Handling Massive Terrains and Unreliable Memory

Thomas Mølhave

PhD Dissertation

Department of Computer Science
Aarhus University
Denmark

Handling Massive Terrains and Unreliable Memory

A Dissertation
Presented to the Faculty of Science
of Aarhus University
in Partial Fulfilment of the Requirements for the
PhD Degree

by
Thomas Mølhave
August 14, 2009

Abstract

Recent technological advances have greatly increased the ability to acquire, store, and analyze data. These developments have significantly improved the potential of many commercial and scientific applications, and lead to many new scientific discoveries. We are growing accustomed to accessing massive amounts of information from almost anywhere using devices ranging from cell phones and tiny GPS navigation systems, to ordinary computers and beyond. The large amount of information available presents a number of problems and opportunities. One of the main obstacles is most software is not designed to handle large amounts of data, resulting in crashes or running for a very long time on even moderately-sized data sets.

Another problem is contemporary memory devices can be unreliable due to a number of factors, such as power failures, radiation, and cosmic rays. The content of a cell in unreliable memory can be silently altered and this can adversely affect most traditional algorithms.

The focus of this dissertation is on the algorithms and data structures specifically designed for solving a number of the problems involving large data sets and unreliable memory devices. The dissertation is divided into two parts.

In Part I we use the classical external memory model by Aggarwal and Vitter, and the cache-oblivious model recently proposed by Frigo et al., to design cache-efficient algorithms. We focus on problems involving terrain models which, due to modern terrestrial scanning techniques, can be very large. We present the TerraStream software package, which solves many common computational problems on big terrains. We also present an I/O-efficient algorithm for computing contour maps of a terrain and a cache-oblivious algorithm for finding intersections between two sets of internally non-intersecting line segments.

In Part II we use the faulty memory RAM, proposed by Finocchi and Italiano, to model unreliable memory circuits and design algorithms that are resilient to memory faults. We present a resilient priority as well as an optimal comparison-based resilient algorithm for searching in a sorted array. We also show how to use this algorithm to get a dynamic resilient dictionary. Finally, we present a model that combines the standard external memory model with the faulty memory RAM and present lower and upper bounds for I/O-efficient resilient dictionaries, an I/O-efficient resilient sorting algorithm and an I/O-efficient resilient priority queue.

Acknowledgements

I want to thank my advisor, Lars Arge, for four years of serious work, discussions about life, and lots of fun inside and outside the walls of the university as well as at various soccer stadiums. I also want to thank Pankaj Agarwal for hosting and working with me during my 2007 visit to Duke University, and Adam Buchsbaum for hosting and working with me during my summer internship at AT&T Research Labs in 2007. My visits to AT&T and Duke meant a lot to me personally and professionally, and I am grateful for the opportunity it gave me to explore a new country and gain many new friends. Also, I want to thank Ivan Damgård, Ian Munro and Ulrich Meyer for agreeing to be on my PhD committee.

My papers have not been written in isolation and I would like to express my sincere gratitude to my coauthors that have worked with me through the years; they have all taught me a lot about the art of computer science and the craftsmanship required to write good scientific papers. First I want to thank the coauthors of all my currently published papers and those that form the backbone of this dissertation: Lars Arge, Gerth Brodal, Norbert Zeh, Allan Jørgensen, Ke Yi, Helena Mitasova, Bardia Sadri, Pankaj Agarwal, Rolf Fagerberg, Gabriel Moruz, Fabrizio Grandoni, Irene Finocchi and Giuseppe Italiano. I would also like to thank my coauthors on those of my papers that are still in preparation and those that are under review: Lasse Deleuran, Kasper Larsen, Morten Revsbæk, Jesper Eshøj, Peter Bøcher and Jens-Christian Svenning.

I want to thank my good friends and office mates at Aarhus University, in particular: Lars Petersen, Mikkel Krøigaard, Martin Geisler, Rune Thorbek, Allan Grøndlund Jørgensen, Gabriel Moruz, Christian Schaffner, Henrik Blunck, Doina Bucur, Morten Revsbæk, Jakob Truelsen and Kasper Larsen. I also want to thank my friends from Duke University and those I met during my visit to AT&T for their great company. To list a few: Jeff Phillips, Andrew Danner, Bei Wang, Ashley Flavel, Svetlana Yarosh and Maritza Johnson. I also want to thank our very talented programmers Thor Prentow and Adam Thomsen who have helped tremendously in making TerraStream user-friendly, stable and powerful.

Additionally I want to thank Lars Arge, Henrik Blunck, Gerth Brodal, Martin Geisler, Shannon Glutting, Allan Grønlund Jørgensen and Thor Prentow for providing valuable feedback on early drafts of this dissertation.

Finally I want to thank my mom, Birgit, and my dad, Anders as well as my sister, Helle, for their constant love, support and encouragement during my upbringing and my eight years in Aarhus. I want to thank Shannon for her love and support, and for showing me how much fun life outside computer science can be.

Thomas Mølhave,
Århus, August 14, 2009.

Contents

Chapter 1

Introduction

In recent years our society has become very "data driven" — we are collecting truly massive amounts of data. In 2003, a Berkeley study [83] concluded that the amount of information gathered in a few years at the turn of the century was more than the total amount of information hitherto produced by the human race. Our increased ability to acquire, store, and analyze data has significantly improved the potential of many commercial and scientific applications, and has lead to many new scientific discoveries. We are also growing accustomed to seamlessly accessing massive amounts of information from almost anywhere using devices ranging from cell phones and tiny GPS navigation systems, to ordinary computers and beyond.

For many scientists, the access to vast amounts of information presents a number of problems and opportunities. In 2008, the journal Nature published a special issue entitled "Big Data," in recognition of these [90]. Besides discussing how to manage and store information so that it can both be used immediately and archived for posterity, the special issue also notes that the art of analyzing the information is getting increasingly important [70]. In some sense, the mindset of many scientists has switched from one where information *gathering* plays a major role to one where information *analysis* is the primary concern. Many new discoveries may be made from the large amounts of data available from different sensors if the information is properly managed.

This situation is similar in the commercial world. Many companies routinely gather massive amounts of information, sometimes with no immediate application in mind. Examples of big commercial data gathering operations include the calling records for major phone companies and Wal-Mart's enormous data base of purchase information. Further examples are the big databases handled by search engines and the transaction logs of banks. Additionally, recent revolutionary improvements in remote sensing are rapidly expanding the quality and quantity of geographical data. In particular, laser altimetry (lidar) gathers georeferenced *elevation* data at unprecedented resolutions, and rates, and enables potentially critically important applications, such as environmental and disaster management.

The growing volume of information is not only beneficial for manufacturers of storage devices, it also creates a growing demand for better and smarter data analysis tools. Creating good data analysis tools involves many aspects from

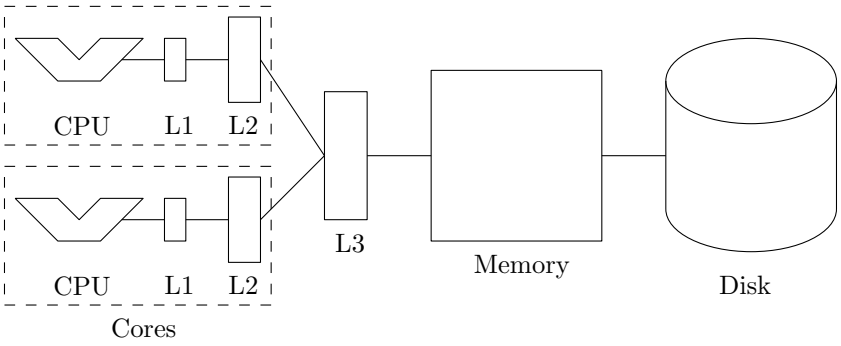


Figure 1.1: A simplified overview of some of the levels present in the cache hierarchy of a typical desktop machine. In this situation each core has a separate L1 and L2 cache and shares a L3 cache with the other core.

computer science such as visualisation, software design, user friendliness, data warehousing, as well as algorithms and data structures. A lot of domain specific knowledge is required in most cases as well, but, the focus of this dissertation is on the algorithms and data structures specifically designed for solving a number of problems involving large data sets.

We start in Section 1.1 with a short introduction to the aspects of modern hardware relevant for the content in this dissertation. Subsequently, we explain why these hardware properties cause problems with standard models of computations and thereby inspire many new models of computation. In Section 1.2, we introduce some of the different types of models and algorithmic design techniques proposed with these hardware limitations in mind. We end the chapter in Section 1.3 with an overview of the contributions presented in this dissertation and a description of the papers where these results have been previously published.

1.1 Properties of Cache-Systems

This dissertation focuses on algorithms and data structures in models where certain aspects of the storage area used by the algorithm is explicitly modelled. In this section we describe some of these properties in order to better understand the models that we present later.

1.1.1 Memory hierarchies

While the availability of high-quality data has enabled many applications, the ever increasing size of manipulated data (or in some instances the decreasing size of computing devices) has also exposed scalability problems with existing systems. One main reason for these issues is, that algorithms in current systems do not take the memory organization of modern machines (it being hand-held devices, desktop machines, or super-computers) into account. One of the essential features of modern memory systems is that they are made up of a hierarchy of several levels [113]. The main reason for this is that price per unit

of storage increases dramatically as the performance characteristics improve. In this section we consider the cache hierarchy of a typical contemporary desktop computer. This hierarchy is depicted in Figure 1.1.

The lowest level, level one (denoted L1), is the fastest, smallest level. The next level (L2) is bigger, but also a bit slower. A modern general purpose processor can also have a third level of cache (L3) which is larger still, but also slower than the first two levels. The next level is usually the main internal memory of the computer, which is significantly bigger than the preceding caches but also, unsurprisingly, slower. Finally, most computers have some form of hard drive as the last level of caching. The hard drive is also the only level that holds its content when the computer is powered down.

At the time of writing, the newest high-end consumer processor is the Intel®Core™ i7. The processor has four cores which is increased to eight virtual cores using Intel's HyperThreading technology. Each of the four cores have a private L1 cache size, which is split between an instruction cache and a data cache getting 32 kilobytes[1] each [72]. Each core also has an L2 cache, which is 256 kilobytes large. Finally, all four cores share an eight megabyte L3 cache. The Core i7 is usually fitted in consumer and gaming computers with two to eight gigabytes of external memory and a few terabytes of hard disk space, depending on drive technology.

The two main performance criteria for levels in a memory hierarchy are *latency* and *bandwidth* [113]. Roughly speaking, the latency of a level is the time it takes from a request is made until the requested elements are being received at the destination. The bandwidth is the number of elements that can be transferred per unit of time. Since the latency can be significant, elements are transferred in fixed-size units, typically called *cache lines* or *blocks* depending on the type of level. In this dissertation we will use the term *block* regardless of the involved levels. The size of these units vary, depending on the particular type of cache, from a few bytes in the lower levels to several kilobytes or even megabytes for hard drives. By transferring many elements per cache access, the access latency can be amortized among all the retrieved elements if they are accessed before they are evicted. Some of the computational models used in this dissertation, and presented in Section 1.2, analyze how good an algorithm is at exploiting this locality in order to minimize block accesses. The latency for on-core caches, like the L1 and L2 cache on the Core i7, are usually measured in clock cycles since they run in sync with the CPU, and possibly with a slightly lower rate as specified by some multiplier tuning parameter. The access latency for the higher levels of cache and the main memory is measured in nanoseconds [113].

1.1.2 Disk drives

The cost of access latencies is especially noticeable on the level between main memory and a hard drive. A standard hard drive consists of a few drive *platters*, which are discs storing the actual bits using the magnetic properties of the

[1] In this dissertation a kilobyte is 1024 bytes, a megabyte is 1024^2 bytes, a gigabyte is 1024^3 bytes and so on.

platter material. The platters rotate at a speed between 4.200 rounds per minute for laptop drives up to 15.000 rounds per minute for high-end server drives. Data is read and written by drive *heads* that move along an axis perpendicular to the platters. The drive head reads (resp. writes) information on the platters by reading (resp. altering) the magnetic properties of the parts of the spinning platter passing underneath it.

The latency of a hard drive is bounded by its *seek time*, which is the time it takes for the drive head to be placed in the right position and for the part of the drive platters containing the requested data to rotate to a position underneath the drive head. Due to these mechanics, the latency of a typical hard drive is measured in milliseconds and is very high compared to the compact memory circuitry used in the main memory and CPU cache levels. Thus, the latency of hard drives is about a million times higher, from nanoseconds to milliseconds, than that of main memory. The bandwidth of the drive depends, among other things, on the speed in which the platters rotate and on the density of the data on the platter. It is in the order of a few hundred megabytes per second for standard consumer level drives. Due to the nature of the drive heads and the spinning platters, there are no big performance difference between sequential read and sequential write operations.

Recently, hard drive manufacturers started making *solid state discs (SSDs)* based on flash memory chips. These drives have properties that are different from standard hard drives. Solid state drives are still rather expensive and are not yet available with capacities matching the standard disk drives. However, the solid state drive technology is rapidly developing, and many high-end consumer laptops are now available with a solid state drive instead of the standard hard drive. The advantages of the solid state drives are many-fold. For instance, since they contain no moving mechanical parts they are much more durable than the relatively fragile hard drives, and they also consume less power [9]. Solid state drives are also interesting because they provide faster access times than regular hard drives, and also because the latency and bandwidth for read operations is better than for write operations. Since there are no spinning platters and moving disk heads the solid state drives can also perform random accesses significantly faster than regular disk drives. We refer to a recent survey by Ajwani et al. [9] for an in-depth discussion of the performance characteristics of solid state drives.

1.1.3 Cache reliability

Performance is only one of several different properties of a cache system. In this dissertation we also study how we can design algorithms that perform in a relatively well defined manner if the cache used is unreliable.

Modern memory chips are made from increasingly smaller and complicated circuits that work at low voltage levels and offer large storage capacities [45]. Unfortunately, these improvements have increased the likelihood of *soft memory errors* where arbitrary bits flip, corrupting the contents of the affected memory cells [110]. Soft memory errors can be triggered by phenomena such as power failures, cosmic rays and manufacturing defects. The rate of soft memory errors is predicted to increase in the future [28]. Furthermore, since the amount of

cosmic rays increase dramatically with the altitude, soft memory errors are a serious concern in fields like avionics and space research. We refer to Petrillo et al. [94] for more information about soft memory errors and the mean time between failures of memory circuitry.

Even though the occurrence rate of soft memory errors in individual memories is low they are a serious concern in applications running on clusters where the frequency of these errors is significantly larger. Similarly, applications running on massive data sets are more likely to encounter memory corruptions since they run for longer amounts of time.

1.2 Models of Computation

The traditional models used for designing algorithms do not consider the cost of different cache levels and the risk that the cache is unreliable. For instance, in the standard Random Access Machine (RAM) all memory accesses are assigned the same cost. From the discussion in the previous section, it follows that this assumption does not hold in real machines where the cost depends on the specific level in a multi-level hierarchy that contains the requested element. Furthermore, subsequent accesses to other elements in the same block can be very cheap if the block has not yet been evicted from the cache. In this section we present models of computation and algorithmic design techniques that are useful when trying to reduce the time spent waiting for the caches and we also present a recently proposed model for dealing with systems that has unreliable caches.

In Section 1.2.1 we give a brief overview of techniques used to limit the size of the working space needed for algorithms and the size of data structures. If the size of this information can be kept down, it has a bigger chance of fitting in the fast lower level caches. Next, in Section 1.2.2 we present the most heavily used model in this dissertation: the external memory model. In this model the block transfers between two consecutive levels of the memory hierarchy are modelled explicitly, and the complexity measure is the number of block transfers between these levels. Subsequently, we present the cache-oblivious model in Section 1.2.3. This model is an interesting generalization of the external memory model that can be used to understand the performance of algorithms across all the levels of a memory hierarchy. A novel model that resembles the external memory model, but in a setting with solid state drives, is presented in Section 1.2.4 Finally, in Section 1.2.5 we discuss the faulty memory RAM, which we use to design algorithms that are, to some extent, resilient to soft memory errors.

1.2.1 Minimizing space usage

One way to cope with large amounts of data is to design algorithms and data structures that use very little space. If the space usage is low, the amount of data stored on the slow higher-level caches can be limited .

There has a been a lot of work in this area and many different models and approaches have been used. The size of the data produced by can be reduced greatly with the use of general purpose compression algorithms, but the resulting

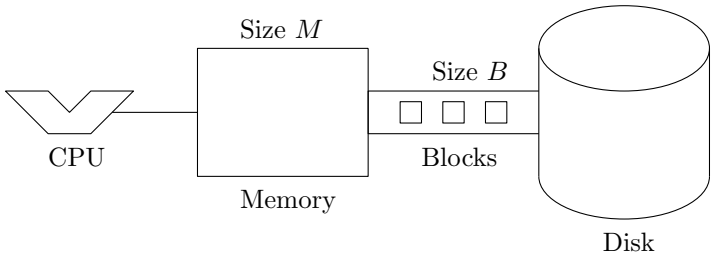


Figure 1.2: The external memory model [6]. The CPU operates on the M elements in the memory and elements are moved between disk and memory in blocks of B consecutive elements.

compressed files often have to be decompressed before any non-trivial operation on the data can be performed.

Many commonly known algorithms work directly on the input without using space for more than a constant number of extra elements. Algorithms that work in this manner are called *in-place* (or *in situ*). For instance, bubble sort and heap sort are both in-place sorting algorithms that permute the elements directly in the input buffer while using only $O(1)$ additional words to store extra variables. There has also been some work on designing in-place algorithms that are also efficient in terms of cache usage [62, 63].

For data structures there has been a lot of work on so-called *succinct* data structures. Succinct data structures store a set of data while supporting a certain set of operations using a number of bits very close to the information theoretically lower bound on the space required for the particular problem being considered. For more information on succinct representation of data structures we refer to a survey by Munro and Rao [88].

1.2.2 External-memory model

As discussed, data is transferred between levels in large blocks to amortize the large difference in the latency of different levels. This implies that it is important to design algorithms with a high degree of locality in their memory access pattern, that is, algorithms where data accessed close in time is also stored close in memory. A local access pattern is important when dealing with massive data sets — especially when data cannot reside in the main memory. The Input-Output (I/O) communication between internal memory and disks often becomes a severe bottleneck due to the huge difference in access time of these two levels.

In the two-level *external-memory model* [6], the memory hierarchy consists of an *internal memory* with a capacity of M elements and an arbitrarily large *external memory* partitioned into blocks of B elements (Figure 1.2). A *memory transfer* moves one block between internal and external memory. Computation can occur only on data in internal memory. The complexity of an algorithm in this model (an *external-memory algorithm*) is measured in terms of the number of memory transfers it performs. For instance, the complexity of a linear scan of N elements stored consecutively in the external memory is $O(\text{Scan}(N)) = O(N/B)$

disk transfers. Aggarwal and Vitter proved that the number of memory transfers needed for comparison-based sorting of N data items in the external-memory model is $\text{Sort}(N) = \Theta(\frac{N}{B} \log_{M/B} \frac{N}{B})$ [6]. Subsequently, a large number of algorithms have been developed in this model (see [10, 118] for overviews). The external memory model is commonly called the *I/O model* and we will use both terms interchangeably in this dissertation.

1.2.3 Cache-oblivious model

While I/O-efficient algorithms can also be used on other levels than the level between the disk and the main memory, it is often complicated to do so. This is partly caused by the many different memory hierarchy architectures with different and complicated inter-level interactions. Additionally, I/O-efficient algorithms only work efficiently on *one* particular level of the hierarchy, which implies that the user of the algorithm must know in advance which particular level of a multi-level hierarchy is the main bottleneck and tune the algorithm for that level specifically. However, Frigo et al. [64] recently proposed the so-called *cache-oblivious* algorithm design technique which can be used to design algorithms that are efficient on any memory hierarchy and on all levels simultaneously. In the cache-oblivious model, algorithms are *designed* in the standard RAM-model with no knowledge of the parameters of the memory hierarchy. However, the algorithms are *analyzed* in the external-memory model assuming that an offline optimal paging strategy performs the memory transfers necessary to bring accessed elements into memory. Often it is also assumed that $M \geq B^2$ (the *tall-cache assumption*). Since cache-oblivious algorithms are designed without knowledge of the parameters of any particular memory hierarchy, and the analysis is performed for arbitrary values of M and B, the model allows us to design algorithms that are efficient on *any* multi-level memory hierarchy (see [64] for details).

Frigo et al. [64] developed optimal cache-oblivious sorting algorithms, as well as algorithms for a number of other fundamental problems. Subsequently, algorithms and data structures for a range of problems [15] were developed.

1.2.4 Modelling solid state drives

As discussed in Section 1.1 solid state drives differ from standard hard drives in a couple of places, one being the different read and write performance. Since the external memory model does not capture this asymmetry, the emerging availability of affordable flash drives has prompted the development of two computational models tailored specifically for these devices. The two models, proposed by Ajwani et al. [8], are the *general flash model* and the *unit cost flash model*.

Like the external memory model, the general flash model consists of an infinite size disk, a memory with a capacity of M elements and a CPU that can operate on the elements in memory only. Unlike the I/O model, the general flash model has two block sizes. A read operation transfers B_r elements from disk to the memory and a write operation transfers B_w elements from the memory

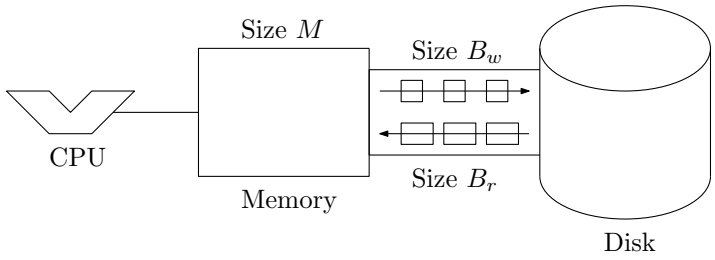


Figure 1.3: The flash memory models [8]. Both the general and the unit cost model have a read block size B_r and a write block size B_w.

to disk. The complexity of an algorithm in this model is $r + cw$, where r is the number of read operations performed, w is the number of write operations performed and c is the *write penalty*. The different block sizes capture the fact that the read and write latencies on a solid state drive differ, and the write penalty is meant to capture the ratio between the bandwidth (or throughput) for sequential read and sequential write operations. It is generally the case that $B_r < B_w$ [9]. Note the I/O model is equivalent to the general flash model with $B = B_r = B_w$ and $c = 1$. By setting c close to zero write operations become almost free, whereas a high value of c models a situation where little more than the output can be written due to the high costs of write operations.

Because it permits an arbitrary value of c, the general flash model is complicated and it is hard to get interesting results for extreme values of c. In practice, the bandwidth for sequential reads and sequential writes differ only by a small constant factor, so Ajwani et al. [8] also propose the much simpler unit cost flash model. The model has a read and write block size like the general model, but the complexity of an algorithm in the unit cost model is simply the number of elements transferred: $rB_r + wB_w$. This implies that algorithms can still use the fact that the block sizes differ to optimize how random I/Os are performed, but the sequential throughputs of read and write operations are the same. The unit cost flash model resembles the general flash model with $c = B_w/B_r$.

1.2.5 Faulty memory RAM

Corrupted memory cells can have important negative consequences for the output of algorithms. For instance, a single corruption in a sorted array can force a standard binary search to end up $\Omega(n)$ cells away from the correct position, and a single corruption in the wrong place of a sequence during a merge sort can lead to $\Theta(n^2)$ inversions in the final output. Soft memory errors can also be exploited to break the security of software systems. This was demonstrated in work breaking Java Virtual Machines [67], cryptographic protocols [26, 31, 121] and smart-cards [104].

The risk of soft memory errors can be reduced by using replication and error correcting codes at the hardware level, however, this approach is not always popular since the increased circuitry requirements are costly with respect to performance, storage capacity and money. As an extreme case, the triple modular redundancy technique, which stores three identical copies of each circuit

3	4	34	10	12	13	18	21	14	23	25	29	31	32	35	41	47
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Figure 1.4: A faithfully ordered sequence containing two cells with corrupted content.

with their outputs fed to a majority voting mechanism, induces more than 200% overhead in terms of chip area and power [120]. Even though this reduces the number of soft errors significantly, they can still occur.

In software, memory errors have been addressed in a variety of settings with the main focus on ensuring that code runs as expected: anticipating critical errors caused by hardware malfunctions and malicious attacks. Errors are detected using techniques such as algorithm-based fault tolerance [71], assertions [99], control flow checking [1, 122], procedure duplication [96] and employing randomization to automatically correct heap-based memory errors [92]. Recent work focuses on repairing data structures using specifications and assertions [49, 54].

Most algorithms and data structures, however, assume their storage is perfectly reliable, however, several algorithms dealing with unreliable information have been proposed. These include fault-tolerant pointer-based data structures [25], the liar model [32, 97], fault-tolerant sorting networks [80], fault-tolerant parallel models [43, 44] and locally mendable distributed networks [78].

Finocchi and Italiano [60] introduced the *faulty-memory random access machine* based on the traditional RAM model. In this model, memory corruptions can occur at any time and place in memory during the execution of an algorithm, and corrupted memory cells cannot be distinguished from uncorrupted cells. In the faulty-memory RAM, it is assumed that there is an adaptive adversary that chooses how, where and when corruptions occur. The model is parametrized by an upper bound, δ, on the number of corruptions the adversary can perform during the lifetime of an algorithm, and $\alpha \leq \delta$ denotes the actual number of corruptions that take place. Motivated by the fact that registers in the processor are considered incorruptible, $O(1)$ safe memory locations are provided. Moreover, it is assumed that reading a word from memory is an atomic operation. In randomized computation, as defined in [60], the adversary does not see the random bits used by an algorithm. A sequence of elements is *faithfully ordered* if all the uncorrupted elements in the sequence appear in sorted order. An algorithm is *resilient* if it performs correctly on the set of uncorrupted elements. For instance, a resilient sorting algorithm produces a faithfully ordered sequence. We refer to a recent survey by Finoccho et al. [57], and to Chapter 6, for a overview of previous results in the faulty memory RAM.

1.3 Overview and Results

In this section we provide a brief overview of the problems considered in this dissertation. The dissertation is divided into two parts, each containing four

chapters. Part I presents work on problems inspired by Geographic Information Systems (GISs). In Part II we discuss resilient algorithms in the faulty memory RAM. The first chapter in each part give a high-level overview of the area presented in that part and give a more detailed presentation of the results in that part.

