal, hydrological conditioning alternatives to flooding, and improved flow routing on flat areas could help

significantly improve the quality of data derived from hi-resolution terrains. It is likely that many of these improved models can be incorporated into our flexible and scalable pipeline so that future model improvements can quickly be integrated into a framework that scales to massive modern elevation data sets.

Bibliography

[1] P. K. Agarwal, L. Arge, G. S. Brodal, and J. S. Vitter. I/O-efficient dynamic point location in monotone planar subdivisions. In *Proc. ACM-SIAM Symposium on Discrete Algorithms*, pages 1116–1127, 1999.

[2] P. K. Agarwal, L. Arge, and A. Danner. From point cloud to grid DEM: A scalable approach. In *Proc. International Symposium on Spatial Data Handling*, 2006.

[3] P. K. Agarwal, L. Arge, T. M. Murali, K. Varadarajan, and J. S. Vitter. I/O-efficient algorithms for contour line extraction and planar graph blocking. In *Proc. ACM-SIAM Symposium on Discrete Algorithms*, pages 117–126, 1998.

[4] P. K. Agarwal, L. Arge, O. Procopiuc, and J. S. Vitter. A framework for index bulk loading and dynamization. In *Proc. International Colloquium on Automata, Languages, and Programming*, pages 115–127, 2001.

[5] P. K. Agarwal, L. Arge, and K. Yi. I/O-efficient construction of constrained Delaunay triangulations. In *Proc. European Symposium on Algorithms*, pages 355–366, 2005.

[6] P. K. Agarwal, L. Arge, and K. Yi. I/O-efficient batched union-find and its applications to terrain analysis. In *Proc. 22nd Annu. ACM Sympos. Comput. Geom.*, 2006.

[7] A. Aggarwal and J. S. Vitter. The Input/Output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[8] Applied Imagery. http://www.appliedimagery.com, 5 March 2006.

[9] L. Arge. External memory data structures. In J. Abello, P. M. Pardalos, and M. G. C. Resende, editors, *Handbook of Massive Data Sets*, pages 313–358. Kluwer Academic Publishers, 2002.

[10] L. Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003.

[11] L. Arge, R. Barve, D. Hutchinson, O. Procopiuc, L. Toma, D. E. Vengroff, and R. Wickremesinghe. *TPIE User Manual and Reference (edition 082902)*. Duke University, 2002. The manual and software distribution are available on the web at `http://www.cs.duke.edu/TPIE/`.

[12] L. Arge, R. Barve, O. Procoiuc, L. Toma, D. E. Vengroff, and R. Wickremesinghe. *TPIE User Manual and Reference (edition 0.9.01a)*. Duke University, 1999. The manual and software distribution are available on the web at `http://www.cs.duke.edu/TPIE/`.

[13] L. Arge, G. S. Brodal, and L. Toma. On external memory MST, SSSP and multi-way planar graph separation. In *Proc. Scandinavian Workshop on Algorithms Theory, LNCS 1851*, pages 433–447, 2000.

[14] L. Arge, J. Chase, P. Halpin, L. Toma, D. Urban, J. S. Vitter, and R. Wickremesinghe. Flow computation on massive grid terrains. *GeoInformatica*, 7(4):283–313, 2003.

[15] L. Arge, A. Danner, H. Haverkort, and N. Zeh. I/O-efficient hierarchical watershed decomposition of grid terrain models. In *Proc. International Symposium on Spatial Data Handling*, 2006.

[16] L. Arge, A. Danner, and S. Teh. I/O-efficient point location using persistent B-trees. *The ACM Journal of Experimental Algorithmics*, 8, 2003.

[17] L. Arge, A. Danner, and S.-H. Teh. I/O-efficient point location using persistent B-trees. In *Proc. Workshop on Algorithm Engineering and Experimentation*, 2003.

[18] L. Arge, M. de Berg, H. J. Haverkort, and K. Yi. The priority R-tree: A practically efficient and worst-case optimal R-tree. In *Proc. SIGMOD International Conference on Management of Data*, 2004.

[19] L. Arge, K. H. Hinrichs, J. Vahrenhold, and J. S. Vitter. Efficient bulk operations on dynamic R-trees. *Algorithmica*, 33(1):104–128, 2002.

[20] L. Arge, U. Meyer, L. Toma, and N. Zeh. On external-memory planar depth first search. *Journal on Graph Algorithms and Applications*, 7(2):105–129, 2003.

[21] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter. Scalable sweeping-based spatial join. In *Proc. International Conference on Very Large Databases*, pages 570–581, 1998.