The first chapter of Part I is Chapter 2, which provides some background information on algorithms and data structures in the GIS setting. In Chapter 3 we introduce the TerraStream software package. TerraStream is a scalable solution that consists of a sequence of algorithms that form a pipeline in which each algorithm scales to massive data sets. This pipeline is flexible and allows users to choose from various models and parameters, with no or minimal manual intervention between stages. The TerraStream package has proven to be very interesting for GIS users in both industry and research. Since March 2007, representatives from more than 50 different institutions worldwide have requested access to the available limited test releases. In total there have been five major TerraStream releases and two bug fix releases. TerraStream has been used to produce scientific papers in the area of flood risk management [C5] and has also been the subject of non-scientific papers describing the system to GIS practitioners in Denmark [C2].

In Chapter 4 we examine the problem of computing contour maps of terrains I/O-efficiently. Given a sorted list $\ell_1 < \cdots < \ell_s$ of levels in $\mathbb{R}$, we present an algorithm that reports all contours of a terrain at levels $\ell_1, \ldots, \ell_s$ using $O(\mathrm{Sort}(N) + T/B)$ I/Os and $O(N/B)$ blocks of space where T is the total number of edges in the output contours and N is the number of vertices in the terrain. We also show how we can pre-process the terrain, using $O(\mathrm{Sort}(N))$ I/Os, into a linear-size data structure where all contours at a given level can be reported using $O(\log_B N + T/B)$ I/Os. Each contour is generated individually with its edges sorted in clockwise or counterclockwise order.

Finally, in Chapter 5 we develop a cache-oblivious algorithm for the *red-blue line segment intersection problem*, that is, for finding all intersections between a set of non-intersecting red segments and a set of non-intersecting blue segments in the plane. Our comparison-based algorithm uses optimal $O(\mathrm{Sort}(N) + T/B)$ I/Os where T is the number of intersections in the output and N is the total number of segments. Our algorithm is optimal and, to the best of our knowledge, the first efficient cache-oblivious algorithm for any intersection problem involving non-axis-parallel objects.

The focus in Part II is on resilient algorithms. In Chapter 6 we give an introduction to the field of resilient algorithms, including a description of some techniques used in the later chapters, and an overview of an optimal resilient sorting algorithm [59].

In Chapter 7 we design and analyze a priority queue in the faulty-memory RAM model. It uses $O(N)$ space for storing N elements and performs both Insert and Deletemin in $O(\log N + \delta)$ time amortized. The priority queue does not store elements in reliable memory between operations, only structural information such as pointers and indices. We prove that any comparison-based resilient priority queue behaving this way requires worst case $\Omega(\log N + \delta)$ time for either Insert or Deletemin.

In Chapter 8 we present our work on resilient dictionaries. We introduce a resilient randomized static dictionary that supports searches in $O(\log n + \delta)$ time, matching the bounds for randomized searching in [56]. We present the first optimal resilient static deterministic dictionary. It supports searches in a sorted array in $O(\log N + \delta)$ time in the worst case, matching the lower bounds from [60]. We also present a deterministic dynamic dictionary that supports searches in $O(\log N + \delta)$ in the worst case, and insertions and deletions in $O(\log N + \delta)$ time amortized.

Finally we present in Chapter 9, some very recent work on making resilient algorithms efficient in the external memory model. We present a model that combines the faulty memory RAM and the external memory model in the natural way. The model has three levels of memory: a disk, an internal memory of size M and $O(1)$ CPU registers. All computation takes place on elements placed in the registers. The content of any cell on disk or in internal memory can be corrupted at any time, but at most δ such corruptions can occur. In two natural variants of our model it is assumed that corruptions take place only on disk or only in memory.

1.3.1 Short overview of the included papers

In this section we give a brief overview of the papers that are presented in this dissertation, sorted by the year of publication. In citations, these papers are prefixed with a 'D' (for dissertation) and are numbered D1 through D6. Other papers coauthored by the author of this dissertation are cited using the prefix of 'C'.

[D6] *Priority Queues Resilient to Memory Faults* with A. Jørgensen, and G. Moruz. *Proc. 10th International Workshop on Algorithms and Data Structures (WADS), 2007*

This paper forms the basis of Chapter 7.

[D3] *Optimal Resilient Dynamic Dictionaries* with R. Fagerberg, G. S. Brodal, A.G. Jørgensen and G. Moruz. *DAIMI Technical Report, 2007.*

This paper is the technical report version of a paper that was presented at ESA'2007 [C3], and forms the basis of Chapter 8.

[D5] *From Elevation Data to Watershed Hierarchies* with A. Danner, K. Yi, P. K. Agarwal, L. Arge, and H. Mitasova. *Proc. 15th International Symposium on Advances in Geographic Information Systems (ACM GIS), 2007.*

In this paper we describe the first version of TerraStream, which implements a solution to the problem of extracting a river network and a watershed hierarchy from a set of points sampled from a terrain. Chapter 3 is loosely based on this paper, but contains information on the new features and structural changes made to TerraStream since the 2007 paper.

[D1] *I/O-Efficient Algorithms for Computing Contour Lines on a Terrain* with P. K. Agarwal, L. A. Arge, B. Sadri. *Proc. 24th Annual Symposium on Computational Geometry (SoCG), 2008*

Chapter 4 is based on an extended version of this SoCG'2008 paper.

[D2] *Cache-Oblivious Red-Blue Line Segment Intersection* with L. A. Arge and N. Zeh. *Proc. 16th Annual European Symposium on Algorithms (ESA), 2008*

Chapter 5 is based on an extended version of this ESA'2008 paper.

[D4] *Fault Tolerant External Memory Algorithms* with G. S. Brodal and A. G. Jørgensen. *Proc. 11th Algorithms and Data Structures Symposium (WADS), 2009.*

This is the most recent publication referenced in this dissertation, and it will be presented at WADS'2009. Chapter 9 is based on an extended version of this paper.

Part I

Cache-Efficient Algorithms with GIS Applications

Chapter 2

Background

Spatial databases and geographic information systems (GISs) are used to store and manipulate geographically referenced data, and are good examples of modern data driven applications. They have emerged as extremely powerful management and analysis tools in science, engineering, government administration and commercial applications, precisely because massive amounts of geographically based data is being generated from many different sources. Besides space shuttle and satellite based observation equipment, aircraft mounted sensor technology is also able to efficiently generate vast amounts of terrain information.

In this chapter we give a brief overview of the techniques used to gather elevation data and how these techniques are used to create elevation models. We will argue that these techniques enable us to gather large quantities of data which makes I/O-efficiency vital. Finally, we present an overview of the contributions presented in this part.

We will cover the development and application of practical I/O-efficient algorithms for terrains via the TerraStream software package (Chapter 3), as well as theoretical results on I/O-efficient algorithms for computing contour maps (Chapter 4) and a cache-oblivious algorithm for solving the so-called red-blue line-segment intersection problem (Chapter 5).

2.1 Constructing Digital Elevation Models

Getting terrain elevation data was previously a very time-consuming task. Surveyors went into the field and measured the elevation at carefully chosen points in the terrain by hand, or tried to extract elevation information from existing hand-drawn maps. The number of high-precision elevation data points one can realistically gather this way is limited. Today, many organizations gather terrain elevation data by a *remote sampling* of the survey area, which is performed without direct physical contact with the terrain. This can be accomplished by using equipment installed on an aircraft or other vehicles. These remote sampling techniques are used in practice to produce high-resolution elevation information for large areas.

Surveyor companies often make use of the increasingly popular *Light Detection and Ranging (LIDAR)* technology, which uses a laser-equipped airplane.

As the plane flies over the study area the laser emits pulses toward the ground and measures the time it takes for the pulse to return back to the airplane. Using high precision GPS equipment as well as information about the speed and orientation of the airplane, the surveyor can compute the elevation and ground coordinates of the point where the pulse hits with a horizontal and vertical precision of a centimeter or better.

Synthetic aperture radar (SAR) interferometry is another popular technique in which two or more radar images of the same terrain are acquired by two radars mounted on opposite ends of the same platform. By looking at the differences between the returns, one can extract detailed elevation information for large surfaces. This technique is implemented using radars mounted on airplanes and was also used for the NASA Shuttle Radar Topography Mission (SRTM) [115], which produced elevation points for most of the inhabited parts of the planet at 80m horizontal resolution. In the summer of 2009 this map was updated when NASA and Japan's Ministry of Economy, Trade and industry (METI) announced the immediate release of the Advanced Spaceborne Thermal Emission and Reflection Radiometer (ASTER) Global Digital Elevation Model (GDEM) [89]. The ASTER GDEM is a massive grid that covers 99 percent of the land mass from 83 degrees north to 83 degrees south latitude with elevation points measured every 30 meters.

2.1.1 Generating point clouds

The data produced by most terrain sampling methods are usually represented as big point clouds, but the data generated from the LIDAR equipment is usually more complex. Although the emitted pulse can be thought of as a travelling point in space, it will expand in the air and will have a non-negligible size by the time the it reaches the surface of the terrain. This implies that parts of the emitted pulse will hit different obstacles and be reflected back at different times. This is depicted in Figure 2.1. This modern LIDAR scanning equipment will measure the strength of the returned signal through a period of time, producing one *waveform* per emitted pulse. Software will subsequently analyse the individual waveforms and try to decide what part of the return signal corresponds to the real surface. In fact, the newest specification for the standard LIDAR data format (LAS 1.3) released in August 2009 added support for storing the waveform data, which can then subsequently be used and transmitted in a portable way [24]. However, when trying create a model of the surface of the terrain we are only interested in the part of waveform that represents the portion of the pulse hitting the surface. Thus, the first step is to turn these waveforms into points. A relatively simple strategy is to define a point for each of the peaks in the waveform, which will provide us with multiple points per emitted pulse. Alternatively, we can use the lowest peak only. For more information about LIDAR, waveforms and how to define "peaks" we refer to a recent study by Mallet and Bretar [84].

Once a point cloud has been constructed, it is usually run through a so-called *classification* algorithm. The goal of this algorithm is to assign a *class* to each point. Some of the most common classes are *ground*, *vegetation* and *building*.

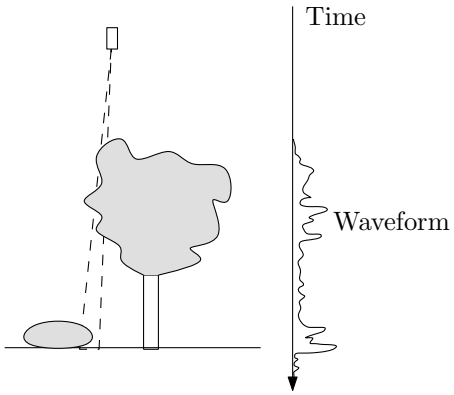


Figure 2.1: This figure shows how the pulse emitted from the laser forms a cone shape. The right side of the figure shows the intensity of the returned signal as a function of time. The width of the cone results in returns from the tree and the stone as well as the ground.

The classification algorithm uses heuristics to attempt to assign all sample points that were on the ground to the ground class, all sample points that hit man-made structures to the building class and all points that hit vegetation (e.g. trees and bushes) to the vegetation class.

2.1.2 Terrain models

In a GIS, a terrain is rarely stored as just a point cloud. Instead it is represented as a digital elevation model (DEM), either in the form of a *Triangulated Irregular Network* (TIN) or a *grid*.

A TIN DEM is a triangulation of the point cloud containing the sample points. It is common to derive a TIN DEM from a point cloud C by projecting all the points into the horizontal plane and then computing the Delaunay triangulation of the projected points. In many terrain processing applications, however, the raw elevation data is often supplemented with line segments or *breaklines* that provide additional elevation information along linear features such as roads or rivers. Breaklines can be incorporated into the TIN by constructing a Delaunay triangulation where edges of the TIN are forced to match the provided breakline segments and preserve important topological features. Such a triangulation is called a *constrained Delaunay triangulation* [39]. In some cases, like hydrological modeling, it may be beneficial to construct higher-order Delaunay triangulations as introduced by Gudmundsson et al. [68]. A k-th order Delaunay triangulation is a triangulation where the circle defined by the three vertices of each triangle contain at most k vertices. Setting $k = 0$, we get a standard Delaunay triangulation.

Recently, Moet et al. [87] defined *realistic terrains*, which are triangulated terrains with certain restrictions. By restricting the terrains considered to be *realistic* terrains, they show the complexity of some classic problems can be reduced since traditional worst-case examples fail to satisfy the conditions for realistic terrain. A terrain is realistic if it satisfies the following three conditions:

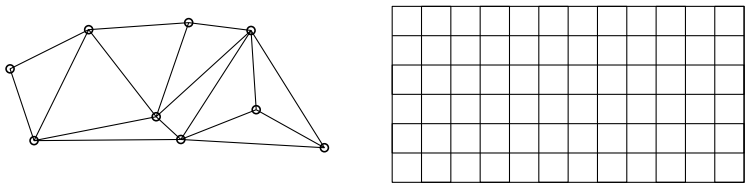


Figure 2.2: A TIN constructed from the input points and a grid covering the same area.

the projection of any triangle to the xy-plane must be *fat* (i.e. it has a minimum angle of at least some constant $\alpha > 0$), the smallest bounding rectangle of the horizontal projection of the terrain must have side lengths 1 and c for some constant $c > 0$, and the longest xy-projection over all edges of the triangulation must be at most a constant $d > 0$ times as long as the shortest one. Moet et al. proved the visibility map of a realistic terrain with N vertices has complexity $\Theta(N\sqrt{(}N))$ as opposed to $\Theta(N^2)$ for general terrains.

A grid DEM is a uniform terrain grid where each cell stores an elevation value. This grid is created from a point cloud by constructin a surface that interpolates or approximates the points in the cloud and evaluating this surface at the grid points. For instance, a grid can easily be constructed from a Delaunay triangulation of the point cloud; we find the grid points contained in each triangle and use linear interpolation across its surface to find the point elevations. The slightly more sophisticated *natural neighbor interpolation* [101] is based on the Voronoi diagram defined by the point cloud. The interpolated value is computed as a weighted sum of the area of Voronoi cells in the neighborhood of the grid point being processed. Another common way to construct grid DEMs without going directly from a triangulation is to build a quad tree on the point set. At each leaf node of the quad tree, a surface interpolating (or approximating) the points in the leaf is constructed, and this surface is subsequently used to compute the height of the cells at the grid vertices relevant for that particular leaf [2]. By using the quad tree segmentation (or other types of segmentation) to limit the number of points being considered, one can create complicated locally approximated surfaces for each leaf node and use those surfaces for interpolating the elevation of the grid points. Since computing such surfaces is expensive, it would be unfeasible to use them on the entire point cloud at once.

One of the biggest advantages of a grid model is its simplicity. Grids are easy to visualize and one can quickly implement simple algorithms working directly on them. It is most likely for this reason that grids are still the most common DEM representation in the commercial sector.

A terrain model built from a classified point cloud containing vegetation, man-made features and ground points is called a *digital surface model (DSM)*, and a model containing only ground points is a *digital terrain model (DTM)*. The problems dealing with terrains in this dissertation are primarily relevant for DTMs.

2.2 Contributions

In this section we give an overview of the problems considered in the next chapters. Terrain modeling and analysis is studied extensively in many different communities, and algorithms have been developed for many fundamental problems. Refer to [119] and the references therein for a survey. While the new technologies make it relatively easy to obtain high-quality terrain data, the full potential of the data is not being used. With modern LIDAR scanners companies around the globe have been creating nationwide point clouds with a resolution down to a meter. A recently produced point cloud for Denmark had multiple points per square meter and takes up almost one terrabyte of raw storage. With these massive datasets we run into the memory-disk bottleneck detailed in Chapter 1, which means these massive datasets are not being used to their full potential. Instead they are cut into pieces (tiles) that are handled individually (meaningful only for simpler problems), thinned by discarding billions of points, or by creating coarse grids that do not capture all the features of the original point cloud.

The main issue is that the data sets are simply orders of magnitudes larger and detailed, than the data the algorithms in current GISs were designed to handle. In the next few chapters of this dissertation we will present solutions (both theoretical and practical) to some of these problems.

DEM construction

One of the fundamental issues is the problem of constructing a DEM from a point cloud. This is a prerequisite for most applications.

Isenburg et al. [73, 74] use an interesting streaming technique to compute TINs and rasters from a point cloud while trying to keep a very low memory footprint. Their solution is not efficient in the worst case, but under reasonable assumptions about the distribution of input data their algorithms work efficiently and only require a limited amount of memory. Their main assumption is that the point cloud is presented to the algorithm using some relatively nice spatial distribution. For example, it is common for data providers to store the point cloud in rectangular tiles or in *flight lines* corresponding to straight segments (flight lines) in the flight path of the aircraft carrying the LIDAR equipment. The combination of sequential streaming and a low memory footprint makes their solutions able to compute large TINs and grids.

The TerraStream software package presented in Chapter 3 contains an implementation of I/O-efficient algorithms to construct grids and TINs based on work by Agarwal et al. [2, 4]. These algorithms are worst-case efficient.

Flow modeling and terrain conditioning

A very common operation on DEMs is modeling the flow of water on the surface of the terrain. Such modeling can reveal how water accumulates into creeks that converge and form streams, and later rivers, and can be used to extract the watersheds of the terrain. Intuitively, a *river network* is a collection of paths

that indicate where large amounts of water, or rivers, are likely to flow on the terrain. A watershed hierarchy is a hierarchical partition of the terrain into connected regions, or *watersheds*, where all water within a region flows toward a single common outlet.

A major part of TerraStream is its implementations of I/O-efficient algorithms that can perform this modeling. The I/O-efficient algorithms for certain water flow problems, including river network extraction, on grid DEMs were first implemented in the TerraFlow software package [16] and TerraStream was started in attempt to build a modern and more flexible alternative to TerraFlow and contains a new and more efficient and modular implementation of the functionality found in TerraFlow as well as an implementation of an I/O-efficient algorithm for extracting watershed hierarchies from a grid DEM river network [17]. The above algorithms typically use $O(\text{SORT}(N))$ I/Os. Recently Haverkort and Janssen [69] showed that flow modeling on grids can be done in $O(\text{Scan}(N))$ in certain cases.

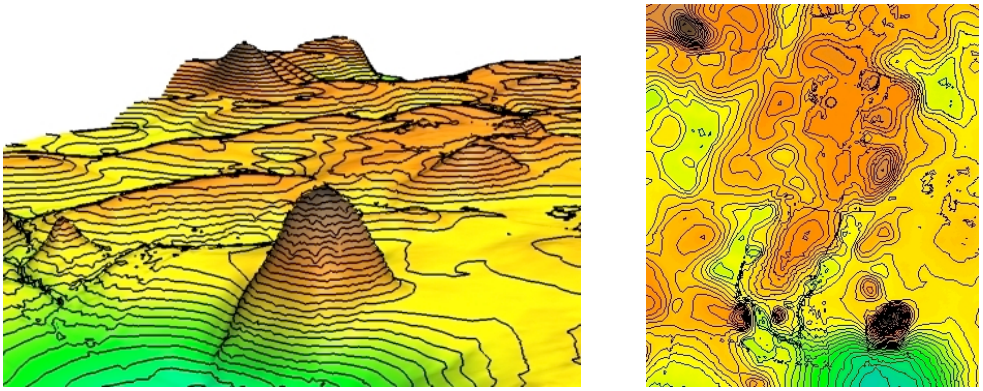
Most common modeling techniques for the flow of water on terrains assume that the water only flows downhill, however, this implies that water is impeded by local minima (sinks) in the terrain. The traditional solution to this problem, and the solution used in TerraFlow, is to eliminate all local minima (except one representing the ocean or some other global sink) by raising the elevation of vertices to the level of the spill (saddle) points. In TerraStream we have implemented a more refined *partial flooding* algorithm, based on *topological persistence* [51, 52], that detects and removes only insignificant sinks. We refer to Chapter 3 for more information.

Most flow modeling applications (including TerraFlow and TerraStream) take place on the edges of the triangulation, or from center to center on cells in a grid DEM. There has been some work on routing the water directly on the faces of a triangulated terrain, treating it as a continuous surface. However, it was proven that the complexity of the flow paths in such a terrain with N vertices is $\Theta(N^3)$ [47]. Very recently, de Berg et al. [48] studied the complexity of the river network in *α-fat terrains*, which are terrains where the minimum angle in a triangle is α (satisfying the minimum angle requirement for a realistic terrain as defined in Section 2.1.1). They prove the complexity of the river network in an α-fat terrains is $O(N^2/\alpha^2)$, and they also describe I/O-efficient algorithms for computing these river networks.

Contour mapping

A *contour* (or *isoline*) of a terrain $\mathbf{M}$ is a connected component of a level set of $\mathbf{M}$. *Contour maps* (aka *topographic maps*), consisting of contour lines at regular height intervals, are widely used to visualize a terrain and compute certain topographic information of a map. This representation goes back to at least the eighteenth century [103].

On a TIN or grid DEM each individual contour consists of a chain of non-intersecting (but possibly overlapping on edges and vertices) line segments. These segments can easily be computed by considering each face of the terrain model in turn, however, this will generate a big soup of unsorted line segments



(a) (b)

Figure 2.3: Examples of equidistant contours of a terrain. (a) Rendered on a perspective view of the terrain in 3d. (b) Projected onto the 2d plane.

that are hard to use for further processing or for vizualization. Instead, we want to compute the contours such that the segments forming each contour can be labeled and reported, preferably in clockwise order of the segments. We refer the reader to the tutorial [102] and references therein for a review of practical algorithms for contour and iso-surface extraction problems.

A simple way of computing such a contour K of a terrain $\mathbf{M}$ is to start at one triangle of $\mathbf{M}$ intersecting the contour and trace out K by walking through $\mathbf{M}$ until we reach the starting point. If we have a starting point for each contour of a level set of $\mathbf{M}$, for a given level ℓ, we can compute all contours of that level set in time linear in the size of the output in the internal-memory model. Such a starting point can be computed using the so-called *contour tree* [38] which encodes a "seed" for each contour of K. Many efficient internal-memory algorithms are known for computing a contour tree (see e.g. [38]). Hence, one can efficiently construct a contour map of K. This approach of tracing a contour extends to higher dimensions as well, e.g. the well-known marching-cube algorithm for computing iso-surfaces [82]. Although, as explained in Chapter 4, recent work by Agarwal et al. [5] makes it possible to compute the contour tree I/O-efficiently, which is hard to do by tracing. In fact, there is no known I/O-efficient algorithm for tracing general paths on the surface of a triangulation.

This dissertation contains the description of two very different approaches to the problem of computing the contour map. In TerraStream we developed and implemented a simple practical $O(\text{Sort}(N+T))$ algorithm where T is the total number of edges in the output contours. The algorithm works under reasonable assumptions about the input model and the size of memory. By combining the contour generation code with the partial flooding available in TerraStream we are able to remove "insignificant" contours in a well-defined and controlled manner. Our approach is briefly explained in Chapter 3 and in a paper that is currently being prepared [C1].

In Chapter 4 we describe a more efficient algorithm that can compute the

contour map without the extra assumptions on the memory size. The contour map can be generated in $O(\text{Sort}(N)+T/B)$ I/Os and linear space. Each contour is generated individually with its edges sorted in clockwise or counterclockwise order. We also present a linear sized data structure that can be computed in $O(\text{Sort}(N) + T/B)$ and can answer level-set queries in $O(\log_B N + |T'|)$, where T' is the number of segments in the requested level set.

Spatial join

The final GIS related problem considered in this dissertation is the *red-blue line segment intersection problem*, which, unlike the previous mentioned problems, does not involve elevation models. It belongs to the class of *spatial join* problems, in which two spatial databases are merged. In the red-blue line segment problem we are given a set each of internally non-intersecting red and blue segments in the plane, and the objective is to report all intersections between a red and a blue segment. A solution to this problem can, for instance, be used to find where roads intersect rivers by finding all intersections between roads in a set of road segments and a set of river segments in the plane.

Arge et al. [23] developed an optimal I/O-efficient algorithm that solves the red-blue line segment intersection problem using $O(\text{Sort}(N) + T/B)$ I/Os, where T is the number of reported intersections. In Chapter 5 we present a cache-oblivious algorithm that solves the problem in the same bound. To the best of our knowledge, this is the first efficient cache-oblivious algorithm for any intersection problem involving non-axis-parallel object.

Chapter 3

TerraStream: I/O-Efficiency in Practice

Many GIS applications use a *pipeline* (or *work-flow*) approach for combining many small, simple algorithms into a larger, more complex application. Often, the individual stages in a pipeline are developed independently and require manual intervention to pre-process or post-process the data between different stages. Furthermore, while a typical GIS can manage gigabytes of data consisting of hundreds or thousands of smaller individual data sets, most systems are not designed to handle multi-gigabyte data sets. Moreover, previous GIS algorithms designed to scale to massive data have focused on individual stages of the pipeline, and have been designed for either grid or TIN DEMs.

In this chapter we describe TerraStream; a scalable solution that consists of a pipeline of components where each component uses I/O-efficient algorithms. Thus, each algorithm in the pipeline scales to massive data sets. The highly modular and configurable pipeline is designed to reduce manual intervention, and to allow for easy addition of new modeling features. Our approach provides several parameters to control the behavior of each pipeline stage and users can choose between several popular models in each stage. Additional models and features can also be added with minimal effort.

TerraStream consists of eight main stages: DEM construction, hydrological conditioning (sink removal), flow modeling, extraction of river networks, extraction of watershed hierarchies, quality metrics, flood map computation and contour map computation. Figure 3.2 and 3.1 illustrate the overall structure of the pipeline and the outputs of its several stages. TerraStream builds upon and extends a number of previously developed I/O-efficient terrain algorithms, with several new algorithms designed to form the whole pipeline. In addition, a considerable amount of engineering effort is devoted to making TerraStream efficient and practical. Our main technical contributions in this chapter include the following:

- We take a unified approach for handling both TIN and grid DEMs. We represent TIN and grid DEMs as a graph, which we refer to as a *height graph*. We then design or modify algorithms in the subsequent pipeline stages to use height graphs. Our methods therefore work on both grid and TIN DEMs, and most of the stages in our pipeline are compatible with both grids and TINs on the source code level. Such a unified approach

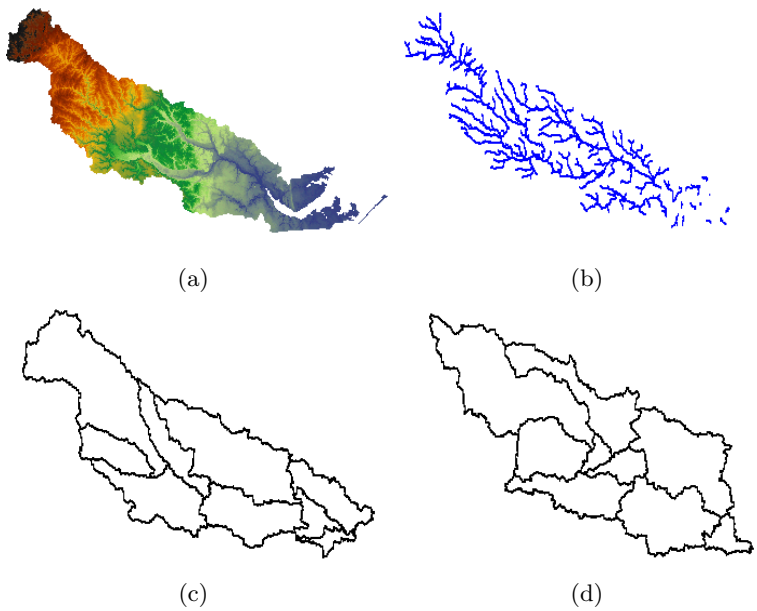


Figure 3.1: An overview of the flow modelling components of TerraStream. (a) DEM of Neuse river basin derived from lidar points (b) Rivers with drainage greater than 5000 acres (2023 hectares) extracted from DEM. (c) First level of Pfafstetter watershed labels for largest basin in Neuse. (d) Recursive decomposition of basin four.

makes software maintenance much easier and reduces the amount of effort when additional features are to be supported. Note the unified approach does not come at a cost of decreased performance. Our pipeline works on a given DEM type as efficiently as if the code were written solely for that particular type.

- We design and implement an $O(\text{SORT}(N))$-I/O algorithm for assigning a numerical score or *significance* to each sink, or local minimum, in a height graph. We then use a sink's significance for *hydrologically conditioning*, in which we remove small or insignificant sinks from a terrain while preserving significant sinks such as large closed basins with no outlet. This step of removing unimportant sinks is crucial to all known flow models.

- In addition to extending earlier grid based flow modeling algorithms [16] to height graphs, we develop a simple and practical algorithm for detecting flat areas in a terrain. We also implement improved flow routing on flat areas which commonly cause problems in flow modeling algorithms. Flat areas may exist in either the original sample data or be introduced into the terrain as a side-effect of hydrological conditioning.

- We develop and implement I/O-efficient algorithms for computing exactly what part of a height graph will be flooded if the water level in the oceans rise by a specified certain amount. Our algorithm takes dikes and natural

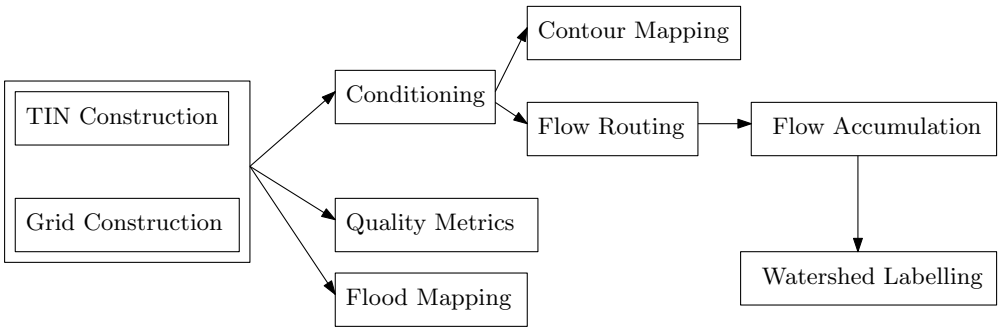


Figure 3.2: Overview of pipeline stages showing inputs, outputs, and optional modeling parameters for each stage.

flood barriers into account.

- We briefly present an algorithm for assesing the quality of a raster DEM generated from some point cloud C. Our simple quality measure is the distance from each grid cell to the nearest point in the cloud.
- We also briefly present practical algorithms for generating the contour maps of a terrain. These algorithms are more practical than algorithms presented in Chapter 4.

In the rest of the chapter we cover these pipeline stages one by one.

3.1 DEM Construction

The first stage of our pipeline constructs a grid or TIN DEM from a set $\mathcal{S}$ of N input points in $\mathbb{R}^3$. Below we briefly review the algorithms we utilize; the reader is referred to [2, 4] for a complete overview of, and comparison with, previous work. We also introduce the unified height graph that we use in later stages.

Grid DEM construction

As described in Chapter 2, the common approach for constructing a grid DEM of a user-specified cell size from $\mathcal{S}$ is to use one of many interpolation or approximation methods to compute a height value for each grid point (refer to e.g., [85] and the references therein). For inputs with more than a few thousand points, applying an interpolation method directly is infeasible because of the computational complexity of solving large systems of linear equations. We chose for TerraStream a recently developed I/O-efficient algorithm [2] that uses a quad-tree segmentation in combination with a regularized spline with tension interpolation method [86] to construct a grid DEM in $O(\frac{N}{B}\frac{h}{\log \frac{M}{B}} + \textsc{sort}(T))$ I/Os, where h is the height of a quad tree on $\mathcal{S}$ and T is the number of cells in the desired grid DEM. Our implementation is modular and allows users to implement a variety of interpolation methods. One advantage of the spline method we use is that it allows for smooth approximation of data and can also

be used to accurately compute properties such as slope, profile curvature, and tangential curvature (which are important for landform analysis and landscape process modeling). Note that the algorithm uses $O(\text{SORT}(N) + \text{SORT}(T))$ I/Os if $h = O(\log N)$, that is, if the points in $\mathcal{S}$ are distributed such that the quad tree is roughly balanced. We store the output grid in a simple row-major format to allow efficient row access to grid cells in later stages.

TIN DEM construction

In TerraStream, we use a randomized I/O-efficient algorithm [4] for constructing a *constrained Delaunay triangulation* of a set $\mathcal{S}$ of N points and a set $\mathcal{L}$ of K line segments. The algorithm uses $\text{SORT}(N)$ expected I/Os if the number of constraining segments K is smaller than then memory size M. In most applications, K considerably smaller than both N and M. We store the output TIN in an "indexed triangle" format, which is a common, simple, and compact representations TINs. In this format, the coordinates of the TIN vertices are stored consecutively on disk along with a unique vertex ID, followed by a list of triangles each identified by three vertex IDs in clockwise order.

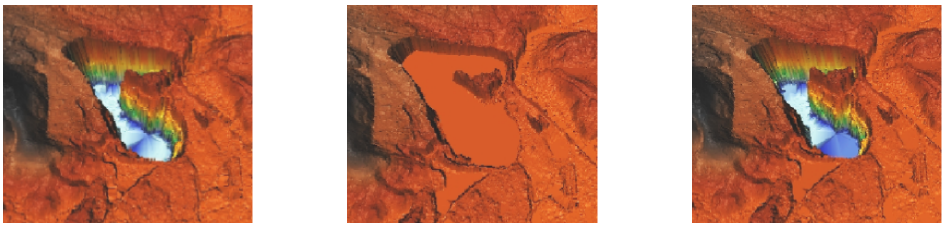
Height graph

To avoid designing separate grid and TIN algorithms for each of our successive pipeline stages, we define a graph, which is typically referred to as the *height graph*, that unifies both DEM formats. A height graph $G = (V, E)$ is an undirected graph derived from a DEM, with a *height* $h(v)$ and an *id* $id(v)$ associated with each $v \in V$. The id's are assumed to be unique. 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 $id(u) > id(v)$. The concept of *lower* than is defined similarly. The vertices and edges of a TIN DEM naturally form a height graph. To obtain a height graph from a grid DEM, we include all the DEM vertices along with all the boundary edges of the grid cells and the diagonals for each grid cell. Note that in certain applications when planarity is required we could add only one of the diagonal edges, or not at all. Our pipeline works with any of these choices.

In both the TIN and the grid case, we add an additional "outside" vertex ξ with $h(\xi) = -\infty$, which is connected to all the vertices on the boundary of the DEM. A height graph can be constructed from a grid or TIN DEM of size N in $O(\text{SORT}(N))$ I/Os.

3.2 Hydrological Conditioning

Most flow modeling algorithms assume water will flow downhill until it reaches a local minimum or *sink*. In practice however, local minima in DEMs fall into two primary categories; *significant* and *insignificant*, or spurious, sinks. When modeling flow one naturally assumes that water flows downhill until it reaches a sink. Significant sinks correspond to large real geographic features such as quarries, sinkholes or large natural *closed* basins with no drainage outlet. The insignificant sinks may be due to noise in the input data or correspond to small



(a) (b) (c)

Figure 3.3: (a) Original terrain. (b) Terrain flooded with $\tau = \infty$. (c) Terrain partially flooded with persistence threshold $\tau = 30$.

natural features that flood easily. When modeling water flow these insignificant sinks impede flow and result in artificially disconnected hydrological networks. The second stage of our pipeline "hydrologically conditions" a DEM for the flow modeling stage by removing insignificant sinks, while preserving significant sinks.

The known hydrological conditioning algorithms remove *all* sinks using a so-called *flooding* approach [75], which simulates uniformly pouring water on the terrain until a steady-state is reached. A weakness of this approach is that it removes even significant sinks. See Figures 3.3(a) and (b). Furthermore, the previous I/O-efficient algorithm [16] works only for grids and assumes that all sinks fit in memory. This assumption does not hold for large high-resolution terrains. Instead we use a *partial flooding* algorithm, based on *topological persistence* [51,52], that detects and removes only insignificant sinks, as indicated in Figure 3.3(c). We briefly describe topological persistence and then present our algorithm.

3.2.1 Topological persistence

In the context of a terrain T represented by a height graph, topological persistence [51, 52], matches each local minimum (sink) vertex v of T to a higher "saddle" vertex w (see [50] for the precise definition of a saddle) and assigns a *persistence* value, denoted by $\pi(v)$, to v. In [51], $\pi(v)$ is defined to be the difference in the heights of v and w, i.e., $\pi(v) = h(w) - h(v)$. The persistence $\pi(v)$ denotes the *significance* of the sink v. Intuitively, the saddle vertex w is a vertex at which two distinct connected components of the portion of T lying strictly below w merge. Each connected component is represented by the lowest vertex in the component. Suppose v is the highest representative of the two connected components merged by w and let u denote the representative of the other component. Then topological persistence induces a *merge tree* on the sinks of T, in which u is the parent of v. The merge tree has the property that the heights of vertices on any root-to-leaf path increase, while the persistence values decrease along such a path.

Agarwal et al. [5] developed an $O(\text{SORT}(N))$-I/O algorithm for computing the persistence of all sinks in a triangular planar height graph, and computing the merge tree. They also developed and implemented a simpler and practical

$O(\text{SORT}(N)\log(N/M))$-I/O algorithm. We extend this algorithm to form our partial flooding algorithm given below.

3.2.2 Partial flooding

We use topological persistence as a measure of the significance of a sink. Given a user-specified threshold τ, we declare all sinks with persistence less than τ to be the insignificant sinks and remove all such sinks using a partial flooding method described below. The user can change the threshold to control the smallest feature size to be preserved.

We define partial flooding of a height graph by generalizing the flooding definition for grid DEMs [16, 75]. Let G be a height graph with one or more significant sinks $\zeta_1, \ldots, \zeta_k$. Let the *height of a path* in G 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 any $1 \leq i \leq k$. In *partial flooding*, we change the height of each vertex in G to its raise elevation. Partial flooding produces a modified height graph containing only significant sinks whose persistence value is greater than τ. Note that if $\tau = \infty$, our definition of partial flooding is the same as the original definition of flooding. Thus, partial flooding is a tunable way to condition the terrain for the purpose of flow modeling.

To efficiently condition a terrain using partial flooding, we utilize the following property of the merge tree (proof can be found in [46]). Let u be a node in the merge tree that does not correspond to a significant sink, but whose parent does. Let v be any node in the sub-tree rooted at u. Then the raise elevation $r(v)$ of v is $r(v) = r(u) = h(u) + \pi(u)$, where $h(u)$ is the height of u and $\pi(u)$ is its persistence value. Thus, we obtain a simple way to compute the raise elevations for each sink in the merge tree (or more precisely, the sinks of G corresponding to vertices in the merge tree): For each insignificant sink u in the merge tree whose parent corresponds to a significant sink, we propagate $r(u)$ to all vertices rooted below u. To compute the raise elevations efficiently we direct tree edges from a node to its children. By traversing the vertices in height order and forwarding $r(u)$ along all outgoing paths starting from u, we can assign each vertex in a subtree of u the proper raise elevation. This traversal can be performed in $O(\text{SORT}(N))$ I/Os using standard techniques [11, 40].

What remains is to compute the raise elevations for all non-sink vertices in the height graph G. To do so we first assign a sink label to each vertex in G. A vertex u is assigned sink label v if there is a path of monotonically decreasing height from u to a sink v; if several such paths exists, we chose the one to the lowest sink v. With each sink label v we also store the raise elevation $r(v)$ of v. To assign labels to each vertex, we construct a DAG by directing edges in G from lower height vertices to higher height vertices. The vertices in this DAG are naturally sorted in topological order by increasing height. We traverse the DAG in topological order and forward sink labels along outgoing edges; the sink label for a vertex u is simply the label corresponding to the lowest sink among the labels received from preceding vertices. This traversal is similar to the merge tree-traversal and can be performed in $O(\text{SORT}(N))$ I/Os [11, 40]. We can show that (proof in [46]) the raise elevation of a vertex u in the height graph

with elevation $h(u)$ and sink label v is $r(u) = \max\{h(u), r(v)\}$. Thus we have computed the raise elevations for all vertices in G.

In summary, for a given threshold τ, we can partially flood the terrain represented as a height graph in $O(\text{SORT}(N))$ I/Os.

3.3 Flow Modeling

The third stage of our pipeline models the flow of water on a hydrologically conditioned DEM, represented as a height graph. It consists of two phases. The first *flow-routing* phase, we compute a *flow direction* for each node v in the height graph that intuitively indicates the direction water will flow from v. In the second *flow-accumulation* phase, we intuitively compute the area of the terrain represented by nodes upslope of each node v.

Flow routing

Given a height graph $G = (V, E)$, the flow-routing phase computes a directed subgraph $\mathcal{F}(G) = (V, E_r)$ of G called the *flow graph*. An edge (v, u) in $\mathcal{F}(G)$ indicates that water can flow from v to u. E_r is constructed from G by looking at each vertex v and its neighbors and applying a *flow-direction* model. We implemented two popular flow-direction models:

- *Single-flow-direction* (SFD) model: for each vertex v, the edge from v to the neighbor with lowest height lower than the height of v is selected.
- *Multi-flow-directions* (MFD) model: for each vertex v, all edges going to neighbors of less height than v are selected.

Several other flow-direction models have also been proposed (e.g., [79, 109]), and most of them can be incorporated in our pipeline, we refer to [16] for more information on SFD and MFD routing. If the height of every vertex in G is distinct, we can easily construct $\mathcal{F}(G)$ in $O(\text{SORT}(N))$ I/Os using standard techniques, by simply examining the neighbors of every vertex in the height graph and assign a flow directions to all but the sinks. In the SFD and MFD models, the resulting flow graph is a forest or DAG, respectively. However, realistic terrains G can have large *flat areas* of vertices with no neighbors with lower height. Flat areas can be natural plateaus in the terrain, or they can appear as by-products of the hydrological conditioning stage. Detecting these flat areas and routing flow through them in a realistic way is challenging, and we discuss these steps further in Section 3.3.1. We have implemented extensions of SFD and MFD models that incorporate routing on flat areas.

Flow accumulation

Given a flow graph $\mathcal{F}(G)$ with flow directions, the flow accumulation [93] phase intuitively computes the area of the terrain represented by vertices upslope of each node v. More precisely, each vertex v in the flow graph $\mathcal{F}(G)$ is assigned some initial flow. Each vertex then receives incoming flow from upslope neighbors

and distribute all incoming and initial flow to one or more downslope neighbors. The flow accumulation of a vertex v is the sum of its initial flow and incoming flow from upslope neighbors.

Our flow accumulation algorithms visits the nodes of $\mathcal{F}(G)$ in topological order and for each vertex v, computes the total incoming flow and distributes flow to each downslope neighbor u with an edge (v, u) in $\mathcal{F}(G)$ using a given function. Our $O(\text{SORT}(N))$ I/O implementation adapts an algorithm by Arge et al. [21] developed for grid DEMs and implemented in TerraFlow. Given the flow accumulations for all vertices, we can extract *river networks* [93] simply by extracting edges incident to vertices whose flow accumulation exceeds a given threshold. We can easily do so in $O(\text{SORT}(N))$ I/Os.

In our implementation we distribute flow distributed to a lower neighbor u of v, in proportion to the height difference between v and u. However, our flexible pipeline allows for other distribution functions. In terms of initial flow, one typically assigns a "unit" of initial flow to each vertex if G represents a grid DEM, since all grid cells have the same area. If G represents a TIN DEM, one typically distributes the xy-projection of the area of each triangle in G equally among its three vertices. We have implemented these choices, but TerraStream allows users to adapt to other applications by specifying an initial flow for each vertex.

3.3.1 Handling flat areas

A robust flow model must handle extended flat areas in a terrain. A vertex v in a height graph G is *flat* if $h(v) \leq h(u)$ for all neighbors u of v in G, or if v has a neighbor of the same height that has no lower neighbors. A *flat area* is a maximal connected component of flat vertices of the same height. A *spill point* of a flat area is a vertex with a downslope neighbor. Routing flow across flat areas is composed of two steps; detecting all flat areas and routing flow across each flat area.

Detecting flat areas

Detecting flat areas is equivalent to finding connected components of same-height vertices in G. A previous theoretical $O(N/B)$-I/O algorithm for computing connected components on grid DEMs [21] exists, but is too complex to be of practical interest and can not be extended to work on height graphs. In fact, TerraFlow [16] implements a different $O(\text{SORT}(N))$ algorithm for grids. We developed and implemented a simpler algorithm for height graphs that scans the vertices and their neighbors and uses a batched union-find structure to merge vertices in the same flat area into a single connected component. Theoretically our algorithm uses $O(\text{SORT}(N))$ I/Os [5]. However, in the actual implementation we have used a simple practical union-find implementation [5] such that the algorithm uses $O(\text{SORT}(N) \log(N/M))$ I/Os.

Improved grid DEM flat detection algorithm

Since TerraStream is modular and allows us to plug in customized modules easily, we have implemented a novel and simple $O(N/B)$-I/O algorithm for detecting flat areas on grid DEMs in the case where a constant number of rows of the grid fit in memory. In this case, we can, in practice, handle grid DEMs containing 2^{44} cells occupying more than 128 TB of space using only 256 MB of main memory. Note that a grid row fits in memory when $\sqrt{N} \leq M$.

Intuitively, our algorithm performs two row-by-row sweeps of the grid DEM and assigns every cell in the same connected flat area the same unique *connected component label*, while only keeping two grid rows and a small union-find structure in main memory at all times. The union-find structure maintains connected component labels for the two grid rows currently in memory. The first sweep is a *down-sweep* from the topmost to bottommost row in the grid that assigns provisional connected component labels to each flat cell. After the down-sweep all flat cells with the same label are in the same connected component. However, a single flat area may have multiple labels. We therefore perform a second *up-sweep* from the bottommost to topmost row in the grid and assign a single unique connected component label to all cells in the same flat area. The sweeps are described in some detail below, but we refer to [46] for the full analysis.

In the down-sweep, we keep the current row and the row immediately above it in memory. In the top row, each flat cell has already been assigned a connected component label. To process the current row we first visit the cells in the row from left to right and assign a new unique label $l(u)$ to each flat cell u. Then we visit the cells in the current row again and perform a UNION on $l(u)$ and $l(v)$ for any pair of neighboring flat cells (u, v) currently in memory. We implement the union-find structure such that the unique representative for a set of labels is the label that was assigned earliest (at the highest row). Finally, we update the label of each flat cell u in the current row to be the label FIND(u). We can prove [46] that after processing the current row, two cells in the current row and in the same flat area have the same label if and only if they are connected by a path completely contained in the current row and the rows above it.

In the subsequent up-sweep, we also keep two rows in memory; the current row and the row immediately below it. To process the current row we first visit the cells in the row from left to right and determine for each flat cell u if it has a flat neighbor v in the row below the current row; if so we perform a UNION on $l(u)$ and $l(v)$. As in the down sweep, we then update the label of each flat cell u in the current row to be the label FIND(u). After the up-sweep, cells in the same connected flat area have the same connected component label.

Since the union-find structure used during the two sweeps never contains more than $2\sqrt{N}$ different labels, we can implement it such that it uses $O(\sqrt{N})$ space. Thus it fits in main memory at all times and does not require any I/Os. Therefore our algorithm uses $O(N/B)$ I/Os, because we scan the grid DEM twice.

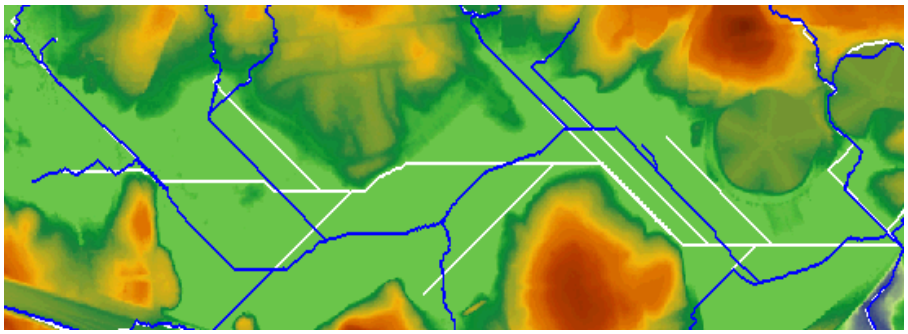


Figure 3.4: Comparison of routing methods on a flat area with a single spill point on the right. Rivers indicated in white were extracted by using the smallest Euclidian distance. Blue (black) river lines were computed using the Soille et al. approach

Improved routing on flat areas

When routing flow on flat areas we distinguish between flat areas that have at least one spill point and those that do not; in the first case water should be able to flow out of the flat area through the spill points, while in the second case water is simply absorbed into the extended sink.

In many early flat area routing approaches (see e.g. [16] and references therein) flow directions were assigned in a simple way such that each cell v was assigned a flow direction to the adjacent neighbor that was along the shortest Euclidean path along grid edges from v to the closest spill point. However, these approaches are not hydrologically realistic and tend to create many parallel flow lines [111]. Recently, a new more realistic flat area routing approach was proposed by Soille et al. [105]. Their approach, based on geodesic time and distance, improves an earlier approach by Garbrecht and Martz [65]. Given a flat area, define H to be the set of flat vertices having an upslope neighbor. The algorithm of Soille et al. [105] computes the minimum distance d_v from each of the other flat vertices v to a cell in H. Let $d_{\max}$ be the maximum distance d_v computed in the flat area. Each vertex v is assigned a flow direction to the first vertex on the minimum-cost path from v to a spill vertex, where the cost of a path is defined as the sum of $d_{\max} - d_u$ for all cells u along the path. If no spill vertex exists, the minimum cost paths from a vertex with distance $d_{\max}$ is used. Since $d_{\max} - d_u$ is large near the upslope boundaries, the shortest paths will converge toward the low cost vertices away from the boundaries. This substantially increases the convergence of the flow routing paths.

We implement both the Soille et al. [105] approach, and a simple shortest path approach in TerraStream, Figure 3.4 compares the two. Both approaches are implemented under the assumption that each flat area fits in main memory; in our experience with high resolution floating point elevation data this is a reasonable assumption.

3.4 Watershed Hierarchy Extraction

The final stage of our pipeline computes a watershed given a flow graph $\mathcal{F}(G)$ in which each vertex in $\mathcal{F}(G)$ is augmented with its flow accumulation. As mentioned earlier, a watershed hierarchy is a hierarchical decomposition of the terrain into a set of disjoint regions, or watersheds, where all water flows towards a single outlet. Such a decomposition is the basis of several GIS algorithms for hydrological and pollutant transport modeling.

Depending on the application, users may want to study a small hydrological unit in their neighborhood or study major river basins at a state-wide or nation-wide scale. Therefore it is advantageous to have a hierarchical decomposition of the terrain into nested units of arbitrary small size. Furthermore, it is useful if the units are assigned a unique label that also encodes topological properties such as upstream and downstream neighbors; thus making it possible to automatically identify hydrological units of interest based on the label alone.

Verdin and Verdin [117] described a Pfafstetter labeling method, which hierarchically divides a terrain into arbitrarily small regions, each with a unique label, such that the Pfafstetter labels encode topological properties such as upstream and downstream ordering. At the topmost level the terrain is divided into nine disjoint watersheds; each of these watersheds are recursively divided into nine smaller watersheds. Arge et al. [17] previously developed an algorithm using $O(\text{SORT}(N) + T/B)$ I/Os for computing the Pfafstetter labels of a grid DEM of N cells, where T is the total size of the labels. The algorithm uses a data structure equivalent to a flow graph $\mathcal{F}(G)$ computed using a single flow direction model and augmented with flow accumulations for each vertex. For our final pipeline stage, we modified the algorithm to use the flow graph $\mathcal{F}(G)$ created in Section 3.3, and thus extend the previous algorithm to work for flow graphs derived from both grid and TIN DEMs.

3.5 Recent Developments and Ongoing Work

In this section we briefly describe some recent and ongoing developments to TerraStream. The author of this dissertation has not been directly involved with all of these developments, but they are included here for completeness.

3.5.1 Volume and area computations

Based on work by Revsbæk [100], TerraStream now has the ability to compute the *volume* and *area* of the sinks in the terrain. This gives rise to a new way of defining the importance of sinks for the purpose of the partial flooding described in Section 3.2. Besides using the persistence of a sink, the user can now use information about the volume and the area (projected and surface area) of the sink to decide whether or not a sink should be flooded. We refer to [20, 100] for more details.

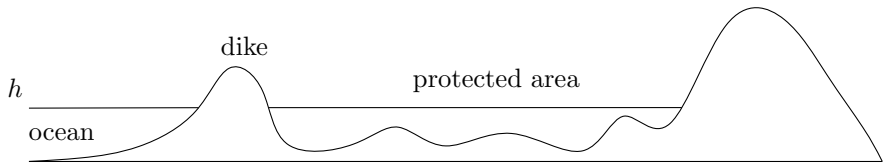


Figure 3.5: Flood map computation, the protected area behind the dike is not flooded when the water level becomes h.