[22] L. Arge, O. Procopiuc, and J. S. Vitter. Implementing I/O-efficient data structures using TPIE. In *Proc. European Symposium on Algorithms*, pages 88–100, 2002.

[23] L. Arge, L. Toma, and J. S. Vitter. I/O-efficient algorithms for problems on grid-based terrains. *ACM Journal on Experimental Algorithmics*, 6(1), 2001.

[24] L. Arge, L. Toma, and N. Zeh. I/O-efficient topological sorting of planar dags. In *Proc. ACM Symposium on Parallel Algorithms and Architectures*, pages 85–93, 2003.

[25] L. Arge and J. Vahrenhold. I/O-efficient dynamic planar point location. *Computational Geometry: Theory & Applications*, 29(2):147–162, 2004.

[26] L. Arge, D. E. Vengroff, and J. S. Vitter. External-memory algorithms for processing line segments in geographic information systems. In *Proc. European Symposium on Algorithms, LNCS 979*, pages 295–310, 1995. To appear in special issues of *Algorithmica* on Geographical Information Systems.

[27] L. Arge, D. E. Vengroff, and J. S. Vitter. External-memory algorithms for processing line segments in geographic information systems. *Algorithmica*, 1998.

[28] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972.

[29] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion B-tree. *VLDB Journal*, 5(4):264–275, 1996.

[30] M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *Latin American Theoretical INformatics*, pages 88–94, 2000.

[31] G. S. Brodal and J. Katajainen. Worst-case efficient external-memory priority queues. In *Proc. Scandinavian Workshop on Algorithms Theory, LNCS 1432*, pages 107–118, 1998.

[32] M. J. Browne and R. E. Hoyt. The demand for flood insurance: Empirical evidence. *Journal of Risk and Uncertainty*, 20(3):291–306, 2000.

[33] A. L. Buchsbaum, M. Goldwasser, S. Venkatasubramanian, and J. R. Westbrook. On external memory graph traversal. In *Proc. ACM-SIAM Symposium on Discrete Algorithms*, pages 859–860, 2000.

[34] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proc. ACM-SIAM Symposium on Discrete Algorithms*, pages 139–149, 1995.

[35] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[36] A. Crauser, P. Ferragina, K. Mehlhorn, U. Meyer, and E. Ramos. Randomized external-memory algorithms for some geometric problems. *International Journal of Computational Geometry & Applications*, 11(3):305–337, June 2001.

[37] A. Crauser and K. Mehlhorn. LEDA-SM: Extending LEDA to secondary memory. In *Proc. Workshop on Algorithm Engineering*, 1999.

[38] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry – Algorithms and Applications*. Springer Verlag, Berlin, 1997.

[39] R. Dementiev, L. Kettner, and P. Sanders. Stxxl: Standard template library for XXL data sets. In *Proc. of the 13th European Symposium on Algorithms*, October 2005.

[40] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38:86–124, 1989.

[41] M. Edahiro, I. Kokubo, and T. Asano. A new point-location algorithm and its practical efficiency — Comparison with existing algorithms. *ACM Trans. Graph.*, 3(2):86–109, 1984.

[42] H. Edelsbrunner. *Geometry and Topology for Mesh Generation*. Cambridge University Press, England, 2001.

[43] H. Edelsbrunner, L. J. Guibas, and J. Stolfi. Optimal point location in a monotone subdivision. *SIAM J. Comput.*, 15(2):317–340, 1986.

[44] H. Edelsbrunner, J. Harer, and A. Zomorodian. Hierarchical morse complexes for piecewise linear 2-manifolds. In *Proc. 17th Annu. ACM Sympos. Comput. Geom.*, pages 70–79, 2001.

[45] H. Edelsbrunner, D. Letscher, and A. Zomorodian. Topological persistence and simplification. In *Proc. 41st IEEE Sympos. Found. Comput. Sci.*, pages 454–463, 2000.

[46] T. Freeman. Calculating catchment area with divergent flow based on a regular grid. *Computers and Geosciences*, 17:413–422, 1991.

[47] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. IEEE Symposium on Foundations of Computer Science*, pages 285–298, 1999.

[48] H. N. Gabow, J. L. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. In *Proc. of 16th ACM Symposium on Theory of Computing*, pages 135–143, 1984.

[49] J. Garbrecht and L. Martz. The assignment of drainage directions over flat surfaces in raster digital elevation models. *Journal of Hydrology*, 193:204–213, 1997.

[50] GDAL. http://remotesensing.org/gdal/, 12 Dec 2006.

[51] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *Proc. IEEE Symposium on Foundations of Computer Science*, pages 714–723, 1993.