3.5.2 Flood mapping

Recently, we added support for computing flood maps to TerraStream. Computing highly accurate flood risk information is a very complex task and involves many types of fine-grained information about the study area. However, with modern high resolution DEMs that contain small but important features (e.g. dikes), it is possible to get good initial flood risk estimates using only the elevation information, and ignoring the effect of sewers and the groundwater. The simplest way of computing what is flooded when the sea level rises is to intersect the terrain with a horizontal plane at the flooding height and mark everything underneath this plane as being flooded. This can easily be done in $O(\text{Scan}(N))$ I/Os by simply scanning the height graph and marking all vertices that have a lower elevation than the flooding height. This simple computation is problematic because it ignores the effect of dikes and wrongly marks many depressions as being flooded, an example of this can be seen in Figure 3.5.

TerraStream contains an implementation of a better flooding algorithm where a cell is only marked as flooded if there is a path from that cell to the flooding source (the ocean in most cases) that never passes through a point that is higher than the new water level. In other words, if the water level is h a cell u is marked as being flooded if there is a path $u = p_1, \ldots, p_k = s$ in the height graph from u to some flooding source vertex s such that $h(p_i) < h$ for all $i = 1, \ldots, k$. The general flooding output can be easily computed using the flat detection algorithm from Section 3.3. We transform the terrain into one where all vertices with an elevation less than h, have been raised to height h. We can now perform flat detection, and it can be seen that a vertex is flooded if and only if it is of height h in the new terrain and is in the same flat component as a flooding source. Among other things, we have used this algorithm on a high resolution DEM of the city of Aarhus, Denmark [C5]

3.5.3 Quality metric

We also developed and implemented an I/O-efficient method that can be used to assess the quality of a grid DEM derived from a point cloud. More precisely, our algorithm computes the distance from each grid cell to the nearest point in the original point cloud. This can be used to check if any particular region contains too sparsely gathered data, resulting in big interpolated areas in generated DEMs. By using this in the field, surveyours can ensure that they have a good coverage of the target area, and schedule new flights immediately if there

are areas where the point density is too low. Doing the new flights quickly is significantly cheaper than having to make an entire team return to a survey site to patch holes in the data. A publication with the algorithm and practical results is in preparation [14].

3.5.4 Contour maps

As explained in Chapter 2, TerraStream contains an implementation of a practical algorithm for generating contour maps I/O-efficiently. The algorithm consists of a couple of sweeps using a generalization of the flat algorithm presented in Section 3.3 that works on both grids and TINs. The algorithm can be generalized to TINs by assuming that for any horizontal line, all the triangles intersecting this line fit in memory. This can be exploited to label contour segments such that two independent contours have different labels, and we can then sort the contours in clockwise order by assuming that the segments of any one contour fit in memory (without this assumption the algorithm still works since the individual contours can be sorted using list ranking).

Contours obtained from a very detailed grid or TIN DEM are often not very visually pleasing. They are simply too detailed, that is, contain too many small contours and many contours consisting of many very small segments. Many, but not all, of the small contours are a result of small features (depressions or bumps) in the terrain. Thus by performing a topological conditioning on the DEM before computing contours, one can remove the insignificant contours. For example, if one is generating a contour for each meter of elevation, it is natural to remove all features of depth less than one meter. The method is promising because it removes insignificant contours in a well-defined way. A publication on our contour generation methods is in preparation [C1, 20].

Chapter 4

I/O-Efficient Construction of Contour Maps

In this chapter we propose efficient algorithms for computing contour maps as well as computing contours at a given level.

Recall from the discussion in Chapter 2 that a natural way of computing a contour K of a terrain $\mathbf{M}$ is simply to start at one triangle of $\mathbf{M}$ intersecting the contour and then tracing out K by walking through $\mathbf{M}$ until we reach the starting point. The starting point for each contour of a level set of $\mathbf{M}$, for a given level ℓ can be computed using the contour tree of the terrain *contour tree* [38] An $O(\text{Sort}(N))$ algorithm in the I/O-model was recently proposed by Agarwal et al. [5] for constructing a contour tree of $\mathbf{M}$, so one can quickly compute a starting point for each contour. However, it is not clear how to trace a contour efficiently in the I/O-model, since a naive implementation requires $O(T)$ instead of $O(T/B)$ I/Os, to trace a contour of size T. Even using a provably optimal scheme for blocking a planar (bounded degree) graph, so that any path can be traversed I/O-efficiently [3, 91], one can only hope for an $O(T/\log_2 B)$ I/O solution. Nevertheless, I/O-efficient algorithms have been developed for computing contours on a terrain. Chiang and Silva [42] designed a linear-size data structure for storing a TIN terrain $\mathbf{M}$ on disk such that all T edges in the contours at a query level ℓ can be reported in $O(\log_B N + T/B)$ I/Os, but their algorithm does not sort the edges along each contour. Agarwal et al. [3] designed a data structure with the same bounds so that each contour at level ℓ can be reported individually, with its edges sorted in either clockwise or counterclockwise order. However, while the space and query bounds of these structures are optimal, preprocessing them takes $O(N \log_B N)$ I/Os. This bound is more than a factor of B away from the desired $O(\text{Sort}(N))$ bound. Thus using this structure one can at best hope for an $O(N \log_B N + T/B)$ I/O algorithm to compute a contour map; here T is the total size of all the output contours.

4.1 Our Contribution

Let $\mathbf{M}$ be a terrain represented as a triangulated surface (TIN) with N vertices. For a contour K of $\mathbf{M}$, let $F(K)$ denote the set of triangles intersecting K. We prove (in Section 4.3) that there exists a total ordering '$\lhd$' on the triangles of $\mathbf{M}$ that has the following two crucial properties:

(C1) For any contour K, if we visit the triangles of $F(K)$ in $\triangleleft$ order, we visit them along K in either clockwise or counter clockwise order.

(C2) For any two contours K_1 and K_2 on the same level set of $\mathbf{M}$, $F(K_1)$ and $F(K_2)$ are not interleaved in $\triangleleft$ ordering, i.e., suppose the first triangle of $F(K_1)$ in $\triangleleft$ appears before that of $F(K_2)$, then either all of the triangles in $F(K_1)$ appear before $F(K_2)$ in $\triangleleft$, or all triangles of $F(K_2)$ appear between two consecutive triangles of $F(K_1)$ in $\triangleleft$.

We call such an ordering a *level-ordering* of the triangles of $\mathbf{M}$. We show that $\triangleleft$ can be computed using $O(\text{Sort}(N))$ I/Os. Next, we present two algorithms that rely on this ordering.

Computing a contour map Given as input a sorted list $\ell_1 < \cdots < \ell_s$ of levels in $\mathbb{R}$, we present an algorithm (Section 4.4) that reports all contours of a terrain $\mathbf{M}$ at levels $\ell_1, \ldots, \ell_s$ using $O(\text{Sort}(N) + T/B)$ I/Os and $O(N/B)$ blocks of space, where T is the total number of edges in the output contours. Each contour is generated individually with its edges sorted in clockwise or counterclockwise order. Moreover, our algorithm reports how the contours are nested; see Section 4.4 for details.

Answering a contour query We can preprocess $\mathbf{M}$, using $O(\text{Sort}(N))$ I/Os, into a linear-size data structure so that all contours at a given level can be reported using $O(\log_B N + T/B)$ I/Os, where T is the output size. Each contour is generated individually with its edges sorted in clockwise or counterclockwise order (Section 4.4.5).

This Chapter does not discuss how contours can be simplified by removing insignificant contours and smoothing the remaining ones. The technique described in Section 3.5.4 removes insignificant contours based on their volume, area and/or persistence [C1, 20, 100].

4.2 Preliminaries

For the purpose of this Chater we give a more careful definition of what a terrain is and we will restrict the definition to triangulations.

Let $\mathbb{M} = (V, E, F)$ be a triangulation of $\mathbb{R}^2$, with vertex, edge, and face (triangle) sets V, E, and F, respectively. We assume that V contains a vertex v_∞, set at infinity, and that each edge $\{u, v_\infty\}$ is a ray emanating from u. The triangles in $\mathbb{M}$ incident to v_∞ are unbounded. Let $h : \mathbb{R}^2 \to \mathbb{R}$ be a continuous *height function* with the property that the restriction of h to each triangle of $\mathbb{M}$ is a linear map. Given $\mathbb{M}$ and h, the "graph" of h is a *terrain* $\mathbf{M} = (\mathbb{M}, h)$ which describes an xy-monotone triangulated surface in $\mathbb{R}^3$ whose triangulation is induced by $\mathbb{M}$. That is, vertices, edges, and faces of $\mathbf{M}$ are in one-to-one correspondence with those of $\mathbb{M}$. With a slight abuse of notation, in what follows we write V, E, and F, to respectively refer to the sets of vertices, edges, and triangles of both the terrain $\mathbf{M}$ and its underlying plane triangulation $\mathbb{M}$.

For convenience we assume that $h(u) \neq h(v)$ for all vertices $u \neq v$, and that $h(v_\infty) = -\infty$. Within each bounded triangle $f \in F$, h is uniquely determined

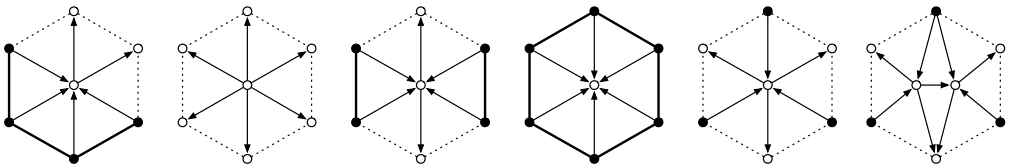


Figure 4.1: Link of a vertex; lower link is depicted by filled circles and bold edges. The type of a vertex is determined by its lower link.

as the linear interpolation of the height of the vertices of f. This is not the case for an unbounded face f since interpolation using $h(v_\infty) = -\infty$ is undefined; in which case to determine h on f an extra parameter, such as the height of a point in f, is needed.

For a given terrain $\mathbf{M}$ and a *level* $\ell \in \mathbb{R}$, the *ℓ-level set* of $\mathbf{M}$, denoted by $\mathbb{M}_\ell$, is defined as $h^{-1}(\ell) = \{x \in \mathbb{R}^2 \mid h(x) = \ell\}$. Equivalently, $\mathbb{M}_\ell$ is the vertical projection of $\mathbf{M} \cap z_\ell$ on the xy-plane, where z_ℓ is the horizontal plane $z = \ell$. The *closed ℓ-sublevel* and *ℓ-superlevel sets* of $\mathbf{M}$ are defined respectively as $\mathbb{M}_{\leq \ell} = h^{-1}((-\infty, \ell])$ and $\mathbb{M}_{\geq \ell} = h^{-1}([\ell, +\infty))$, and the *open ℓ-sublevel* and *ℓ-superlevel sets* $\mathbb{M}_{<\ell}$ and $\mathbb{M}_{>\ell}$ are $\mathbb{M}_{\leq \ell} \setminus \mathbb{M}_\ell$ and $\mathbb{M}_{\geq \ell} \setminus \mathbb{M}_\ell$, respectively. For any $R \subseteq \mathbb{R}^2$, let $\mathbf{M}(R)$ denote the subset of the surface $\mathbf{M}$ whose vertical projection into the xy-plane is R, i.e. $\mathbf{M}(R) = \{(x, y, z) \in \mathbf{M} : (x, y) \in R\}$. We shall also use the shorthand notations of $\mathbf{M}_\ell$, $\mathbf{M}_{<\ell}$, etc, for $\mathbf{M}(\mathbb{M}_\ell)$, $\mathbf{M}(\mathbb{M}_{<\ell})$, etc, respectively.

In much of what follows we need to compare the heights of two neighboring vertices of a terrain $\mathbb{M}$. To simplify the exposition we "orient" each edge of $\mathbb{M}$ toward its *higher* endpoint, and treat $\mathbb{M}$ as a directed triangulation in which a directed edge (u, v) indicates that $h(u) < h(v)$.

The *dual* graph $\mathbb{M}^* = (F^*, E^*, V^*)$ of the triangulation $\mathbb{M}$ is defined as the planar graph that has a vertex $f^* \in F^*$ for each face $f \in F$, called the *dual* of f. For any directed edge $e \in E$, there is a directed *dual edge* $e^* = (f_1^*, f_2^*) \in E^*$ where f_1 and f_2 are the faces to the left and to the right of e respectively. The graph $\mathbb{M}^*$ is naturally embedded in the plane as follows: the vertex f^* is placed inside the face f and e^* is is drawn as a curve that crosses e but no other edges of $\mathbb{M}$. A vertex $v \in V$ leads to a dual face v^* in $\mathbb{M}^*$ that is bounded by the duals of the edges incident to v. The dual of $\mathbb{M}^*$ is $\mathbb{M}$ itself. For a given subset V_0 of V, we use the notation V_0^* to refer to the set of duals to the vertices in V_0, i.e., $V_0^* = \{v^* : v \in V_0\}$. A similar notation is also used for subsets of F or E.

Links and critical points For a vertex v of $\mathbb{M}$, the *link* of v, denoted by $\mathrm{Lk}(v)$, is the cycle in $\mathbb{M}$ consisting of the vertices adjacent to v, as joined by the edges from the triangles incident upon v. The *lower link* of v, $\mathrm{Lk}^-(v)$, is the subgraph of $\mathrm{Lk}(v)$ induced by vertices lower (of smaller height) than v. The *upper link* of v, $\mathrm{Lk}^+(v)$ is defined analogously; see Figure 4.1.

If a level parameter ℓ varies continuously along the real line, the topology

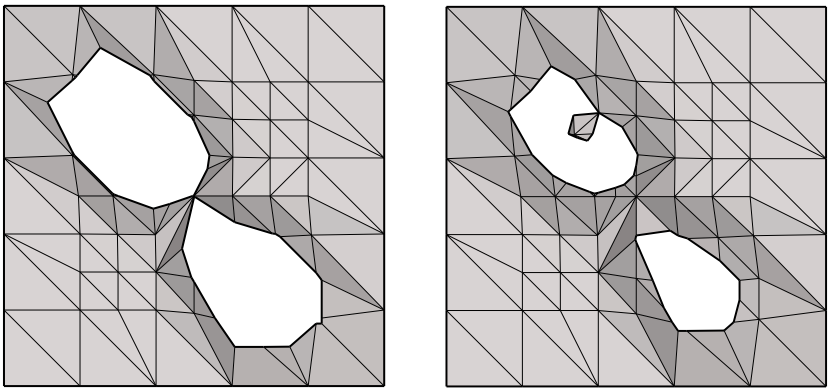


Figure 4.2: Examples of sublevel sets of negative (left) and positive (right) saddle points.

of $\mathbb{M}_{\leq \ell}$ changes only at a discrete set $\{\ell_1, \ldots, \ell_m\}$ of *critical levels* of h, where each ℓ_i is $h(v_i)$ for some vertex $v_i \in V$. $v_1, \ldots, v_m$ are *critical vertices* of $\mathbb{M}$. A non-critical level of h is also called *regular*. Vertices with regular heights are *regular vertices*. By our assumption that the height of every vertex is distinct, there is only one critical vertex at each critical level.

There are three types of critical vertices: *minima*, *saddles*, and *maxima*. The type of a vertex v can be determined from the topology of $\mathrm{Lk}^-(v)$: v is minimum, regular, saddle, or maximum if $\mathrm{Lk}^-(v)$ is empty, a path, two or more paths, or a cycle, respectively. We assume that all saddles are *simple*, meaning that the lower link of each saddle consists of precisely two paths. Multifold saddles can be split symbolically into simple saddles; see Figure 4.1. Equivalently, a vertex can be classified based on the clockwise ordering of its incoming and outgoing edges: a minimum has no incoming edges, and a maximum has no outgoing edges. For other vertices v, we count the number of times incident edges switch between incoming to outgoing as we scan them around v in clockwise order. This number is always even. Two switches indicate that v is regular while four or more switches take place if v is a saddle.

A saddle vertex v is further classified into two types. At $\ell = h(v)$ the topology of $\mathbb{M}_{\leq \ell}$ differs from that of $\mathbb{M}_{<\ell}$ in one of two possible ways: either two connected components of $\mathbb{M}_{<\ell}$ join at v to become the same connected component in $\mathbb{M}_{\leq \ell}$, or the boundary of the same connected component of $\mathbb{M}_{<\ell}$ "pinches" at v introducing one more "hole" in $\mathbb{M}_{\leq \ell}$. Saddles of the former type are *negative* saddles and those of the latter type are *positive* saddles; see Figure 4.2. It is well-known that the number of minima (resp. maxima) is one more than the number of negative (resp. positive) saddles, and therefore

$$\#\text{saddles} = \#\text{minima} + \#\text{maxima} - 2. \tag{4.1}$$

This classification of saddles is related to persistent homology and a more general statement is proved in [53].

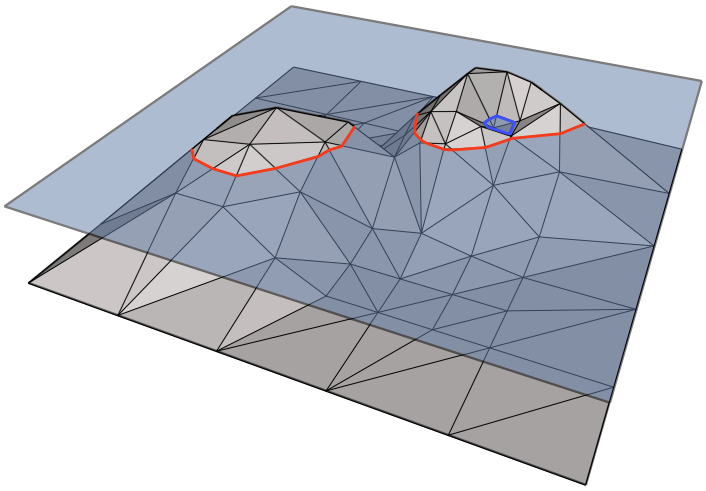
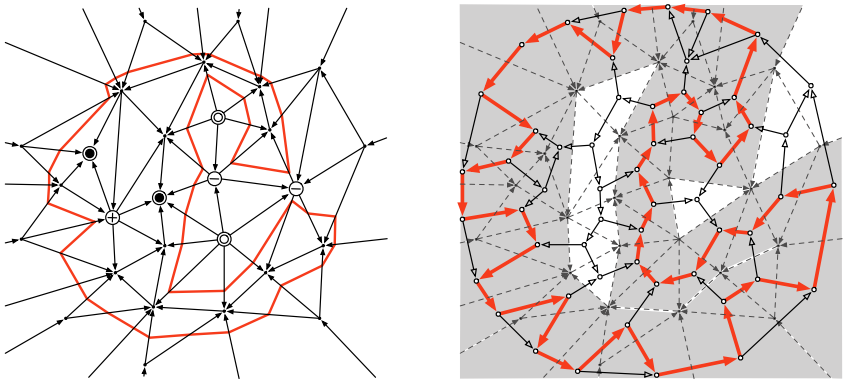


Figure 4.3: Red and blue contours in a level-set of a terrain.

Contours A *contour* of a terrain $\mathbf{M}$ is a connected component of a level set of $\mathbf{M}$. Each contour K at a regular level is a simple closed curve and partitions $\mathbb{R}^2 \setminus K$ into two open sets: a bounded one called *inside* of K and denoted by K^{i}, and an unbounded one called *outside* of K and denoted by K^{o}. This is violated at critical levels at which a contour may shrink into a point (an extremum), or may consists of two simple closed curves whose intersection is the critical point (a saddle). When the level parameter ℓ scans the open interval between two consecutive critical values of h the contours of $\mathbb{M}_\ell$ change continuously and the topology of $\mathbb{M}_\ell$ remains unchanged. However, at a critical level the contours that contain the corresponding critical point undergo topological changes. Let K_1 and K_2 be two contours at levels ℓ_1 and ℓ_2 respectively with $\ell_1 < \ell_2$. We regard K_1 and K_2 as "the same" if K_1 continuously deforms into K_2 (without any topological changes), as z_ℓ sweeps $\mathbb{M}$ in the interval $[\ell_1, \ell_2]$.

Following [3], we call a contour K in $\mathbb{M}_\ell$ *blue* if, "locally", $\mathbb{M}_{<\ell}$ lies in K^{i}, and *red* otherwise; see Figure 4.3. Every blue contour is born as a single point at a minimum. Conversely, a blue contour is born at every minimum except at v_∞. Because of being placed at infinity, a red contour is born at v_∞. Likewise, a red contours "dies" by shrinking into a single point at a maximum, and conversely, some red contour dies at every maximum. Two contours, with at least one of them being blue, merge into the same contour at a negative saddle. The resulting contour is red if one of the merging contours is red, and blue otherwise. A contour splits into two contours at a positive saddle. A red contour splits into two red contours while a blue contour splits into one red and one blue contour.

Two contours K_i and K_j of a level set $\mathbb{M}_\ell$ are called *neighbors* if no other contour K of $\mathbb{M}_\ell$ separates them, i.e., one of K_i and K_j is contained in K^{i} and the other in K^{o}. If K_i is neighbor to K_j and $K_i \subset K_j^{\mathrm{i}}$, then K_i is called a *child* of K_j. If $K_i \subset K_j^{\mathrm{o}}$ and $K_j \subset K_i^{\mathrm{o}}$ then K_i is called a *siblings* of K_j. It can be verified that all children of a red (resp. blue) contour are blue (resp. red)



(a) (b)

Figure 4.4: (a) Orientation of the edges in the plane triangulation $\mathbb{M}$ of a terrain. Critical points and a contour K in a regular level set are shown. (b) the dual $\mathbb{M}^*$ of $\mathbb{M}$. The representing cycle of K in $\mathbb{M}^*$ is shown with bold edges. Triangles in $C(K, \mathbb{M})$ are shaded.

contours while all siblings of a red (resp. blue) contour are red (resp. blue) contours.

We conclude this section by making a key observation, which is crucial for our main result. Each regular contour of $\mathbf{M}$ corresponds to a cycle in $\mathbb{M}^*$: let K be a contour in an arbitrary level set $\mathbb{M}_\ell$, and let $F(K)$ (resp. $E(K)$) denote the set of faces (resp. edges) of $\mathbb{M}$ that intersect K. If K is a red (resp. blue) contour, all the edges in $E(K)$ are oriented toward K^{i} (resp. K^{o}). Consequently, the vertices in $F^*(K)$ are linked by the edges in $E^*(K)$ into a cycle in $\mathbb{M}^*$. We refer to this cycle as the *representing cycle of* K. We use $C(K, \mathbb{M})$ to denote the circular sequence of triangles dual to the representing cycle of K in $\mathbb{M}$. The sequence in $C(K, \mathbb{M})$ is oriented clockwise (resp. counterclockwise) if K is blue (resp. red); see Figure 4.4.

4.3 Level-ordering of Triangles

In this section we present our main result, i.e. the existence of a level-ordering on triangles of any terrain $\mathbf{M}$, i.e., an ordering that satisfies conditions (C1) and (C2). We begin by proving the existence of a level-ordering for terrains that do not have saddle vertices. We call such terrains *basic*. Next we prove certain structural properties of terrains and show that any arbitrary terrain can be transformed into a basic terrain through a *surgery* that effectively "preserves" the contours of the original terrain. We then argue that a level-ordering on the transformed terrain corresponds to a level-ordering on the original one.

4.3.1 Basic terrain

Let $\mathbf{M}$ be a basic terrain. The above discussion and (4.1) imply that $\mathbf{M}$ has one (global) minimum, $\check{v}$, which coincides with v_∞, and one (global) maximum,

$\hat{v}$, and that every level set consists of a single red contour. At $\hat{v}$ this contour collapses into a single point.

Lemma 4.1 *Let $P \subset E$ be a directed (monotone) path in $\mathbb{M}$ from $\check{v}$ to $\hat{v}$. Then every cycle of $\mathbb{M}^*$ contains exactly one edge from P^*. In particular, the graph $\mathbb{M}^* \setminus P^*$ obtained from deleting the edges in P^* from $\mathbb{M}^*$ is acyclic.*

Proof. We claim that $\hat{v}$ is reachable in $\mathbb{M}$ from every vertex $v \in V$. Recall that $\hat{v}$ is the only local maximum in $\mathbb{M}$ and that every other vertex has at least one outgoing edge. If one starts at v and follows an arbitrary outgoing edge at each step, the height of the vertex at which we arrive is greater than that of the previous one. This process can only stop at $\hat{v}$. By a similar argument, every vertex $v \in V$ is reachable from $\check{v}$.

Consider an arbitrary cycle C^* in $\mathbb{M}^*$. In the plane drawing of $\mathbb{M}^*$, C^* is a Jordan curve. Let $V_0 \subset V$ be the set of vertices that are contained in the inside of C^* (equivalently, $V_0^* \subset V^*$ is the set of faces of $\mathbb{M}^*$ whose union is bounded by C^*). Let $C \subset E$ be the set of edges in $\mathbb{M}$ dual to those in C^*. Since C^* is a cycle, the edges in C are either all oriented from V_0 to $V \setminus V_0$ or all from $V \setminus V_0$ to V_0.

The former case cannot happen because $v_\infty \notin V_0$ and every vertex in V is reachable from v_∞. If all edges of C are oriented from $V \setminus V_0$ to V_0, then $\hat{v} \in V_0$ because otherwise $\hat{v}$ cannot be reachable from the vertices of V_0. Since $\hat{v} \in V_0$ and $v_\infty \in V \setminus V_0$, $|P \cap C| \geq 1$. There is no edge directed from V_0 to $V \setminus V_0$, so once P reaches a vertex of V_0 it cannot leave V_0, implying that $|P \cap C| = 1$. Thus, every cycle of $\mathbb{M}^*$ is destroyed by the removal of the edges in P^*, implying that $\mathbb{M}^* \setminus P^*$ is acyclic. □

Let P be the path from $\check{v}$ to $\hat{v}$ as defined in Lemma 4.1. The graph $\mathbb{M}^* \setminus P^*$ has all of the vertices of $\mathbb{M}^*$. Thus every face f of $\mathbb{M}$ is represented by f^* in $\mathbb{M}^* \setminus P^*$. Let $\prec$ be the a binary relation on F (triangles in $\mathbb{M}$) defined as $f_1 \prec f_2$ if $(f_1^*, f_2^*) \in E^* \setminus P^*$. Since by Lemma 4.1 $\mathbb{M}^* \setminus P^*$ is acyclic, $\prec$ is a partial order on F. We call $\prec$ the *adjacency partial order* induced by the acyclic graph $\mathbb{M}^* \setminus P^*$. A linear extension of $\prec$ is any total order $\lhd$ on F that is consistent with $\prec$, i.e. $f_1 \prec f_2$ implies $f_1 \lhd f_2$. Such a linear extension can be obtained by topological sorting of $\mathbb{M}^* \setminus P^*$. By definition, the existence of a directed path from f_i^* to f_j^* in $\mathbb{M}^* \setminus P^*$ implies that $f_i \lhd f_j$. Thus condition (C1) of the definition of level ordering holds for $\lhd$. Since in a basic terrain each level consists of only one contour, condition (C2) holds trivially. This results the following statement.

Corollary 4.1 *Let $\mathbf{M}$ be a basic terrain, and let P be a directed (monotone) path from $\check{v}$ to $\hat{v}$ in $\mathbb{M}$. Let $\lhd$ be a linear extension of the adjacency partial order induced by $\mathbb{M}^* \setminus P^*$. Then $\lhd$ is a level-ordering of the triangles of $\mathbf{M}$.*

4.3.2 Red and blue cut-trees

Consider now a non-basic terrain with saddle vertices. We first introduce the notions of *ascending (red)* and *descending (blue) cut-trees* of $\mathbb{M}$ as subgraphs

of the triangulation $\mathbb{M}$, which we later use to turn $\mathbf{M}$ into a basic terrain $\tilde{\mathbf{M}}$. Contours of each level set of $\mathbf{M}$ will then be encoded in a corresponding level set of $\tilde{\mathbf{M}}$ which consists of a single contour.

A *descending* (resp. *ascending*) path on $\mathbb{M}$ from a vertex $v \in V$ is a path $v_0, v_1, \ldots, v_r$ where $v_0 = v$ and $h(v_i) < h(v_{i-1})$ (resp. $h(v_i) > h(v_{i-1})$) for $i = 1, \ldots, r$. For each negative saddle v, let $P_1(v) = u_0, u_1, \ldots, u_r$ and $P_2(v) = w_0, \ldots, w_s$ be two descending paths from v such that u_r and w_s are both minima and u_1 and w_1 belong to different connected components of $\mathrm{Lk}^-(v)$. Furthermore assume that for any two negative saddles u and w, if $P_i(u) = u_0, \ldots, u_r$ and $P_j(w) = w_0, \ldots, w_s$, for some $i, j \in \{1, 2\}$, and $u_k = w_l$ for some $1 \leq k \leq r$ and $1 \leq l \leq s$, then $u_{k+1} = w_{l+1}$; in other words, descending paths from different vertices can join but then cannot diverge. Such a set of paths always exist: one can assign such paths to negative saddles in the increasing order of their heights. At any negative saddle u, we follow a descending paths through each of the two connected components of $\mathrm{Lk}^-(u)$ until it either reaches a minimum or joins a path already assigned to a lower negative saddle. Let $P(u) = P_1(u) \cup P_2(u)$ for any negative saddle u. Since $P_1(u) \setminus \{u\}$ and $P_2(u) \setminus \{u\}$ are contained in different connected components of $\mathbb{M}_{<h(u)}$, the underlying undirected graph of $P(u)$ is a simple path. For a positive saddle u, $P_1(u)$ and $P_2(u)$ are defined similarly using ascending paths that start at different connected components of $\mathrm{Lk}^+(u)$ and end in maxima.

We define the *descending (blue) cut-tree* $\check{\mathbb{T}} = (\check{V}, \check{E})$ of $\mathbf{M}$ to be the union of the paths $P(u)$ over all negative saddles u. Similarly, we define the *ascending (red) cut-tree* $\hat{\mathbb{T}} = (\hat{V}, \hat{E})$ of $\mathbf{M}$ to be the union of all the paths $P(u)$ over all positive saddles u. It is, of course, not clear that $\check{\mathbb{T}}$ and $\hat{\mathbb{T}}$ are trees but this and some of their other properties are proven below.

Remark. The definitions of red and blue cut-trees are closely related to the notions of *split* and *join trees* in the context of *contour trees* [114]. Intuitively the contour tree is the result of contracting each contour of $\mathbf{M}$ into a single point [1] and is topologically a connected collection of simple curves whose endpoints correspond to the critical points of $\mathbf{M}$. These curves can only intersect at their endpoints and thus realize the edges of a graph on a set of vertices that correspond to the critical points of $\mathbf{M}$. It can be shown that this graph is always connected and acyclic — hence the name contour tree.[2] Each point on a curve realizing an edge represents a contour at some height. The heights of points along any edge of the contour tree vary monotonically from one end to the other. The contour tree is often described as the union of two of its subtrees, namely the merge and split trees. The join tree is the minimal connected subtree of the contour tree that contains minima and negative saddles and the split tree is defined analogously using maxima and positive saddles.

[1] Taking the terrain $\mathbf{M}$ as a topological space with the usual topology of $\mathbb{R}^2$ and defining an equivalence relation $\sim$ between points on $\mathbf{M}$ as $x \sim y$ if and only if x and y are on the same contour (connected component of some level set), the contour tree $\mathbf{M}_\sim$ of $\mathbf{M}$ is the quotient space of $\mathbf{M}$ modulo $\sim$.

[2] Indeed such graphs, known generally as *Reeb graphs* [98], can be obtained from arbitrary continuous real valued functions defined on more general topological spaces such as arbitrary manifolds. Contour trees are Reeb graphs of terrains as determined by their height functions.

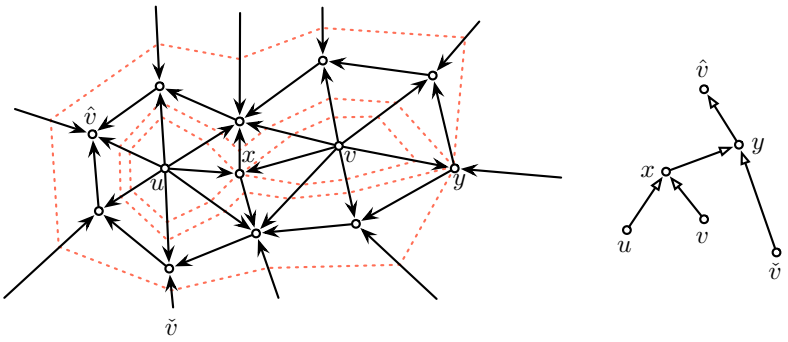


Figure 4.5: A terrain for which the join tree cannot be embedded as a subgraph of the underlying triangulation in such a way that the edges are realized by ascending paths. Level sets of saddles are depicted in dotted lines. The contour tree of the terrain on the left is shown on the right. Notice that there is no directed (ascending) path from x to y on the terrain.

The similarity between the notions of blue (resp. red) cut-tree and join (resp. split) tree naturally poses the question of whether our cut-trees can be replaced by their contour tree counterparts. We emphasize here that our cut-trees are subgraphs of the triangulation $\mathbb{M}$ and this plays a crucial role in our algorithms. It is possible to draw the contour tree on the terrain in such a way that the vertices coincide with their corresponding critical points and edges are realized by monotonically ascending curves on the terrain. It is easy to observe that if one can realize each edge of the join or split tree as a monotonically ascending path in $\mathbb{M}$ then it is indeed possible to simply use the merge or split trees in place of our cut-trees. However, this is not always possible as the terrain depicted in Figure 4.5 demonstrates.

Lemma 4.2 *The underlying undirected graph of a blue (resp. red) cut-tree $\check{\mathbb{T}}$ (resp. $\hat{\mathbb{T}}$) has no cycles.*

Proof. We prove the claims for blue cut-tree. The argument for red cut-tree can be made symmetrically. Let $u_1, \ldots, u_r$ be the list of all negative saddles of $\mathbb{M}$ in the increasing order of their heights. Let $\check{\mathbb{T}}_0$ be the empty graph and for each $i = 1, \ldots, r$, let $\check{\mathbb{T}}_i = \bigcup_{j=1}^{i} P(u_j)$; $\check{\mathbb{T}}_{i-1}$ is a subgraph of $\check{\mathbb{T}}_i$, and $\check{\mathbb{T}}_r = \check{\mathbb{T}}$. We prove by induction on i that the underlying undirected graph of each $\check{\mathbb{T}}_i$ is a forest. This trivially holds for $\check{\mathbb{T}}_0$. Assume now that the underlying undirected graph of $\check{\mathbb{T}}_i$ is a forest. By construction, adding $P(u_{i+1})$ connects two distinct connected components of $\check{\mathbb{T}}_i$, one contained in each of the two distinct connected components of $\mathbb{M}_{<h(u_{i+1})}$ that join at u_{i+1}.

Moreover, once each of $P_1(u_{i+1})$ or $P_2(u_{i+1})$ reaches a vertex of $\check{\mathbb{T}}_i$, it continues by following a path contained in $\check{\mathbb{T}}_i$, and therefore does not introduce a cycle within the corresponding connected component of $\check{\mathbb{T}}_i$. □

For a set $U \subseteq \mathbb{R}^2$, let $\check{\mathbb{T}}(U)$ (resp. $\hat{\mathbb{T}}(U)$) be the union of the paths $P(u)$ for all negative (resp. positive) saddles $u \in U$. In particular, $\check{\mathbb{T}} = \check{\mathbb{T}}(\mathbb{R}^2)$ and

$\hat{\mathbb{T}} = \hat{\mathbb{T}}(\mathbb{R}^2)$.

Lemma 4.3 *For a blue (resp. red) contour K, the underlying undirected graph $\check{\mathbb{T}}(K^i)$ (resp. $\check{\mathbb{T}}(K^o)$) connects all of the minima in K^i (resp. K^o). A symmetric statement holds regarding $\hat{\mathbb{T}}$ and maxima by switching "red" and "blue".*

Proof. We prove the lemma for $\check{\mathbb{T}}$ and blue contours. The other cases are similar. Let K be a blue contour in $\mathbb{M}_\lambda$ for some $\lambda \in \mathbb{R}$. We show that for each $\ell \in \mathbb{R}$, the minima in each connected component of $U_\ell = \mathbb{M}_{<\ell} \cap K^{\mathrm{i}}$ are connected by $\check{\mathbb{T}}(U_\ell)$. The statement of the lemma then follows by taking ℓ to be larger than the height of all vertices in K^{i} and from the fact that in that case U_ℓ consists of a single component.

To prove the lemma we sweep ℓ from $-\infty$ toward $+\infty$ and verify the claim for U_ℓ. Every time ℓ reaches the height of a minimum in K^{i}, a new connected component is added to U_ℓ. The lemma holds for this new component since it originally has only a single minimum which is vacuously connected by $\check{\mathbb{T}}(U_\ell)$ to every other minimum in that component. The validity of the claim as ℓ continues to raise can only be altered when ℓ reaches the height of a negative saddle u in K^{i} at which two connected components U_1 and U_2 of $U_{<\ell}$, where $\ell = h(u)$, join at u. At this time the path $P(u)$ is added to $\check{\mathbb{T}}(U_\ell)$. The crucial observation here is that because K is a blue contour, no descending path started at a vertex $u \in K^{\mathrm{i}}$ can reach K^{o}. Thus the endpoints of $P(u)$ have to be minima in K^{i}. In other words $P(u)$, which reaches a minimum in U_1 and another in U_2, connects $\check{\mathbb{T}}(U_1)$ and $\check{\mathbb{T}}(U_2)$ as desired. □

Corollary 4.2 *The underlying undirected graphs of $\check{\mathbb{T}}$ and $\hat{\mathbb{T}}$ are trees. Moreover, all minima are vertices of $\check{\mathbb{T}}$ and all maxima are vertices of $\hat{\mathbb{T}}$.*

We conclude this discussion by mentioning a property of $\hat{\mathbb{T}}$ and $\check{\mathbb{T}}$ that follows from their constructions.

Lemma 4.4 *Let u be a vertex of $\hat{\mathbb{T}}$ (resp. $\check{\mathbb{T}}$). If u is a positive (resp. negative) saddle, then u has two outgoing (resp. incoming) edges in $\hat{\mathbb{T}}$ (resp. $\check{\mathbb{T}}$) — one to each connected component of the upper (resp. lower) link of u in $\mathbb{M}$. If u is a regular vertex or a negative saddle, then u has one outgoing (resp. incoming) edge. Finally, if u is a maximum (resp. minimum), then it has no outgoing (resp. incoming) edges.*

4.3.3 Surgery on terrain

Let $\hat{\mathbb{T}}$ be a red cut-tree for $\mathbf{M}$. Consider the following combinatorial operation on $\mathbb{M}$. First we duplicate every edge e of $\hat{\mathbb{T}}$, thus creating a face f_e that is bounded by the two copies of e. We then perform an Eulerian tour on the subgraph of $\mathbb{M}$ induced by the copies of the edges of $\hat{\mathbb{T}}$ in which at each vertex the next edge of the tour is the first unvisited edge of the subgraph in clockwise order, relative to the previous edge of the tour. We then combine all of the faces

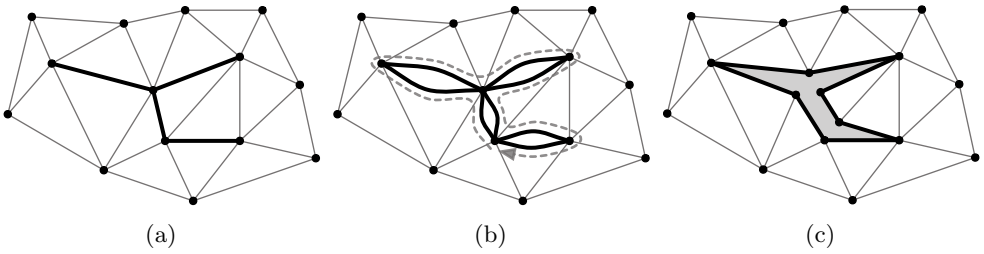


Figure 4.6: Cutting a triangulation along a tree.

f_e into a single face $\hat{f}$ that is bounded by this Eulerian tour by making as many copies of each vertex as its degree in $\hat{\mathbb{T}}$ (or equivalently the number of times the tour has passed through it) and connecting non-tree edges incident on u to appropriate copies of u; see Figure 4.6. Geometrically, the above modification of the terrain triangulation can be interpreted as "puncturing" the plane along the edges of $\hat{\mathbb{T}}$ and introducing a new face $\hat{f}$ bounded by the $2|\hat{E}|$ edges in the Euler tour.

We then subdivide $\hat{f}$ by placing a new vertex $\hat{v}$ inside it and connecting $\hat{v}$ via incoming edges $(u, \hat{v})$ to every vertex u on the boundary of $\hat{f}$. The result is a triangulation $\mathbb{M}_0 = (V_0, E_0, F_0)$; see Figures 4.7 (a) and (b). The newly added triangles are all incident to $\hat{v}$, and we refer to them as $\hat{v}$-triangles. The edge e opposite to $\hat{v}$ in a $\hat{v}$-triangle f (which is a copy of a $\hat{\mathbb{T}}$ edge) is called the *base* of f and f is said to be *based* at e. One can modify the plane drawing of $\mathbb{M}$ into a (singular) plane drawing of $\mathbb{M}_0$, that has faces of zero area and edges that bend and overlap, by jamming all the new faces and edges in the (zero-area) hole that results from cutting the plane along $\hat{\mathbb{T}}$.

$\mathbb{M}_0$ can be regarded as the triangulation of a terrain $\mathbf{M}_0$: Fáry's theorem [55] can be used to straight-line embed $\mathbb{M}_0$ while preserving all its faces and the height function of $\mathbf{M}$ induces a height function on triangles of $\mathbb{M}_0$ that are also in $\mathbb{M}$. The height of $\hat{v}$ is then chosen to be higher than the heights of all vertices of $\mathbf{M}$ and is used to linearly interpolate a height function on $\hat{v}$-triangles.

Lemma 4.5 *$\mathbf{M}_0$ has no positive saddles and exactly one maximum, namely $\hat{v}$. The minima of $\mathbf{M}_0$ are precisely those of $\mathbf{M}$. Each negative saddle of $\mathbf{M}_0$ is a copy of a negative saddle of $\mathbf{M}$, and only one copy of each negative saddle of $\mathbf{M}$ is a negative saddle of $\mathbf{M}_0$.*

Proof. For a vertex $u \notin \hat{\mathbb{T}}$, $\mathrm{Lk}(u)$ is the same in $\mathbb{M}$ and $\mathbb{M}_0$ modulo taking copies of $\hat{\mathbb{T}}$ vertices as identical. In particular, minima of $\mathbf{M}$ stay minima in $\mathbf{M}_0$. Thus it suffices to consider $\hat{v}$ and copies of $\hat{\mathbb{T}}$ vertices. Clearly, $\hat{v}$ is a maximum. Let u be a vertex of $\hat{\mathbb{T}}$, and let u' be a copy of u in $\mathbb{M}_0$. Let e'_1 and e'_2 be copies of $\hat{\mathbb{T}}$ edges that enter and leave u', respectively, in the Eulerian tour of $\hat{\mathbb{T}}$. Both of these edges remain incident to u' in $\mathbb{M}_0$. Let v'_1 and v'_2 be other endpoints of e'_1 and e'_2 in $\mathbb{M}_0$, respectively. Let v_i and e_i, $i = 1, 2$, be the vertex and edge in $\mathbb{M}$ corresponding to v'_i and e'_i, respectively. $\mathrm{Lk}(u')$ consists of a path $\pi(u')$ from v'_1 to v'_2 followed by $\hat{v}$. Moreover, e'_1 and e'_2 are the only edges incident on u' that

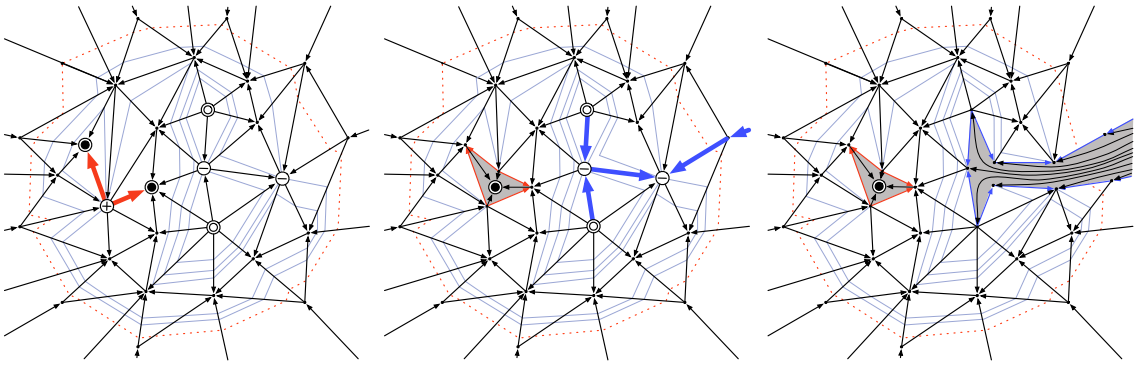


Figure 4.7: (a) A red (ascending) cut-tree marked $\hat{\mathbb{T}}$ marked on the terrain $\mathbb{M}$ of Figure 4.4. (b) Construction of the graph $\mathbb{M}_0$: the terrain is cut along $\hat{\mathbb{T}}$ and a new maximum $\hat{v}$ is inserted in the opened face. On the right, a blue cut-tree of $\mathbb{M}_0$ is marked. (c) Construction of the graph $\tilde{\mathbb{M}}$: the terrain is cut open on the red cut-tree and a new maximum is inserted.

are copies of $\hat{\mathbb{T}}$ edges, and $\pi(u')$ is also a path in $\mathrm{Lk}(u)$ in $\mathbb{M}$, modulo taking copies of $\hat{\mathbb{T}}$ vertices as identical.

First, u' cannot be a maximum because u' is adjacent to $\hat{v}$. It cannot be a minimum either because then $\pi(u') \subseteq \mathrm{Lk}^+(u)$ and e_1 and e_2 are outgoing edges from u in $\hat{\mathbb{T}}$ connected to some component of $\mathrm{Lk}^+(u)$, which contradicts Lemma 4.4. Next, if $\mathrm{Lk}^+(u')$ is not connected, then its component U not containing $\hat{v}$ does not contain v'_1 and v'_2 either and thus u lies in the interior of the path $\pi(u')$. Then U is also a connected component of $\mathrm{Lk}^+(u)$ in $\mathbb{M}$. Unless u is a negative saddle, by Lemma 4.4, there is an outgoing edge in $\hat{\mathbb{T}}$ from u to a vertex in U, contradicting the fact that e'_1 and e'_2 are the only edges adjacent to u' that are copies of $\hat{\mathbb{T}}$ edges. Hence, unless u is a negative saddle, $\mathrm{Lk}^+(u')$ is connected and u' is a regular vertex in $\mathbb{M}_0$.

Finally, suppose u is a negative saddle, with two components U_1 and U_2 in $\mathrm{Lk}^+(u)$. By Lemma 4.4, u has exactly one outgoing edge e in $\hat{\mathbb{T}}$. Without loss of generality assume that e is connected to U_1. Then U_2 will appear as a connected component of the upper link of exactly one copy u' of u, namely if $U \subseteq \pi(u')$, and u' will be a negative saddle in $\mathbb{M}_0$. The upper link of all other copies of u will be connected — consisting of $\hat{v}$ and possibly a portion of U_1. Consequently, one copy of every negative saddle of $\mathbf{M}$ becomes a negative saddle in $\mathbf{M}_0$ and other copies become regular vertices. This completes the proof of the lemma. □

Next we perform a similar surgery on $\mathbf{M}_0$ only using a blue cut-tree $\check{\mathbb{T}}$ of $\mathbf{M}_0$. As above, the idea is to slice the plane along $\check{\mathbb{T}}$ and insert a new vertex $\check{v}$ in the resulting face and connect $\check{v}$ to every copy u of a vertex in $\check{\mathbb{T}}$ by an outgoing edge $(\check{v}, u)$. We call the resulting triangulation $\tilde{\mathbb{M}} = (\tilde{V}, \tilde{E}, \tilde{F})$. A slight technicality arises in this case as a result of the fact that v_∞ is a minimum of $\mathbb{M}_0$ which by Corollary 4.2 is a vertex of $\check{\mathbb{T}}$. As it will become clear later, we only need to treat $\check{v}$ symbolically below v_∞ by connecting them by an edge oriented toward

v_∞. We conclude, using the same argument as in Lemma 4.5, the following:

Lemma 4.6 *$\tilde{\mathbb{M}}$ does not have saddle vertices.*

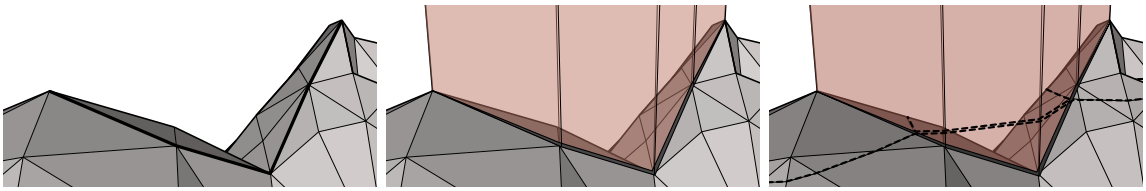
Lemma 4.7 *If (f_1^*, f_2^*) is an edge of $\mathbb{M}^*$, then there is a path from f_1^* to f_2^* in $\tilde{\mathbb{M}}^*$.*

Proof. If f_1 and f_2 are adjacent in $\tilde{\mathbb{M}}$ then (f_1^*, f_2^*) is an edge in $\tilde{\mathbb{M}}^*$. Thus we only need to consider the case in which an edge e shared by f_1 and f_2 in $\mathbb{M}$ is an edge of $\hat{\mathbb{T}}$ or $\check{\mathbb{T}}$ (or both). Suppose e is an edge of $\hat{\mathbb{T}}$. In constructing $\mathbb{M}_0$, e is duplicated to create two edges e_1 and e_2, respectively, incident to f_1 and f_2. Let ϕ_1 and ϕ_2 respectively be the $\hat{v}$-triangles based at e_1 and e_2. By construction, f_1 is to the left and ϕ_1 to the right of e_1 and therefore (f_1^*, ϕ_1^*) is an edge in $\tilde{\mathbb{M}}^*$. Similarly (ϕ_2^*, f_2^*) are edges in $\tilde{\mathbb{M}}^*$. Consider the subgraph of $\tilde{\mathbb{M}}^*$ induced by $\hat{v}$-triangles. Since all the edges incident to $\hat{v}$ are incoming, their duals make a cycle in $\tilde{\mathbb{M}}^*$ which includes ϕ_1^* and ϕ_2^*. Since there is a path from ϕ_1^* to ϕ_2^* on this cycle and there are edges from f_1^* to ϕ_1^* and from ϕ_2^* to f_2^* in $\tilde{\mathbb{M}}^*$, we get a path from f_1^* to f_2^*. It is easy to observe that the same argument extends to neighboring $\mathbb{M}$ triangles that are separated by the edges of $\check{\mathbb{T}}$ or both $\hat{\mathbb{T}}$ and $\check{\mathbb{T}}$. □

4.3.4 Encoding of contours in the resulting basic terrain

Although we argued above that $\tilde{\mathbb{M}}$ can be realized and therefore treated as the triangulation of a terrain $\tilde{\mathbf{M}}$, to relate the level sets of $\tilde{\mathbf{M}}$ to those of $\mathbf{M}$, in the rest of this section we use a degenerate realization of $\tilde{\mathbb{M}}$ as a surface in $\mathbb{R}^3$ that differs from what a straight-line embedding of $\tilde{\mathbb{M}}$ results. This substantially simplifies the arguments that follow. We shall realize the $\hat{v}$- and $\check{v}$-triangles as vertical *curtains*: An *upward* (resp. *downward*) *extending* curtain based at a segment pq in $\mathbb{R}^3$ is the convex hull of two infinite rays shot in positive (resp. negative) direction of the z-axis from the points p and q respectively. A curtain can be regarded as a vertical (orthogonal to the xy-plane) triangle that has a vertex at infinity.

We realize $\tilde{\mathbb{M}}$ by preserving the geometry of every triangles that existed in $\mathbf{M}$ and representing $\hat{v}$-triangles (resp. $\check{v}$-triangles) as upward (resp. downward) extending curtains based at segments corresponding to the edges of $\hat{\mathbb{T}}$ (resp. $\check{\mathbb{T}}$) on $\mathbf{M}$; see Figure 4.8. Note that in this realization of $\tilde{\mathbb{M}}$, the two copies of each $\hat{\mathbb{T}}$ or $\check{\mathbb{T}}$ edge overlap as do the segments that represent them on $\tilde{\mathbf{M}}$ and therefore the curtains tha realize their corresponding $\hat{v}$- or $\check{v}$-triangles also overlap. Although in this sense $\tilde{\mathbf{M}}$ is not the graph of a bivariate function, it can still be regarded as a (self-overlapping) piece-wise linear surface in $\mathbb{R}^3$ and a level set $\tilde{\mathbb{M}}_\ell$ of it can be defined as the projection into the xy-plane of the set $\tilde{\mathbf{M}}_\ell = \tilde{\mathbf{M}} \cap z_\ell$. Let $\hat{\mathbf{T}}$ and $\check{\mathbf{T}}$ respectively be shorthands for $\mathbf{M}(\hat{\mathbb{T}})$ and $\mathbf{M}(\check{\mathbb{T}})$. Note that $\hat{\mathbf{T}}$ and $\check{\mathbf{T}}$ are also contained in $\tilde{\mathbf{M}}$, although under the topology of this surface they are (self-overlapping) closed curves that correspond to Eulerian traversals of $\hat{\mathbf{T}}$ and $\check{\mathbf{T}}$ on $\mathbf{M}$.