[52] GRASS. Development Team, 2006. Geographic Resources Analysis Support System (GRASS) Software. ITC-irst, Trento, Italy http://grass.itc.it, 12 Dec 2006.

[53] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. SIGMOD International Conference on Management of Data*, pages 47–57, 1984.

[54] G. R. Hjaltason and H. Samet. Speeding up construction of quadtrees for spatial indexing. *VLDB*, 11(2):109–137, 2002.

[55] M. Hodgson, J. R. Jensen, L. Schmidt, S. Schill, and B. Davis. An evaluation of LIDAR- and IFSAR-derived digital elevation models in leaf-on conditions with USGS level 1 and level 2 DEMs. *Remote Sensing of the Environment*, 84:295–308, 2003.

[56] S. Huddleston and K. Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17:157–184, 1982.

[57] M. F. Hutchinson. A new procedure for gridding elevation and stream line data with automatic removal of pits. *Journal of Hydrology*, 106:211–232, 1989.

[58] J. R. Jensen. *Remote Sensing of the Environment: An Earth Resource Perspective*. Prentice Hall, 2000.

[59] S. Jenson and J. Domingue. Extracting topographic structure from digital elevation data for geographic information system analysis. *Photogrammetric Engineering and Remote Sensing*, 54(11):1593–1600, 1988.

[60] D. G. Kirkpatrick. Optimal search in planar subdivisions. *SIAM J. Comput.*, 12(1):28–35, 1983.

[61] V. Kumar and E. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proc. IEEE Symp. on Parallel and Distributed Processing*, pages 169–177, 1996.

[62] S. Lee, G. Wolberg, and S. Y. Shin. Scattered data interpolation with multilevel B-splines. *IEEE Transactions on Visualization and Computer Graphics*, 3(3):228–244, July–September 1997.

[63] A. S. Lowe. The Federal Emergency Management Agencys multi-hazard flood map modernization and the national map. *Photogrammetric Engineering & Remote Sensing*, 69(10):1133–1135, 2003.

[64] L. Martz and J. Garbrecht. An outlet breaching algorithm for the treatment of closed depressions in a raster DEM. *Computers and Geosciences*, 25(7):835–844, 1999.

[65] L. Mitas and H. Mitasova. Spatial interpolation. In P. Longley, M. F. Goodchild, D. J. Maguire, and D. W. Rhind, editors, *Geographic Information Systems - Principles, Techniques, Management, and Applications*. Wiley, 1999.

[66] H. Mitasova and L. Mitas. Interpolation by regularized spline with tension: I. theory and implementation. *Mathematical Geology*, 25:641–655, 1993.

[67] H. Mitasova, L. Mitas, W. M. Brown, D. P. Gerdes, I. Kosinovsky, and T. Baker. Modelling spatially and temporally distributed phenomena: new methods and tools for GRASS GIS. *Int. J. Geographical Information Systems*, 9(4):433–446, 1995.

[68] H. Mitasova, L. Mitas, and R. S. Harmon. Simultaneous spline interpolation and topographic analysis for lidar elevation data: methods for open source gis. *IEEE Geoscience and Remote Sensing Letters*, 2(4):375–379, 2005.

[69] NC-Floodmaps. http://www.ncfloodmaps.com, 12 Dec 2006.

[70] M. Neteler and H. Mitasova. *Open source GIS: A GRASS GIS Approach*. Kluwer Academic / Plenum Publishers, New York, 2004.

[71] NOAA-CSC. LIDAR Data Retrieval Tool-LDART http://maps.csc.noaa.gov/tcm, 12 Dec 2006.

[72] J. F. O'Callaghan and D. M. Mark. The extraction of drainage networks from digital elevation data. *Computer Vision, Graphics and Image Processing*, 28, 1984.

[73] T. K. Peucker. Detection of surface specific points by local parallel processing of discrete terrain elevation data. *Computer Graphics and Image Processing*, 4:375–387, 1975.

[74] J. Pouderoux, I. Tobor, J.-C. Gonzato, and P. Guitton. Adaptive hierarchical RBF interpolation for creating smooth digital elevation models. In *ACM-GIS*, pages 232–240, November 2004.

[75] O. Procopiuc, P. K. Agarwal, L. Arge, and J. S. Vitter. Bkd-tree: A dynamic scalable kd-tree. In *Proc. International Symposium on Spatial and Temporal Databases*, 2003.

[76] N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29:669–679, 1986.

[77] P. Seaber, F. Kapinos, and G. Knapp. Hydrologic unit maps, USGS water supply paper 2294, 63 p., 1987.

[78] R. Sibson. A brief description of natural neighbor interpolation. In V. Barnett, editor, *Interpreting Multivariate Data*, pages 21–36. John Wiley and Sons, 1982.