(a) (b) (c)

Figure 4.8: (a) The red cut-tree $\hat{\mathbb{T}}$ of a terrain is drawn using heavier segments on the terrain $\mathbf{M}$. (b) The terrain is sliced along $\hat{\mathbb{T}}$ and $\hat{v}$-triangles are represented by upward extending curtains. Note that each edge e of $\hat{\mathbb{T}}$ results two overlapping curtains one based at each of the two (overlapping) copies of e that results from cutting the terrain along $\hat{\mathbb{T}}$. (c) A contour of the resulting terrain (dashed) overlapping itself on $\hat{v}$-triangles.

Since every triangle of $\mathbb{M}$ is also in $\tilde{\mathbb{M}}$ and is geometrically realized by the same triangle in both $\mathbf{M}$ and $\tilde{\mathbf{M}}$, $\mathbb{M}_\ell \subseteq \tilde{\mathbb{M}}_\ell$ for all $\ell \in \mathbb{R}$ and $\tilde{\mathbb{M}}_\ell \setminus \mathbb{M}_\ell \subset \hat{\mathbb{T}} \cup \check{\mathbb{T}}$ (with some abuse of notation we write $\hat{\mathbb{T}}$ and $\check{\mathbb{T}}$ to refer the red and blue cut-trees as subgraphs $\mathbb{M}$ as well as their drawings as subsets of $\mathbb{R}^2$). In other words $\tilde{\mathbb{M}}_\ell$ consists of the contours of $\mathbb{M}_\ell$ together with fragments of the red and blue cut-trees.

Lemma 4.8 *Let K_0 be a blue (resp. red) contour in a level set $\mathbb{M}_\ell$ and let $K_1, \ldots, K_r$ be its children. Let R be the interior of $K_0^i \setminus (K_1^i \cup \cdots \cup K_r^i)$. Then $\tilde{\mathbb{M}}_\ell \cap R = \hat{\mathbb{T}} \cap R$ (resp. $\check{\mathbb{T}} \cap R$).*

Proof. We prove the lemma for the case where K_0 is blue. The proof for the case where it is red is symmetric. Let Q and $\tilde{Q}$ respectively be shorthands for $\mathbf{M}(R)$ and $\tilde{\mathbf{M}}(R)$. By definition, $Q \subseteq \tilde{Q}$. Since K_0 is a blue contour of $\mathbf{M}$, Q is entirely below the plane z_ℓ. Since Q and $\tilde{Q}$ differ only in curtains based at $\hat{\mathbf{T}}$ or $\check{\mathbf{T}}$ segments, $z_\ell \cap \tilde{Q}$ is contained in such curtains. On the other hand any curtain whose intersection with $\tilde{Q}$ intersects z_ℓ has to be extending upward from some segment of $\hat{\mathbf{T}}$ that intersects Q. Thus $\tilde{\mathbb{M}}_\ell \cap R \subseteq \hat{\mathbb{T}}$. Conversely, any segment of $\hat{\mathbf{T}}$ that intersects Q is the base of an upward extending curtain which intersects z_ℓ. Thus $\hat{\mathbb{T}} \cap R \subseteq \tilde{\mathbb{M}}_\ell$. □

Let us fix a regular level ℓ of h and and let $K_1, \ldots, K_t$ be the contours in $\mathbb{M}_\ell$. For simplicity, we virtually add an infinitely large contour K_0 that bounds the entire plane. Let $\mathcal{K}_\ell = \{K_0, K_1, \ldots, K_t\}$. Consider the set $\mathcal{R}_\ell = \{R_0, R_1, \ldots, R_t\}$ of the connected component of $\mathbb{R}^2 \setminus \mathbb{M}_\ell$. The boundary of each R_i, $i \geq 0$, consists of a set $B(R_i) = \{K_{i_0}, K_{i_1}, \ldots, K_{i_{r(i)}}\} \subseteq \mathcal{K}_\ell$ of $r(i)$ contours in which $K_{i_1}, \ldots, K_{i_{r(i)}}$ are the children of K_{i_0} in the arrangement of contours in $\mathcal{K}_\ell$.

For any $R \in \mathcal{R}_\ell$ we construct an undirected graph $G_\ell(R) = (V_R, E_R)$ as follows. By Lemma 4.8, $\tilde{\mathbb{M}}_\ell \cap R$ is contained in exactly one of $\hat{\mathbb{T}}$ or $\check{\mathbb{T}}$; let $\mathbb{T}$ denote that tree. The vertex set V_R of $G_\ell(R)$ consists of all the vertices of $\mathbb{T}$ that are contained in R together with one *auxiliary vertex* v_K associated with

every contour $K \in B(R)$. The edge set E_R of $G_\ell(R)$ consists of an edge $\{u, v\}$ corresponding to each edges (u, v) of $\mathbb{T}$ whose endpoints u and v are both in R together with an edge $\{v, v_K\}$ for any edge of $\mathbb{T}$ that crosses a contour $K \in B(R)$ and its endpoint in R is v. Equivalently, $G_\ell(R)$ is obtained from the subgraph of $\mathbb{T}$ that is induced by those edges of $\mathbb{T}$ that intersect R, by identifying all vertices that are contained in each component $R' \neq R$ of $\mathcal{R}_\ell$ that is separated from R by a contour $K \in B(R)$ into a single vertex v_K.

Lemma 4.9 *For any $R \in \mathcal{R}_\ell$ the graph $G_\ell(R)$ as defined above is a tree.*

Proof. Let $K_0, K_1, \ldots, K_r$ be the contours bounding R and let K_0 be the parent of $K_1, \ldots, K_r$. We prove the lemma for the case where K_0 is blue. The proof for the case where it is red is symmetric. We prove that the existence of a cycle in $G_\ell(R)$ implies the existence of a cycle in the underlying undirected graph of $\hat{\mathbb{T}}$ which contradicts Corollary 4.2. Consider any contour K_j, $j \geq 1$ and let $e_1 = (u_1, v_1)$ and $e_2 = (u_2, v_2)$ be two edges of $\hat{\mathbb{T}}$ that intersect K_j. Since K_j is red, $v_1, v_2 \in K_j^{\mathrm{i}}$. Since e_1 is an edge of $\hat{\mathbb{T}}$, v_1 is followed in $\hat{\mathbb{T}}$ by an ascending path that ends at a maximum. Since no ascending path can leave K_j^{i}, $\hat{\mathbb{T}}$ reaches a maximum v_1' in K_j^{i} through v_1. Similarly, $\hat{\mathbb{T}}$ reaches a maximum v_2' in K_j^{i} through v_2. Lemma 4.3 implies that v_1' and v_2' are connected by a path in the underlying undirected graph of $\hat{\mathbb{T}}$ that is contained in K_j^{i}. In other words, any two branches of $\hat{\mathbb{T}}$ that enter K_j^{i} meet in K_j^{i}. Thus if we contract every edge of $\hat{\mathbb{T}}$ whose endpoints are both outside R, we precisely get the graph $G_\ell(R)$. The proof of the lemma follows from the fact that the result of contracting a tree edge is a tree. □

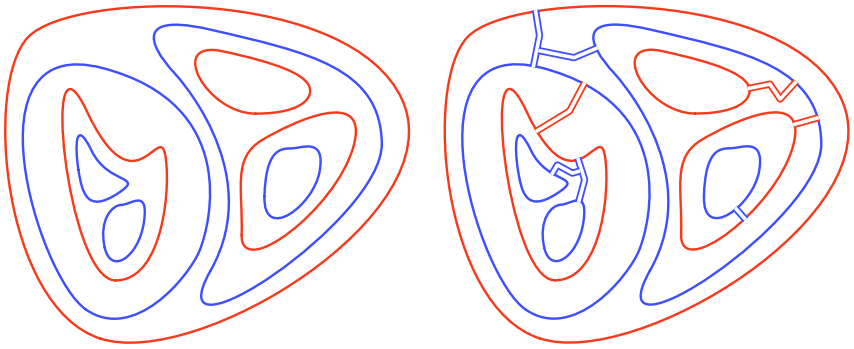
We next combine the graphs $G_\ell(R), R \in \mathcal{R}_\ell$ into a graph G_ℓ by identifying, for any two components R_1 and R_2 that share a contour K in their boundaries, the two auxiliary vertices v_K associated to K in $G_\ell(R_1)$ and $G_\ell(R_2)$. The acyclic structure of the hierarchy of red and blue contours together with Lemma 4.9 result the following statement.

Corollary 4.3 *The graph G_ℓ is a tree.*

Let P be a $\check{v}$-$\hat{v}$ path in $\tilde{\mathbb{M}}$. Since $\tilde{\mathbf{M}}$ is a basic terrain (does not have saddle vertices), by Lemma 4.1 $\tilde{\mathbb{M}}^* \setminus P^*$ is acyclic. Let $\prec$ be the adjacency partial order on $\tilde{F}$ induced by $\tilde{\mathbb{M}}^* \setminus P^*$. Since $F \subset \tilde{F}$, $\prec$ is also a partial order when restricted to F.

Lemma 4.10 *Let $\lhd$ be a linear extension of $\prec$ on F. If K and K' are two contours of a level set $\mathbb{M}_\ell$ and $f_1, f_2 \in F(K)$ and $f_1', f_2' \in F(K')$ are such that $f_1 \lhd f_1' \lhd f_2$, then $f_1 \lhd f_2' \lhd f_2$.*

Proof. Since $\tilde{\mathbf{M}}$ is a basic terrain, $\tilde{\mathbb{M}}_\ell$ consists of a single contour. Let $C^* = C(K, \mathbb{M}^*)$ be the representing cycle of $\tilde{\mathbb{M}}_\ell$ in $\tilde{\mathbb{M}}^*$. If $f_1, f_2 \in F(K)$ for some contour K in $\mathbb{M}_\ell$ and the edge common to f_1 and f_2 does not belong to either of $\hat{\mathbb{T}}$ or $\check{\mathbb{T}}$, then (f_1^*, f_2^*) is an edge in C^*. By Lemma 4.1, C^* has exactly one edge in P^*. Thus $C^* \setminus P^*$ is a path Q^* that is exactly one edge short of C^*.



(a) (b)

Figure 4.9: Contours of $\mathbb{M}_\ell$ (a) versus those of $\tilde{\mathbb{M}}_\ell$ (b).

Let G_ℓ be the tree of Corollary 4.3. We map the path Q^* in $\tilde{\mathbb{M}}_\ell$ into a walk W in G_ℓ as follows: Each vertex f^* of Q^* is mapped to a pair (u, v) where either u and v are neighboring vertices in G_ℓ or $u = v$. Specifically, if $f \in \mathbb{M}_\ell(K)$, then f^* is mapped into (v_K, v_K) where v_K is the vertex of G_ℓ that represents K. Otherwise, f^* is dual to of some $\check{v}$- or $\hat{v}$-triangle f in $\tilde{\mathbb{M}}$ that intersects $\tilde{\mathbb{M}}_\ell$. Let $e = (u, v)$ be the edge in $\mathbb{M}$ at (a copy of) which f is based in $\tilde{\mathbb{M}}$. If the edge e does not intersect $\mathbb{M}_\ell$, it is contained in $G_\ell(R)$ (as an undirected edge) for precisely one $R \in \mathcal{R}_\ell$, in which case we map f^* to (u, v). On the other hand, if e intersects $\mathbb{M}_\ell$ at a contour K, then by Lemma 4.8 there will be precisely one edge $\{v_K, v\}$ in G_ℓ where v is an endpoint of e, in which case we map f^* into (v, v_K). It can be verified that the set of pairs in the image of Q^* under this mapping is indeed a walk W in G_ℓ. Since $\tilde{\mathbb{M}}_\ell$ goes thorough any segment s at most twice, each edge of G_ℓ appears at most twice in W.

For $f_1 \lhd f_1' \lhd f_2$ to hold, Q^* must visit f_1^*, ${f_1'}^*$ and f_2^* in this order. Assume without loss of generality that $f_1' \lhd f_2'$. In order for $f_1 \lhd f_2' \lhd f_2$ not to hold, one must have $f_2 \lhd f_2'$ which means Q^* must visit ${f_2'}^*$ after f_2^*. But this corresponds to going from K to K', then back to K and then again to K'. Since each of K and K' are represented by a vertex in G_ℓ this would mean that W goes thorugh v_K, $v_{K'}$, and again v_K in this order. Corollary 4.3 implies then that W has to traverse some edge of G_ℓ at least three times, a contradiction. □

Lemmas 4.7 and 4.10 respectively prove that the total order $\lhd$ has properties (C1) and (C2) of a level-ordering .

Theorem 4.1 *For any terrain* $\mathbf{M}$ *with triangulation* $\mathbb{M}$*, there is exists a level level-ordering of the triangles of* $\mathbb{M}$*.*

4.4 Contour Algorithms

In this section we describe I/O-efficient algorithms for computing contour maps as well as an I/O-efficient data structure for answering contour queries.

4.4.1 Level-ordering of terrain triangles

We describe an I/O-efficient algorithm for computing, given a terrain $\mathbf{M}$, the triangulation $\tilde{\mathbb{M}}$ of the simplified terrain $\tilde{\mathbf{M}}$, and a monotone path P from $\check{v}$ to $\hat{v}$ in $\tilde{\mathbb{M}}$. We can then compute a level-ordering of the triangles of $\tilde{\mathbb{M}}$ in $O(\mathrm{Sort}(N))$ I/Os using an existing I/O-efficient topological sorting algorithm for planar DAGs [22]. This induces a level-ordering on the triangles of $\mathbb{M}$.

Computing the red cut-tree The first step in computing $\tilde{\mathbb{M}}$ is to compute a red (ascending) cut-tree $\hat{\mathbb{T}}$ of $\mathbb{M}$. The I/O-efficient topological persistence algorithm of Agarwal et al. [5] can determine the type (regular, minimum, negative saddle, ...) of every vertex of $\mathbb{M}$ in $O(\mathrm{Sort}(N))$ I/Os. Moreover, for every vertex $v \in \mathbb{M}$, it can also compute, within the same I/O bound, a vertex from each connected component of $\mathrm{Lk}^+(v)$. Since each saddle of $\mathbb{M}$ is assumed to be simple, $\mathrm{Lk}^+(v)$ has at most two connected components.

To compute $\hat{\mathbb{T}}$, we apply the *time-forward processing* technique [41] using a priority queue Q: we scan the vertices of $\mathbb{M}$ in the increasing order of their heights. We store a subset of vertices in Q, namely the upper endpoints of the edges of $\hat{\mathbb{T}}$ whose lower endpoints have been scanned. The priority of a vertex v in Q is its height $h(v)$. Suppose we are scanning a vertex v of $\mathbb{M}$ and u is the lowest priority vertex in Q. If $h(v) < h(u)$ and v is not a positive saddle, we move to a new vertex in $\mathbb{M}$. Otherwise, i.e. if $h(u) = h(v)$ or v is a positive saddle, we choose a vertex w from each connected component of $\mathrm{Lk}^+(v)$, which we have already computed in the preprocessing step. We add the edge (v, w) to $\hat{\mathbb{T}}$ and add w to Q. Since each operation on Q can be performed in $O\left(\frac{1}{B}\log_{M/B} N/B\right)$ I/Os, $\hat{\mathbb{T}}$ can be computed in $O(\mathrm{Sort}(N))$ I/Os.

Adding the blue cut-tree The second step in computing $\tilde{\mathbb{M}}$ is to compute a blue cut-tree $\check{\mathbb{T}}$ of $\mathbb{M}_0$. However, we can compute $\check{\mathbb{T}}$ directly on $\mathbb{M}$ if we ensure that $\hat{\mathbb{T}}$ and $\check{\mathbb{T}}$ do not cross each other, even though they can share edges. This property can be ensured by choosing the ascending and descending edges, in $\hat{\mathbb{T}}$ and $\check{\mathbb{T}}$, respectively, out of each vertex v, more carefully. Specifically, we use the following rule:

1. On an ascending path, the edge following (u, v) is (v, w) where (v, w) is the first outgoing edge out of v after (u, v) in clockwise order, and
2. On a descending path, the edge following (v, u) is (w, v) where (w, v) is the first incoming edge of v after (v, u) in counterclockwise order.

It can be verified that $\hat{\mathbb{T}}$ and $\check{\mathbb{T}}$ do not cross. One can therefore compute $\check{\mathbb{T}}$ precisely in the same way as $\hat{\mathbb{T}}$ directly on $\mathbb{M}$.

Computing a monotone $\check{v}$-$\hat{v}$ path P While computing $\check{\mathbb{T}}$ we also compute a descending path starting at the lowest *positive* saddle v_1 of $\mathbb{M}$ as though v_1 were another negative saddle. This path P, which ends at a $\check{\mathbb{T}}$ vertex v_0, together with $(\check{v}, v_0)$ and $(v_1, \hat{v})$ serves as a monotone path in $\tilde{\mathbb{M}}$ connecting $\check{v}$ to $\hat{v}$.

Generating $\tilde{\mathbb{M}}^* \setminus P^*$ The topological sorting algorithm of Arge et al. [22] takes as input a planar directed acyclic graph, represented as a list of vertices along with the list of edges incident upon each vertex in circular order. Given $\mathbb{M}$, $\hat{\mathbb{T}}$, $\check{\mathbb{T}}$, and P, we need to compute such a representation of $\tilde{\mathbb{M}}^* \setminus P^*$. Since each face in $\tilde{\mathbb{M}}$ is a triangle, $\tilde{\mathbb{M}}^*$ is 3-regular. It is easy to compute the circular order of edges incident upon a vertex of $\tilde{\mathbf{M}}^*$ whose dual triangle is neither a $\hat{v}$- or $\check{v}$-triangle, nor adjacent to a copy of a $\hat{\mathbb{T}}$ or $\check{\mathbb{T}}$ edge. The main task is then to compute these the $\hat{v}$- and $\check{v}$-triangles. This can be accomplished by computing the Eulerian tours of $\hat{\mathbb{T}}$ and $\check{\mathbb{T}}$, which takes $O(\text{Sort}(N))$ I/Os [22]. Putting everything together, we obtain the main result of this chapter.

Theorem 4.2 *Given a terrain* $\mathbf{M}$ *with triangulation* $\mathbb{M}$*, a level-ordering of the triangles of* $\mathbb{M}$ *can be computed in* $O(\text{Sort}(N))$ *I/Os, where* N *is the number of vertices of* $\mathbb{M}$*.*

4.4.2 Contour maps of basic terrains

Let $L = \{\ell_1, \ldots, \ell_s\}$ be a set of input levels with $\ell_1 < \cdots < \ell_s$. Given a basic terrain $\mathbb{M}$, the goal is to compute the contour map of $\mathbb{M}$ for levels in L. Since $\mathbb{M}$ is simple, each $\mathbb{M}_{\ell_i}$ consists of a single contour. Generating the segments of $\mathbb{M}_{\ell_i}$ in clockwise or counterclockwise order is equivalent to listing the triangles of $\mathbb{M}$ that the contour $\mathbb{M}_{\ell_i}$ intersects in that order, i.e. reporting $C(\mathbb{M}_\ell, \mathbb{M})$.

Our algorithm uses a *buffer tree* $\mathcal{B}$ to store the triangles of $\mathbb{M}$ that intersect a level set. The buffer tree [11] is a variant of B-tree, which propagates updates from the root to the leaves in a lazy manner, using buffers attached to the internal nodes of the tree. As a result, a sequence of N updates (inserts and deletes) can be performed in amortized $O(\text{Sort}(N))$ I/Os. Moreover, one can perform a *flush* operation on a buffer tree that results in the writing of all its stored elements on the disk in sorted order. Flushing a tree with T elements takes $O(T/B)$ I/Os.

It is more intuitive to describe the algorithm as a plane sweep of $\mathbf{M}$ in $\mathbb{R}^3$. At the first step, the algorithm computes a level-ordering $\lhd$ of the terrain triangles using Corollary 4.1. Then starting at $\ell = -\infty$, the algorithm sweeps a horizontal plane z_ℓ in the positive z-direction. A *target level* ℓ_* is initially set to ℓ_1. At any time the algorithm maintains the list of triangles of $\mathbf{M}$ that intersect the sweeping plane in a buffer tree $\mathcal{B}$ ordered by $\lhd$. Whenever the sweep plane encounters the bottom-most vertex of a triangle f, we insert f into $\mathcal{B}$; f is deleted again from $\mathcal{B}$ when the plane reaches the top-most vertex of f. When the sweep plane z_ℓ reaches the target level ℓ_* (or rather the lowest vertex of height ℓ_* or more when the sweeping is implemented discretely), we flush the buffer tree. The generated list of triangles (vertices of $\mathbb{M}^*$) are precisely the set of triangles in $\mathbb{M}$ that intersect z_{ℓ_*} ordered by $\lhd$. Corollary 4.1 implies that the output is exactly $C(\mathbb{M}_{\ell_*}, \mathbb{M})$. The algorithm then raises the target level ℓ_* to the next level in L and continues.

Level-ordering of the terrain triangles takes $O(\text{Sort}(N))$ I/Os (Theorem 4.2). Preprocessing for the sweep algorithm consists of sorting the vertices in their increasing order of heights which can also be done in $O(\text{Sort}(N))$ I/Os. During

the sweep each update on the buffer tree takes $O(\frac{1}{B}\log_{M/B} N/B)$ amortized I/Os [11]. Thus all the $O(N)$ updates can be performed in $O(\text{Sort}(N))$ I/Os in total. Each flushing operation takes $O(T'/B)$ I/Os, where T' is the number of triangles in $\mathcal{B}$. If a triangle is in $\mathcal{B}$ but has been deleted, it is not in $\mathcal{B}$ after the flushing operation, so a "spurious" triangle is flushed only once.

Hence, the total number of I/Os is $O(\text{Sort}(N)+T/B)$, where T is the output size. Finally, in addition to storage used for the terrain the algorithm uses $O(N/B)$ blocks to store the buffer tree and thus uses $O(N/B)$ blocks in total.

4.4.3 Generalization to arbitrary terrains

Given a general terrain $\mathbf{M}$ with saddles, one can still compute by Theorem 4.2 a level-ordering of the triangles of $\mathbb{M}$ in $O(\text{Sort}(N))$ I/Os. If one runs the algorithm of the previous section on $\tilde{\mathbb{M}}$ the output generated for each input level ℓ_i is $C(\tilde{\mathbb{M}}_{\ell_i}, \tilde{\mathbb{M}})$. Running the algorithm on $\mathbb{M}$ is equivalent to running it on $\tilde{\mathbb{M}}$ but ignoring all $\hat{v}$ and $\check{v}$-triangles by omitting their insertions into the buffer tree. Consequently, the produced output for level ℓ_i is the same sequence of triangles only with $\hat{v}$ and $\check{v}$-triangles omitted. By Theorem 4.1 this is a subsequence $R = \langle f_1, \ldots, f_k\rangle$ of $C(\tilde{\mathbb{M}}_{\ell_i}, \tilde{\mathbb{M}}))$ in which $C(K, \mathbb{M})$ of each contour K in $\mathbb{M}_{\ell_i}$ appears as a subsequence R_K. Thus all one needs to do is to extract the subsequence R_K and write it separately in the same order as it appears in R. Property (C2) of a level-ordering allows this to be done in $O(k/B)$ I/Os: if in scanning the sequence R from left to right some elements of R_K are later followed by elements of $R_{K'}$, then the appearance of another element of R_K, indicates that no more elements from $R_{K'}$ remain.

We thus scan the sequence R from left to right and *push* the scanned triangles into a stack S_F. Every time the last element of a contour is pushed into the stack, the triangles of that contour make a suffix of the list of elements stored in S_F. At such a point, we *pop* all the elements corresponding to the completed contour and write them to the disk. To find out when a contour is completed and how many elements on the top of stack belong to it, we keep a second stack S_E of edges. For any triangle $f \in F(\mathbb{M}_{\ell_i})$, two of the edges of f intersect $\mathbb{M}_{\ell_i}$. With respect to the orientation of these edges, f is to the right of one of them and to the left of the other one which we respectively call the *left* and *right* edges of f at level ℓ_i. If $e^* = (f_j^*, f_{j+1}^*)$ is an edge of the representing cycle of a contour in $\mathbb{M}_{\ell_i}$, then e is the right edge of f_j and left edge f_{j+1} at level ℓ_i. We therefore check when scanning a triangle f_j whether its left edge is the same as the right edge of the triangle on top of S_F and insert f_j into S_F if this is the case. Otherwise, we compare the left edge of f_j with the edge on top of S_E. If they are not the same, we are visiting a new contour and we insert the left edge of f_j into S_E and f_j into S_F. Otherwise, f_j is the last triangle of its contour. Therefore we write it to the disk and successively pop and write to disk enough triangles from S_F until the left edge of a popped triangle matches the right edge of f_j. We also pop this edge from S_E. In this algorithm each scanned triangle is pushed to the stack once and popped once. In a standard I/O-efficient stack implementation this costs $O(k/B)$ I/Os.

Theorem 4.3 *Given any terrain* $\mathbf{M}$ *with* N *vertices and a list* $L = \{\ell_1, \ldots, \ell_s\}$ *of levels with* $\ell_1 < \cdots < \ell_s$*, one can compute using* $O(\text{Sort}(N) + T/B)$ *I/Os the contour map of* $\mathbf{M}$ *for levels in* L*, where* T *is the total number of produced segments.*

4.4.4 Extracting nesting of contours

In addition to reporting each contour individually, a number of applications call for computing how various contours are nested within each other. We produce this information by returning the parent of each contour in a computed contour map.

The parent-child relationship between individual contours in a contour map of a terrain $\mathbf{M}$ can be read from the contour tree of $\mathbf{M}$. Each contour in a contour map corresponds to a point on some edge of the contour tree. Two contours K and K' in the map are neighbors (either siblings or parent and child) if and only if their corresponding points on the contour tree can be connected by a path that does not pass through a point corresponding to a third contour $K'' \neq K, K'$ in the given contour map. Each edge of the contour tree can be colored red or blue according to the color of the contours it represents (all contours represented by the points on the same edge of the contour tree have the same color). By the assumption that all saddles are simple, internal nodes of the contour tree which correspond to saddles all have degree three. It can be verified that at a joining (negative) saddle only two color combinations on the edges incident to the saddles are possible. The same holds for a splitting (positive) saddle. In each case, using the edges colors, and using the corresponding patterns in which contours of various colors can merge or split, one can uniquely determine one of the edges incident to the saddle that carries contours that are parents to those carried by the other two. We orient this edge at each saddle away from and the other two toward the saddle. The resulting orientation on the contour-tree is equivalent to orienting each blue edge toward its higher end and each red edge toward its lower end. With this orientation of the contour tree a contour K will be the parent of a contour K', if the path between points representing K and K' on the contour tree follows the orientation of the edges of the tree. The algorithm of Arge et al. [5] for computing the contour tree can return the color of each edge on the generated tree which can be used to orient the tree as explained above.

By determining the edge of the contour tree carrying each contour of the contour map, one can determine the parent-child relationship between individual contours in $O(\text{Sort}(N) + T/B)$ I/Os, where T is the number of contours in the given contour map, through a pre-order traversal of the oriented contour tree.

To facilitate finding of the edge the contour tree that carries a contour in the given map, instead of the contour tree, we compute the *augmented contour-tree* [5]. The augmented contour tree of a terrain replaces each edge of the contour tree with a monotonically ascending path whose vertices are the vertices of the terrain. Every regular vertex of the terrain appears precisely once in one of these paths. All the properties of contour tree are also valid for the augmented contour tree. We store in each vertex u of the terrain a pointer to

each of the (at most two if u is a saddle and one if u is regular) edges in the augmented contour tree that have u as the lower endpoint. To locate the edge in the augmented contour tree corresponding to a computed contour K, we scan the list of triangles that intersect K and determine the vertices u and u' of these triangles respectively highest below and lowest above the level of K. Since there are no vertices between u and u', shifting K down or up between $h(u)$ and $h(u')$ does not change the set of triangles it intersects and therefore its homology class. Consequently, uu' is an edge of the augmented contour tree. All that is needed then is to locate the pointer to the contour tree edge stored at u that matches uu'. Since the augmented contour tree of a terrain of size N can be computed in $O(\text{Sort}(N))$ I/Os [5], we summarize the above discussion as follows:

Theorem 4.4 *For a contour map consisting of T contours of a terrain with N triangles, where each contour is given as a list of triangles that intersect it together with its level, one can report for each contour in the map a pointer to its parent in $O(\text{Sort}(N) + T/B)$ I/Os.*

4.4.5 Answering contour queries

The sweep algorithm described in the previous section can easily be modified to construct a linear space data structure that given a query level ℓ can report the contours in the level set $\mathbb{M}_\ell$ I/O-efficiently. Unlike the previously known structure for this problem [3], our structure can be constructed in $O(\text{Sort}(N))$ I/Os. To obtain the structure we simply replace the buffer tree $\mathcal{B}$ with a partially persistent B-tree [18, 116]. To build the structure, we sweep $\mathbf{M}$ by a horizontal plane in the same way as we did in the algorithm of Section 4.4.2, inserting the triangles when the sweep plane reaches their bottom-most vertex, without checking for them to intersect any target levels, and deleting them when the sweep plane passes their top-most vertex. There will also be no need to flush the tree.

Since $O(N)$ updates can be performed on a persistent B-tree in $\text{Sort}(N)$ I/Os [19, 112], the sweeping of the terrain require $O(\text{Sort}(N))$ I/Os. A persistent B-tree allows us to query any previous version of the structure and in particular produce the list of the elements stored in the tree in $O(\log_B N + T/B)$ I/Os when T is the number of reported elements. Therefore we can obtain $\mathbb{M}_\ell$ in the same bound, simply by querying the structure for the triangles it contained when the sweep-plane was at height ℓ and then utilize Theorem 4.1 and the contour extraction algorithm discussed above to extract individual contours of $\mathbb{M}_\ell$.

Theorem 4.5 *Given a terrain $\mathbf{M}$ with N vertices, one can construct in $O(\text{Sort}(N))$ I/Os a linear size data structure, such that given a query level ℓ, one can report contours of $\mathbb{M}_\ell$ in $O(\log_B(N) + T/B))$ I/Os where T is the size of the query output. Each contour is reported individually, and the edges of each contour are sorted in clockwise order.*

4.5 Conclusions

We defined level-ordering of terrain triangles and proved that every terrain has a level ordering that can be computed I/O-efficiently. Based on this, we provided algorithms that compute contours of a given terrain within similar I/O bounds. An immediate question is whether this approach can be generalized to triangulated surfaces of arbitrary genus with arbitrary piecewise linear functions defined on them. Notice that the problem is valid even if the input surface is not embedded in $\mathbb{R}^3$. Another interesting open problem for surfaces that are embedded in $\mathbb{R}^3$ is computing of level sets or answering contour queries for a height function defined by a variable z direction: is it possible to preprocess a given triangulated surface embedded in the three space so that for any given direction, the contours of the height function for that direction can be computed I/O-efficiently?

Chapter 5

Cache-Oblivious Red-Blue Line Segment Intersection

In this chapter, we develop a cache-oblivious algorithm for the *red-blue line segment intersection problem*, that is, for finding all intersections between a set of non-intersecting red segments and a set of non-intersecting blue segments in the plane. The algorithm uses $O(\text{Sort}(N) + T/B)$ memory transfers where N is the total number of segments and T is the number of intersections. Our algorithm is optimal and, to the best of our knowledge, the first efficient cache-oblivious algorithm for any intersection problem involving non-axis-parallel object.

In the first paper to consider computational geometry problems in external memory [66], Goodrich et al. introduced the *distribution sweeping* technique (a combination of M/B-way distribution sort and plane sweeping) and showed how it can be used to solve a large number of geometric problems in the plane using $O(\text{Sort}(N) + T/B)$ memory transfers, where T is the output size of the problem (e.g., number of intersections). The problems they considered include the orthogonal line segment intersection problem and other problems involving axis-parallel objects. Arge et al. developed an algorithm that solves the red-blue line segment intersection problem using $O(\text{Sort}(N)+T/B)$ memory transfers [23], which is optimal. The algorithm uses the distribution sweeping technique [66] and introduces the notion of *multi-slabs*; if the plane is divided into vertical slabs, a multi-slab is defined as the union of any number of consecutive slabs. Multi-slabs are used to efficiently deal with segments spanning a range of consecutive slabs. The key is that, if there are only $\sqrt{M/B}$ slabs, there are less than M/B multi-slabs, which allows the distribution of segments into multi-slabs during a plane sweep using standard M/B-way distribution. Arge et al. also extended their algorithm to obtain a solution to the general line segment intersection problem using $O(\text{Sort}(N + T))$ memory transfers [23].

Bender et al. [30] developed a cache-oblivious algorithm that solves the offline planar point location problem using $O(\text{Sort}(N))$ memory transfers; Brodal and Fagerberg [34] developed a cache-oblivious version of distribution sweeping and showed how to use it to solve the orthogonal line segment intersection problem, as well as several other problems involving axis-parallel objects, cache-obliviously using $O(\text{Sort}(N) + T/B)$ memory transfers.

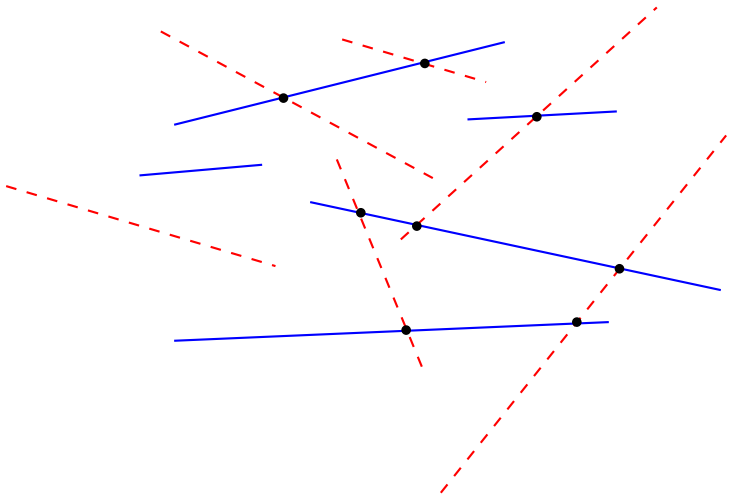


Figure 5.1: An instance of the red-blue line segment intersection problem.

5.1 Our Contribution

As discussed, the external-memory algorithm for this problem [23] is based on an extended version of distribution sweeping utilizing multi-slabs. Our new algorithm borrows ideas from both the external-memory algorithm for the red-blue line segment intersection problem [23] and the cache-oblivious algorithm for the orthogonal line-segment intersection problem [34]. In order to obtain a useful notion of sweeping the plane top-down or bottom-up, we utilize the same total ordering as in [23] on a set of non-intersecting segments, which arranges the segments intersected by any vertical line in the same order as the y-coordinates of their intersections with the line. In the case of axis-parallel objects, such an ordering is equivalent to the y-ordering of the vertices of the objects; in the non-axis-parallel case, this ordering is more difficult to obtain [23]. Similar to the cache-oblivious orthogonal line-segment intersection algorithm [34], we employ the cache-oblivious distribution sweeping paradigm, which uses two-way merging rather than $\sqrt{M/B}$-way distribution. While this eliminates the need for multi-slabs, which do not seem to have an efficient cache-oblivious counterpart, it also results in a recursion depth of $\Theta(\log_2 N)$ rather than $\Theta(\log_{M/B} N)$. This implies that one cannot afford to spend even $1/B$ memory transfers per line segment at each level of the recursion. For axis-parallel objects, Brodal and Fagerberg [34] addressed this problem using the so-called k-merger technique, which was introduced as the central idea in Funnel Sort (ie., cache-oblivious Merge Sort) [64]. This technique allows N elements to be passed through a $\log_2 N$-level merge process using only $O(\mathrm{Sort}(N))$ memory transfers, but generates the output of each merge process in bursts, each of which has to be consumed by the next merge process before the next burst is produced. This creates a new challenge, as a segment may have intersections with all segments in the output stream of a given merge process and, thus, needs access to the entire output stream to report these intersections. To overcome this problem, Brodal and Fagerberg [34] provided a technique to detect, count, and collect

intersected segments at each level of recursion that ensures that the number of additional accesses needed to report intersections is proportional to the output size.

Our main contribution is the development of non-trivial new methods to extend the counting technique of Brodal and Fagerberg [34] to the case of non-axis-parallel line segments. These ideas include a *look-ahead* method for identifying certain critical segments ahead of the time they are accessed during a merge, as well as an *approximate counting* method needed because exact counting of intersected segments (as utilized in the case of axis-parallel objects) seems to be no easier than actually reporting intersections.

5.2 Topological Sorting of Planar *st*-Graphs

In this section we present a cache-oblivious algorithm for constructing a topological ordering of the vertices of a planar *st*-graph. The algorithm is based on the PRAM-algorithm from [108], which we modify to obtain a cache-oblivious algorithm.

5.2.1 Properties

We will use the following properties of a planar *st*-graph G [27, 77, 81, 107].

(i) G can be drawn in the plane such that all edges are directed upwards.

(ii) For every vertex v of G, there is a path from s to t through v.

(iii) In a clockwise ordering of the edges incident to each vertex v, the cycle can be broken so that all incoming edges precede all outgoing edges of v.

(iv) The boundary of each face consists of two directed paths with common origin and endpoint.

We will now define four functions $(\text{left}, \text{right}, \text{low}, \text{high})$ on the elements of G [108]. For every edge $e = (u, v)$, we define $\text{low}(e) = u$ and $\text{high}(e) = v$; $\text{left}(e)$ is the face to the left of e, and $\text{right}(e)$ is the face to the right of e. For a vertex, we refer to the Property (iii) above and define $\text{left}(v)$ ($\text{right}(v)$) to be the faces between the leftmost (rightmost) incoming edge and the leftmost (rightmost) outgoing edge of v; $\text{low}(v) = \text{high}(v) = v$. Finally, for a face, we refer to Property (iv) above and define $\text{low}(f)$ and $\text{high}(f)$ to be the common origin and endpoint, respectively, of the two bounding paths of v; $\text{left}(f) = \text{right}(f) = f$. Some of these definitions are shown in Figure 5.2.

For a planar *st*-graph G, its *dual* G^* is defined as follows: G^* contains one vertex f^* for each face f of G, except the external face; the external face is represented by two vertices s^* and t^*. For every edge e of G, G^* contains a directed edge $(\text{left}(e)^*, \text{right}(e)^*)$; if $\text{left}(e)$ is the external face, $\text{left}(e)^*$ is chosen to be s^*; if $\text{right}(e)$ is the external face, $\text{right}(e)^* = t^*$.

We now define two partial orders on the vertices, edges, and faces of G: We say that x is below y and write $x \uparrow y$ if there is a path from $\text{high}(x)$ to $\text{low}(y)$

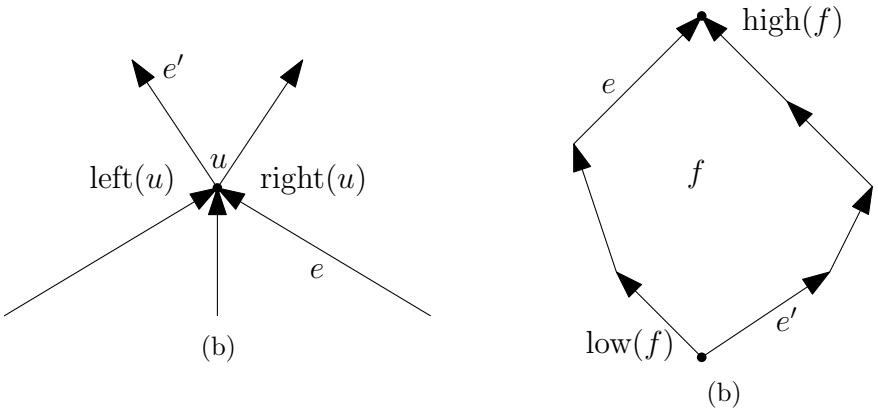


Figure 5.2: (a) A vertex v and the faces left(v) and right(v). The vertex is represented by the pairs (e, u) and (u, e') in L. (b) A face f and high(f) and low(f). The face is represented by the pairs (e, f) and (f, e') in L.

in G. Similarly, we say that x is to the left of y and write $x \rightarrow y$ if there is a path from right$(x)^*$ to left$(x)^*$ in G^*.

Observe [106] that a vertex u has to precede vertex v in the topological ordering of G if $u \uparrow v$. Thus, we aim to extend the partial order $\uparrow$ to a total order $<_t$. Note that $\uparrow$ itself is not a total order unless G is a path. The desired total ordering is given by the following definition.

Definition 5.1 ([106]) *Let x, y be entities of the planar st-graph G, where an entity is a vertex, edge or face. We define $x <_t y$ to hold if and only if $x \rightarrow y$ or $x \uparrow y$.*

Tamassia and Preparata [106] prove that $<_t$ is a total order on the faces, edges and vertices of G. Having defined the order $<_t$, we now want to obtain an algorithm that actually sorts the vertices of G using this order.

5.2.2 Computing the topological order

To sort the vertices of a planar *st*-graph $G = (V, U)$, we proceed as in the PRAM algorithm of [108]. We will construct a linked list L of all elements of $V \cup E \cup F$, where F denotes the sef of faces of G. There will be a link (x, y) in the list if and only if x is y's immediate predecessor with respect to $<_t$. Thus, ordering the vertices, edges, and faces of G by $<_t$ reduces to computing the *rank* of every element v in L, that is, the number of links on the path from the head of L to v. The latter problem can be solved using an efficient cache-oblivious algorithm [12].

To construct list L, recall that G is represented as a collection of adjacency lists of the vertices of G; each adjacency list is sorted clockwise around each vertex, which incoming edges preceding the outgoing edges. To define list L, we consider every vertex and every face of G. For a vertex, we choose its predecessor in L from among its in-edges and its successor from among its out edges. For a face, the predecessor is an edge on its left bounding path, and the successor is an edge on its right bounding path. The details are as follows:

For each vertex $v \neq s$, its predecessor is its rightmost incoming edge. For each vertex $v \neq t$, its successor is its leftmost outgoing edge. If edge e is chosen as the predecessor of v, we write a link (e, v) to L; similarly, we represent the choice of a successor e' of v by writing a link (v, e') to L.

For each face f other than the external face, we define its predecessor to be the topmost edge on its left bounding path and its successor to be the bottommost edge on its right bounding path, and we represent these choices by adding appropriate links to L. This can be done by scanning the list of edges after sorting them by the IDs of their left faces, followed by another scan of the edge list after sorting them by the IDs of their right faces.

The following lemma now shows that list ranking applied to L produces the desired topological ordering $<_t$.

Lemma 5.1 ([108]) *Ranking the list L produced by the above procedure yields the order $<_t$ on G.*

The only missing detail of the algorithm is a way of assigning IDs to the faces of G and labelling every edge with the IDs of its two incident faces. This is what we focus on next.

Finding faces

We now show how to mark each edge e using IDs left(e) and right(e) uniquely identifying the faces to their left and right. We talk about how to compute right(e). The computation of left(e) is analogous.

We will construct linked lists representing the left bounding paths of the faces of G. Using the list-ranking algorithm of [12], we then label every edge e in such a path with the ID of its topmost edge. The edges in the right boundary path is labelled by the ID of the clockwise successor edge of the topmost edge of the path. Thus the unique ID of each face is the topmost edge on the left boundary path. The linked lists are represented by a soup S of links connecting edges on these paths to their successors. These links are easy to generate from the adjacency lists of the vertices of G because, for an edge $e = (u, v)$ on the left boundary path of a face f and its successor $e' = (v, w)$ on this path, e must be the rightmost in-edge of v, and e' must be its rightmost out-edge. Given the order in which the edges in all adjacency lists are arranged, we can thus create the links to be added to S by a simple scan of all adjacency lists.

We now have the following results, analogous to the PRAM version [108].

Theorem 5.1 *A total order of the vertices, edges, and faces of an st-graph G with N vertices consistent with a topological ordering of the vertices in G can be constructed cache-obliviously in $O(\mathrm{Sort}(V))$ memory transfers.*

Proof. Since G is planar, the number of edges and faces is $O(N)$. The construction of the order and the faces is done by a constant number of scans, sorts and list rankings all of which can be done in can be done in $O(\mathrm{Sort}(N))$ memory transfers. [7, 12]. □

Corollary 5.1 *An topological order of the N vertices in an st-graph G can be constructed cache-obliviously in $O(\mathrm{Sort}(N))$ memory transfers.*

5.3 Vertically Sorting Non-Intersecting Segments

Let S be a set of N non-intersecting line segments in the plane, and let s_1 and s_2 be two segments in S. We say that s_2 is *above* s_1, denoted $s_1 <_A s_2$, if and only if there exists a vertical line intersecting s_1 in the point (x, y_1) and s_2 in the point (x, y_2) such that $y_1 < y_2$. Note that not all segments in S are comparable under $<_A$. The problem of sorting S consists of extending the partial order $<_A$ to a total order, that is, to construct a total order $<_t$ on S such that $s_1 <_A s_2$ implies that $s_1 <_t s_2$ [23]. Below we sketch an $O(\mathrm{Sort}(N))$ cache-oblivious algorithm for this problem. We will utilize this algorithm in our cache-oblivious red-blue line segment intersection algorithm presented in Section 5.4.

Our cache-oblivious algorithm for sorting S is an adaption of the corresponding external-memory algorithm, by Arge et al. [23], who proved that to establish the aboveness relation, it suffices to consider vertical lines through segment endpoints: If we define $s_1 \nearrow s_2$ if and only if s_2 is the segment immediately above one of the endpoints of s_1, or s_1 is the segment immediately below one of the endpoints of s_2, then the problem of sorting S is equivalent to extending the relation $\nearrow$ to a total order. To obtain an external-memory algorithm, they then considered the graph G_S containing a vertex v_{s_1} corresponding to each segment s_1 in S, as well as two special vertices v_s and v_t representing horizontal segments s and t of infinite length below, respectively above, all segments in S; there is a directed edge from v_{s_1} to v_{s_2} if $s_1 \nearrow s_2$. They proved that G_S is a *planar st-graph* [108] and that a topological ordering of G_S provides the desired total order on S. More precisely, if $s_1 <_A s_2$ then $v_{s_1} <_t v_{s_2}$, where $<_t$ is a topological ordering of the vertices in G_S. We call $<_t$ a *vertical ordering* of the segments. See Figure 5.3 for an illustration of how this is used for the kind of problem instance considered in the next sections.

Thus, to sort S, it suffices to construct G_S and compute a topological ordering of its vertices which can be done using Corollary 5.1.

Constructing G_S To construct G_S we first use the cache-oblivious batched point location algorithm by Bender et at. [30] to find the four (not nescessarily distinct) segments above and below the two endpoints of each segment s. This is done by running the algorithm twice using the segment endpoints as queries. The first run of the algorihm finds the segments above the query endpoints and by negating all y coordinates in the second run we similarly find the segments below the query points. We use the result of the point location to produce a sequence of triplets (s, q, s') where $s <_A s'$ and where q is an endpoint of s or s' defining the relation. Since the same segment pair may be present in sequence twice in the case where s is dominated by s' at two endpoints, we remove "duplicate" entries (s, q, s') and (s, q', s') from the lists by sorting. Subsequently we sort the sequence by the first segment id and break ties using x coordinate of q. We can now scan the sorted sequence and produce outgoing edges of s in

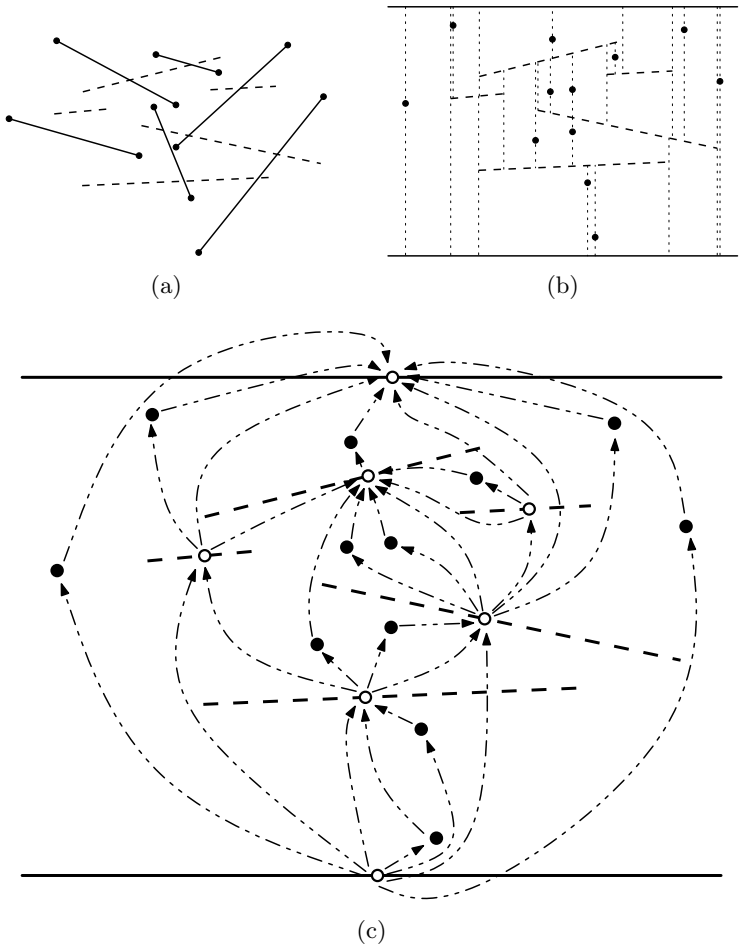


Figure 5.3: Constructing G_S. (a) Input instance containing both red (dashed) and blue (solid) segments. (b) In the set S, each blue segment is represented by two infinitely small segments at each of its endpoints and two point location queries at each point have been issued. (c) The planar *st*-graph G_S.

G_s in their clockwise order around s. We can construct the incoming edges of each segment node in G_S by sorting by the second segment in the sequence.

The cache-oblivious batched point location algorithm by Bender et at. [30] can perform $O(N)$ queries on a set with $O(N)$ segments using $O(\text{Sort}(N))$ I/Os. Since the length of produced sequence is $O(N)$ the subsequent scanning and sorting steps can be done in $O(\text{Sort}(N))$ which is also the complexity of the planar *st*-graph topological sorting routine from Corollary 5.1.

Theorem 5.2 *A vertical ordering of N non-intersecting line segments in the plane can be computed cache-obliviously using $O(\text{Sort}(N))$ memory transfers and linear space.*

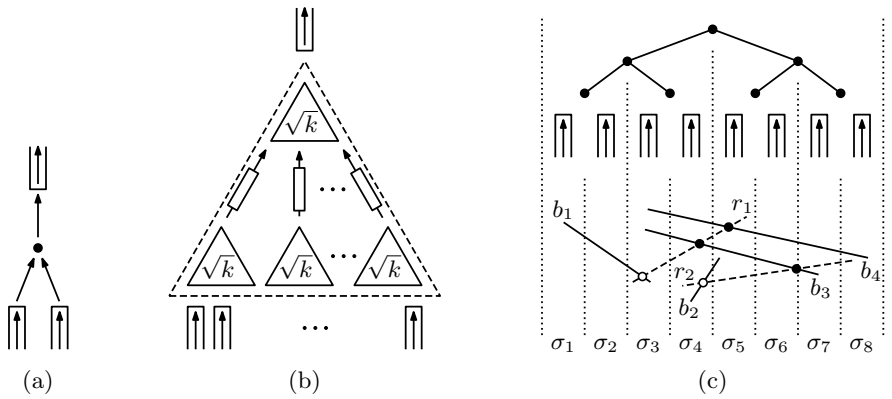


Figure 5.4: (a) A 2-merger. (b) A k-merger for $k > 2$. (c) Slabs and intersection types.

5.4 Red-Blue Line Segment Intersection

In this section, we give an overview of our algorithm for finding all intersections between a set R of non-intersecting red segments and a set B of non-intersecting blue segments. For simplicity we assume that the x- and y-coordinates of all endpoints are distinct. Sections 5.5 and 5.6 present the details of our algorithm.

5.4.1 The $\sqrt{N}$-merger

Our algorithm uses the $\sqrt{N}$-merger technique [34,64] extensively. A $\sqrt{N}$-merger merges $\sqrt{N}$ sorted input streams of length $\sqrt{N}$ into one sorted output stream. It is defined recursively in terms of smaller k-mergers. A k-merger takes k sorted input streams of total length at least k^2 and produces a sorted output stream by merging the input streams. The cost of merging k^2 elements using a k-merger is $O(\mathrm{Sort}(k^2))$, which is $O(\mathrm{Sort}(N))$ for $k = \sqrt{N}$ [34, 64].