[79] J. Snoeyink. Point location. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 30, pages 559–574. CRC Press LLC, Boca Raton, FL, 1997.

[80] P. Soille. Optimal removal of spurious pits in grid digital elevation models. *Water Resources Research*, 40(12), 2004.

[81] P. Soille, J. Vogt, and R. Colombo. Carving and adaptive drainage enforcement of grid digital elevation models. *Water Resources Research*, 39(12):1366–1375, 2003.

[82] D. Tarboton. A new method for the determination of flow directions and contributing areas in grid digital elevation models. *Water Resources Research*, 33:309–319, 1997.

[83] *TIGER/Line*™ *Files, 1997 Technical Documentation*. Washington, DC, September 1998. http://www.census.gov/geo/tiger/TIGER97D.pdf.

[84] L. Toma. *External Memory Graph Algorithms and Applications to Geographic Information Systems*. PhD thesis, Duke University, 2003.

[85] U.S. Geological Survey. 1:100,000-scale line graphs (DLG). Accessible via URL. http://edc.usgs.gov/geodata/ (Accessed 12 December 2006).

[86] J. Vahrenhold and K. H. Hinrichs. Planar point location for large data sets: To seek or not to seek. In *Proc. Workshop on Algorithm Engineering, LNCS 1982*, pages 184–194, 2001.

[87] J. van den Bercken, B. Seeger, and P. Widmayer. A generic approach to bulk loading multidimensional index structures. In *Proc. International Conference on Very Large Databases*, pages 406–415, 1997.

[88] P. J. Varman and R. M. Verma. An efficient multiversion access structure. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):391–409, 1997.

[89] D. E. Vengroff. A transparent parallel I/O environment. In *Proc. DAGS Symposium on Parallel Computation*, 1994.

[90] K. L. Verdin and J. P. Verdin. A topological system for delineation and codification of the Earth's river basins. *Journal of Hydrology*, 218:1–12, 1999.

[91] J. S. Vitter. External memory algorithms and data structures: Dealing with MASSIVE data. *ACM Computing Surveys*, 33(2):209–271, 2001.

[92] J. Vuillemin. A unifying look at data structures. *Commun. ACM*, 23:229–239, 1980.

[93] H. Wendland. Fast evaluation of radial basis functions: Methods based on partition of unity. In C. K. Chui, L. L. Schumaker, and J. Stöckler, editors, *Approximation Theory X: Wavelets, Splines, and Applications*, pages 473–483. Vanderbilt University Press, Nashville, 2002.

[94] D. Wolock and G. McCabe. Comparison of single and multiple flow direction algorithms for computing topographic parameters in topmodel. *Water Resources Research*, 31:1315–1324, 1995.

[95] K. Yi. *I/O-Efficient Algorithms for Processing Massive Spatial Data*. PhD thesis, Department of Computer Science, Duke University, 2006.

Biography

Born

Pittsburgh, Pennsylvania, 4 May 1977

Colleges and Universities

Duke University **Durham, NC**
Ph.D. in Computer Science, December 2006.
M.S. in Computer Science, September 2004.

University of North Carolina **Chapel Hill, NC**
Graduate student in Physics, August 1999 – May 2000.

Gettysburg College **Gettysburg, PA**
B.S. in Physics and Mathematics, May 1999.

Publications

"From Point Cloud to Grid DEM: A Scalable Approach." Pankaj K. Agarwal, Lars Arge, and Andrew Danner. In *Proc. International Symposium on Spatial Data Handling*, 2006.

"I/O-Efficient Hierarchical Watershed Decomposition of Grid Terrain Models." Lars Arge, Andrew Danner, Herman Haverkort, and Norbert Zeh. In *Proc. International Symposium on Spatial Data Handling*, 2006.

"Computing Pfafstetter labellings I/O-efficiently." Lars Arge, Andrew Danner, Herman Haverkort, and Norbert Zeh. In *Münster University, Dept. of Computer Science, technical report 02/05-I*, 2005.

"I/O-efficient point location using persistent B-trees." Lars Arge, Andrew Danner, and Sha-Mayn Teh. *The ACM Journal of Experimental Algorithmics*, 8, 2003.

"Cache-oblivious data structures for orthogonal range searching." Pankaj K. Agarwal, Lars Arge, Andrew Danner, and Bryan Holland-Minkley. In *Proc. ACM Symposium on Computational Geometry*, pages 237–245, 2003.

"'I/O-efficient point location using persistent B-trees." Lars Arge, Andrew Danner, and Sha-Mayn Teh. In *Proc. Workshop on Algorithm Engineering and Experimentation*, 2003.