A k-merger is a complete binary tree over $k/2$ leaves with a buffer associated with each edge. If $k = 2$, the merger consists of a single node with two input streams and one output stream; see Figure 5.4(a). Otherwise, it consists of $\sqrt{k} + 1$ $\sqrt{k}$-mergers as shown in Figure 5.4(b); the buffers associated with the edges between the top merger and the bottom mergers have size k. The merge process is performed by invoking a FILL operation on the root of the merger. A FILL operation on a node u fills the output buffer $S(u)$ of u (the buffer between u and its parent) by repeatedly removing the minimum element from $S(l(u))$ or $S(r(u))$ and placing it into $S(u)$, where $l(u)$ and $r(u)$ denote the left and right children of u. When $S(l(u))$ or $S(r(u))$ becomes empty, a FILL operation is invoked recursively on the corresponding child before continuing to fill $S(u)$. The FILL operation returns when $S(u)$ is full or there are no elements left in any buffer below u. Since the root's output buffer has size N, only one FILL operation on the root is required to place all elements in the input streams into a sorted output stream.

The basic concept in the analysis of a $\sqrt{N}$-merger is that of a *base tree*, which is the largest subtree in the recursive definition of a $\sqrt{N}$-merger such

that the entire tree plus one block for each of its input and output buffers fit in memory. The central observation is that, in order to achieve the $O(\text{Sort}(k^2))$ merge bound, a FILL operation on a base tree root can afford to load the whole base tree into memory and perform $O(1)$ memory transfers per node in the base tree; note that this means that FILL operations on other nodes of the base tree are free. It also means that we can associate $O(1)$ auxiliary buffers with each merger node u and that we can assume that a FILL operation at node u can access the first $O(1)$ blocks of each auxiliary buffer without any memory transfers. See [34] for details.

5.4.2 Distribution sweeping

To find all intersections between red and blue segments, we start by dividing the plane into $q = \sqrt{N}$ vertical slabs $\sigma_1, \ldots, \sigma_q$ containing $2\sqrt{N}$ segment endpoints each, where $N = |R| + |B|$ is the total number of segments. We recurse on each slab σ_i to find the intersections in σ_i between segments with at least one endpoint in this slab; these intersections are shown using white dots in Figure 5.4(c). Each of the remaining intersections, shown as black dots in Figure 5.4(c), involves at least one segment that completely spans the slab containing the intersection. To find these intersections, we use a $\sqrt{N}$-merger whose input streams are sorted lists of segments and/or segment endpoints associated with slabs $\sigma_1, \ldots, \sigma_q$. We also associate slabs with the nodes of the merger. The slab σ_u associated with a node u is the union of the slabs corresponding to the input streams of u's subtree. We use $l(\sigma_u)$ and $r(\sigma_u)$ to denote its left and right boundaries, respectively. We call a segment with an endpoint in σ_u *long* wrt. slab $\sigma_{l(u)}$ if it spans $\sigma_{l(u)}$ (segment b_3 in Figure 5.5(a)), and *short* otherwise (segments b_1, b_2, b_4 in Figure 5.5(a)). We call an intersection in $\sigma_{l(u)}$ *long-long* if it involves two long segments wrt. slab $\sigma_{l(u)}$ (point p_3 in Figure 5.5(a)), and *short-long* if it involves a short and a long segment (points p_1 and p_2 in Figure 5.5(a)). Short and long segments and short-long and long-long intersections in slab $\sigma_{r(u)}$ are defined analogously. It is easy to see that every intersection in a slab σ_i that involves a segment spanning σ_i is long-long or short-long at exactly one merger node. Hence, our goal in merging the streams corresponding to slabs $\sigma_1, \ldots, \sigma_q$ is to report all long-long and short-long intersections at each merger node.

Throughout this chapter, we only discuss finding, at every merger node u, short-long and long-long intersections inside $\sigma_{l(u)}$. The intersections in $\sigma_{r(u)}$ can be found analogously. Our algorithm finds short-long and long-long intersections separately and finds each intersection type using several applications of the $\sqrt{N}$-merger to appropriate input streams associated with slabs $\sigma_1, \ldots, \sigma_q$. We call one such application a *pass* through the merger. In the process of merging the input streams of the merger, each pass either reports intersections or performs some preprocessing to allow a subsequent pass to report intersections. As we show in Section 5.5 and 5.6, $O(1)$ passes are sufficient to report all short-long and long-long intersections, and each pass uses $O(\text{Sort}(N) + T_s/B)$ memory transfers and linear space, where T_s is the number of reported intersections. Let N_i denote the number of short segments in slab σ_i, T_i the number of intersections between these segments, and $C(N, T)$ the complexity of our algorithm on N segments

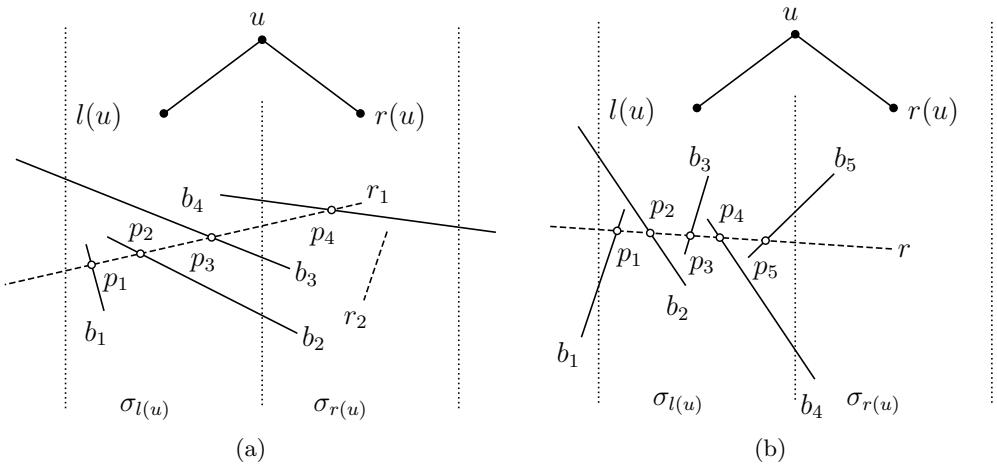


Figure 5.5: (a) Short-long and long-long intersections. (b) Upward and downward intersections.

that have T intersections. Then the complexity of our algorithm is given by the recurrence $C(N,T) = \sum_{i=1}^{\sqrt{N}} C(N_i, T_i) + O(\text{Sort}(N) + T_s/B)$, which solves to $C(N,T) = O(\text{Sort}(N) + T/B)$ because each original segment participates as a non-spanning segment in at most two slabs on each level of the recursion.

Theorem 5.3 *The red-blue line segment intersection problem can be solved cache-obliviously using $O(\text{Sort}(N) + T/B)$ memory transfers and linear space, where N is the total number of line segments and T is the number of intersections.*

5.5 Short-Long Intersections

In this section, we discuss how to find all short-long intersections at all merger nodes using $O(1)$ passes through the merger. Recall that we focus only on intersections inside $\sigma_{l(u)}$. We call such an intersection between a long red segment r and a short blue segment b *upward* if b has at least one endpoint in $\sigma_{l(u)}$ that is below r (points p_2, p_3, p_5 in Figure 5.5(b)); otherwise, the intersection is *downward* (points p_1 and p_4 in Figure 5.5(b)). We focus on finding upward short-long intersections between long red and short blue segments in the remainder of this section. The other types of short-long intersections can be found analogously. We discuss first how to find these intersections in the desired number of memory transfers using linear extra space per merger node. Then we discusses how to reduce the space bound to $O(N)$ in total.

Our algorithm uses two passes through the $\sqrt{N}$-merger. The first pass associates a *red list* $R(u)$ of size N (big enough to hold all segments in the input if necessary) with every merger node u and populates it with all red segments that are long wrt. $\sigma_{l(u)}$ and are involved in upward short-long intersections at node u. The second pass uses these red lists to report all upward short-long intersections. Both passes merge segment streams sorted by the vertical segment ordering from Section 5.3. More precisely, we construct a set R' containing

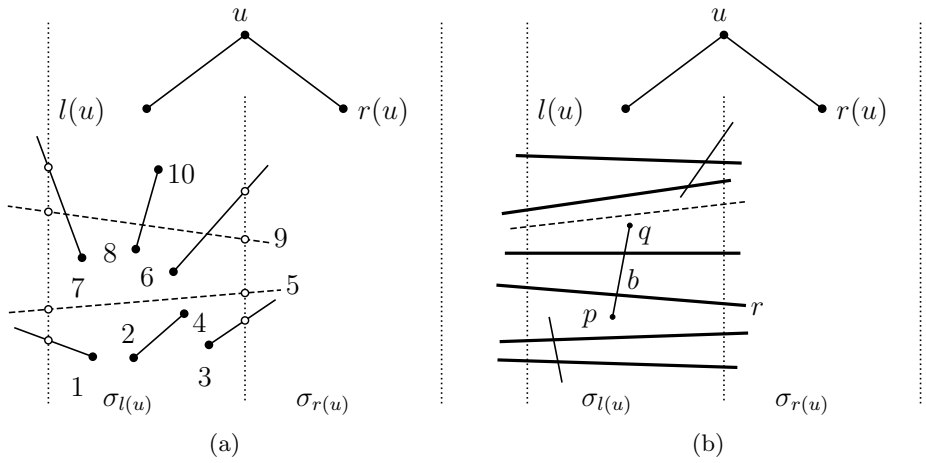


Figure 5.6: (a) Detecting long segments involved in upward short-long intersections. (b) Reporting upward short-long intersections. Dashed segments are not in $R(u)$.

all red segments and one zero-length segment per blue segment endpoint and use the vertical ordering on R' as a total ordering of red segments and blue segment endpoints, bottom-up, see Figure 5.3. The *rank* of a red segment or blue segment endpoint is its position in this ordering.

5.5.1 Populating red lists

To populate all red lists, we initialize the input streams of the merger so that the stream corresponding to slab σ_i stores all red segments whose right endpoints are in σ_i, as well as all blue segment endpoints in σ_i. The entries of the stream are sorted bottom-up (by increasing rank). Now we merge these streams to produce one sorted output stream, where the output stream of each merger node u contains all red segments with right endpoints in σ_u and all blue segment endpoints in σ_u, again sorted bottom-up. The FILL operation at a node u is the standard FILL operation of a $\sqrt{N}$-merger, except that, when placing a red segment r into u's output stream $S(u)$, we check whether r is involved in an upward short-long intersection at node u. If it is, we also append segment r to u's red list $R(u)$.

To see how this test is performed, consider an upward short-long intersection between a short blue segment b and a long red segment r. Segment b must have at least one endpoint in $\sigma_{l(u)}$ that is below r (has lower rank than r). Since b and r intersect in $\sigma_{l(u)}$, either b's other endpoint q also lies in $\sigma_{l(u)}$ and is above r (has higher rank than r), or b intersects one of the slab boundaries of $\sigma_{l(u)}$ above r; see Figure 5.6(a).

Since we merge segments and segment endpoints at each node u bottom-up, we process (ie., place into $S(u)$) all short blue segment endpoints below r before we process r. We call a blue segment *processed* if we have processed at least one of its endpoints. A segment b with one endpoint in $\sigma_{l(u)}$ is *internal*, *left-intersecting*, or *right-intersecting* depending on whether both its endpoints are

in $\sigma_{l(u)}$, b intersects $l(\sigma_{l(u)})$ or b intersects $r(\sigma_{l(u)})$. Let $\rho(u)$ be the highest rank of all endpoints of processed internal blue segments, and $y_l(u)$ the y-coordinate of the highest intersection between $l(\sigma_{l(u)})$ and processed left-intersecting blue segments; $y_r(u)$ is defined analogously for processed right-intersecting blue segments. By our previous discussion, r has an upward short-long intersection at u if and only if r has rank less than $\rho(u)$, intersects $l(\sigma_{l(u)})$ below y-coordinate $y_l(u)$ or intersects $r(\sigma_{l(u)})$ below $y_r(u)$; see Figure 5.6(a).

Values $\rho(u)$, $y_l(u)$, and $y_r(u)$ are easily maintained as the FILL operation at node u processes blue segment endpoints. When processing a red segment r, it is easy to test whether it is long wrt. $\sigma_{l(u)}$ and its rank is less than $\rho(u)$, its intersection with $l(\sigma_{l(u)})$ has y-coordinate less than $y_l(u)$ or its intersection with $r(\sigma_{l(u)})$ has y-coordinate less than $y_r(u)$. If this is the case, r has at least one upward short-long intersection at u, and we append it to u's red list $R(u)$.

5.5.2 Reporting short-long intersections

Given the populated red lists, the second pass starts out with the input stream of each slab σ_i containing all blue segment endpoints in σ_i, sorted top-down (ie., by decreasing ranks). We merge these points so that every node u outputs a stream of blue segment endpoints in σ_u, sorted top-down. To report all short-long intersections at a node u, the FILL operation at node u keeps track of the *current position* in $R(u)$, which is the segment with minimum rank in $R(u)$ we have inspected during the current pass. Initially, this is the last segment in $R(u)$. Now when processing an endpoint $p \in \sigma_{l(u)}$ of a blue segment b, we first scan backwards in $R(u)$ from the current position to find the segment r with minimum rank in $R(u)$ whose rank is greater than that of p. Segment r becomes the new current position in $R(u)$. Segment r is the lowest segment in $R(u)$ that can have an upward intersection with b, and all segments having such intersections with b form a contiguous sequence in $R(u)$ starting with r. Therefore, we scan forward from r, reporting intersections between scanned segments and b until we find the first segment in $R(u)$ that does not have an upward short-long intersection with b; see Figure 5.6(b).

Since every segment placed into $R(u)$ is involved in at least one intersection and all but $O(1)$ accesses to a segment in $R(u)$ can be charged to reported intersections, the scanning of red lists adds only $O(T_s/B)$ to the $O(\text{Sort}(N))$ cost of the merger.

5.5.3 Reducing the space usage

The space usage of the algorithm can be reduced to $O(N + T_s)$ by ensuring that the size of $R(u)$ at node u is bounded by the number of intersections, T_u, reported at u. We can easily compute T_s using a slightly different version of the algorithm to populate the red lists. Whenever we would have put a red segment r into $R(u)$ we simply increment a counter c_u. After the merging completes c_u is the number of elements that would have been placed in u. We now proceed with populating the red lists and reporting interesections as before but allocate room for exactly c_u elements in the red list $R(u)$. Since all elements placed in

$R(u)$ is involved in at least one intersection at u we know that $c_u \leq T_u$ which means that the space needed for all the red lists is $O(\sum_u c_u) = O(T_s)$. Since the merger itself uses linear space the total space used is $O(N + T_s)$.

We can use the trick from [34] to reduce the space to $O(N)$. First we compute the sizes of all the red lists as before. We split the rest of the computation into $\Theta(\frac{T_s}{N})$ *rounds* where each round, except possibly the last, outputs $\Omega(N)$ intersections using $O(N)$ space. Let C_u be the sum of the c_v for all the nodes v in the sub tree rooted at u. At the beginning of each round we perform a post-order traversal of the underlying tree of the merger in which we compute C_u for the nodes u that ve visit. When we encounter the first node u where $C_u > N$ we stop the traversel and allocate all the red lists needed for the sub-tree rooted at u. We then run the algorithms populating the lists and reporting the intersections for the sub-tree rooted at u only, and then proceed to the next round. The output at u is put into a global buffer of size $\Theta(N)$ which is read by the parent of u. The next round continues like the first one by doing a post-order traversal of the merger tree but this time it ignores u and all its descendants. It puts the output of the next round into the same global array at the position corresponding to that node. Since $c_v < N$ for all nodes we know that $C_u \leq 2N$. If no such u is found, we finish by running the algorithm directly on the root of the merger. Since tree traversals can be performed in $O(\text{Sort}(N))$ I/Os [12] each round can be performed in $O(\text{Sort}(N))$ and all (except for the possibly last one) reports $\Theta(N)$ intersections. We get the following lemma.

Lemma 5.2 *Short-long intersections can be reported using* $O(\text{Sort}(N) + T_s/B)$ *memory transfers and linear space.*

5.6 Long-Long Intersections

In this section, we discuss how to find the long-long intersections at all merger nodes. Again, we focus on finding, at every node u, only long-long intersections inside slab $\sigma_{l(u)}$. Similar to the short-long case, we first describe our procedure assuming we can allocate *two* lists of size N to each node. Later we discuss how to reduce the space usage to $O(N)$.

5.6.1 A simple solution using superlinear space

After some preprocessing discussed later in this section, long-long intersections can be found using one pass through the $\sqrt{N}$-merger. This time, the input stream corresponding to slab σ_i contains all segments whose right endpoints are inside σ_i and which intersect $l(\sigma_i)$. The segments are sorted by decreasing y-coordinates of their intersections with $l(\sigma_i)$. The goal of the merge process at a merger node u is to produce an output stream of all segments with right endpoints in σ_u and which intersect $l(\sigma_u)$. Again, these segments are to be output sorted by decreasing y-coordinates of their intersections with $l(\sigma_u)$. In the process of producing its output stream, each merger node u reports all long-long intersections inside $\sigma_{l(u)}$.

This merge process in itself poses a challenge compared to the short-long case, as segments in $S(r(u))$ that intersect both $r(\sigma_{l(u)})$ and $l(\sigma_{l(u)})$ may have to be placed into $S(u)$ in a different order from the one in which they arrive in $S(r(u))$; see Figure 5.7(a). Thus, we need to allow segments to "pass each other", which we accomplish using two buffers $B(u)$ and $R(u)$ of size N associated with each node u in the merger. Buffer $B(u)$ is used to temporarily hold blue segments that need to be overtaken by red segments at u; these segments are sorted by the y-coordinates of their intersections with $l(\sigma_u)$. Buffer $R(u)$ serves the same purpose for red segments. Initially, $B(u)$ and $R(u)$ are empty.

To implement the merge process, we also need a "look-ahead" mechanism that allows each node u to identify the next long segment of each color to be retrieved from $S(r(u))$ without actually retrieving it. We discuss below how to provide such a mechanism. Again, the need for such a mechanism arises because long red and blue segments may change their order between $S(r(u))$ and $S(u)$. If the topmost segment b in $S(r(u))$ is long and blue, we can decide whether it is the next segment to be placed into $S(u)$ only if we know whether the next long red segment r intersects $l(\sigma_u)$ above b; but there may be an arbitrary number of blue and short red segments between b and r in $S(r(u))$, and we cannot afford to scan ahead until we find r in $S(r(u))$. Look-ahead provides us with r without the need to scan through $S(r(u))$.

A Fill operation at node u now reduces to repeatedly identifying the next segment s to be placed into $S(u)$. This segment is currently in $S(l(u))$, $S(r(u))$, $R(u)$ or $B(u)$ and is the one with the highest intersection with $l(\sigma_u)$ among the segments remaining in these streams. Thus, if s belongs to $S(l(u))$, it must be the next segment s' in $S(l(u))$ because the segments in $S(l(u))$ are sorted by their intersections with $l(\sigma_{l(u)}) = l(\sigma_u)$. If s belongs to $S(r(u))$, $R(u)$ or $B(u)$, it must be the next long red segment r or the next long blue segment b to be placed into $S(u)$. Note that our look-ahead mechanism provides us with r and b. To decide which of s', r, and b is the next segment s to be placed into $S(u)$, it suffices to compare their intersections with $l(\sigma_u)$.

In order to place s into $S(u)$, we need to locate it in $S(l(u))$, $S(r(u))$, $B(u)$ or $R(u)$, remove it, and output it into $S(u)$. If $s \in S(l(u))$, $B(u)$ or $R(u)$, this is easy because s is the next segment in $S(l(u))$ or the first segment in $B(u)$ or $R(u)$. So assume that s is long, wlog. red, and stored in $S(r(u))$. Then we retrieve segments from $S(r(u))$ until we retrieve s. Since the segments in $S(r(u))$ are sorted by their intersections with $l(\sigma_{r(u)})$ and red segments do not intersect, there cannot be any long red segment in $S(r(u))$ that is retrieved before s. Thus, all segments retrieved from $S(r(u))$ before s are blue or short. Short segments can be discarded because they cannot be involved in any long-long intersections at u or any of its ancestors. Long blue segments are appended to $B(u)$ in the order they are retrieved, which is easily seen to maintain the segments in $B(u)$ sorted by their intersections with $l(\sigma_{l(u)})$.

So far we have talked only about outputting the segments at each node u in the correct order. To discuss how to report intersections, we say that a segment is placed into $S(u)$ *directly* if it is never placed into $R(u)$ or $B(u)$; otherwise, we say that it is *overtaken* by at least one segment. It is not hard to see that every long-long intersection at a node u involves a segment s placed directly into $S(u)$

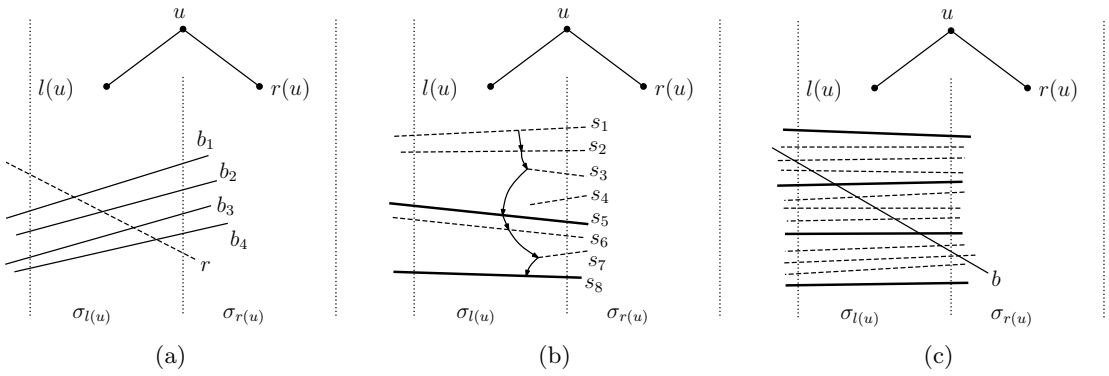


Figure 5.7: (a) Segments b_1, b_2, b_3, b_4 arrive before r in $S(r(u))$ but need to be placed into $S(u)$ after r. Thus, r must be able to overtake them at u. (b) Implementation of look-ahead. Bold solid segments are in $R_t(u)$, dashed ones are not. Arrows indicate how every long segment finds the next long segment. (c) Approximate counting using sampling. The bold segments are in the sample, the dashed ones are not.

and a segment that is overtaken by s; a segment s placed directly into $S(u)$ has long-long intersections with exactly those segments of the other color that are in $B(u)$ or $R(u)$ at the time when s is placed into $S(u)$. Thus, we can augment the merge process at u to report long-long intersections as follows. Immediately before placing a long red segment r directly into $S(u)$, we scan $B(u)$ to report all intersections between r and the segments in $B(u)$. When a long blue segment b is placed directly into $S(u)$, we scan $R(u)$ instead. Since only segments that are overtaken (and thus involved in at least one intersection) are placed into $R(u)$ and $B(u)$ and every scan of $R(u)$ and $B(u)$ reports one intersection per scanned segment, the manipulation of these buffers at all merger nodes adds only $O(T_s/B)$ memory transfers to the $O(\text{Sort}(N))$ cost of the merger. Next we discuss how to implement the look-ahead mechanism using only $O(\text{Sort}(N))$ additional memory transfers, which leads to an $O(\text{Sort}(N) + T_s/B)$ cost for finding all long-long intersections.

5.6.2 Look-ahead

Consider the merge process reporting long-long intersections at a node u. Given look-ahead at u's children, it is easy to ensure that every segment in $S(l(u))$ or $S(r(u))$ knows the next segment s' of the same color in $S(l(u))$ or $S(r(u))$, respectively. When placing a long segment s from $S(r(u))$ into $S(u)$, however, we need to identify not the next segment of the same color as s in $S(r(u))$ but the next *long* such segment s''. If s' is long, then $s'' = s'$. Otherwise, we say that s' *terminates* at node u, as it is not placed into $S(u)$. In this case, s' comes between s and s'' in $S(r(u))$. Note also that every segment terminates at exactly one node in the merger.

To allow us to identify segment s'', we preprocess the merger and associate two lists $R_t(u)$ and $B_t(u)$ with every node u. List $R_t(u)$ (resp., $B_t(u)$) contains

all those long red (resp., blue) segments in $S(r(u))$ that are immediately preceded by red (resp., blue) segments that terminate at u. Given these lists, a long segment s in $S(r(u))$ that is succeeded by a terminating segment of the same color in $S(r(u))$ can identify the next long segment of the same color by retrieving the next segment from $R_t(u)$ or $B_t(u)$, depending on its color; see Figure 5.7(b). These lists are easily constructed in $O(\text{Sort}(N))$ memory transfers by merging the blue and red segments independently. In order to ensure that each list uses only as much space as it needs — and, thus, that all look-ahead lists use only $O(N)$ space — we run each merge twice. The first pass counts the number of segments to be placed into each list, the second one populates the lists after allocating the required space to each list.

During the merge that reports long-long intersections, each list $R_t(u)$ or $B_t(u)$ is scanned exactly once, as the segments in these lists are retrieved in the order they are stored. Thus, scanning these lists uses $O(N/B)$ memory transfers.

5.6.3 Linear space via approximate counting of intersected segments

Finally, we discuss how to reduce the space usage of the merge that finds long-long intersections to $O(N + T_s)$. Using the same technique as in Section 5.5.3, the space usage can then be reduced further to $O(N)$.

To achieve this space reduction, we need to reduce the total size of the red and blue buffers $R(u)$ and $B(u)$ to $O(N + T_s)$. We observe that $R(u)$ and $B(u)$ never contain more than $c_b(u)$ and $c_r(u)$ segments, respectively, where $c_b(u)$ and $c_r(u)$ denote the maximum number of red (resp., blue) segments intersected by any long blue (resp., red) segment at u. Hence, it suffices to determine these values and allocate $c_b(u)$ space for $R(u)$ and $c_r(u)$ space for $B(u)$. Since these values summed over all nodes of the merger do not sum to more than T_s, this would ensure that the total space usage of all buffers $R(u)$ and $B(u)$ is at most T_s. However, it seems difficult to determine $c_b(u)$ and $c_r(u)$ exactly without already using buffers $R(u)$ and $B(u)$. Instead, we compute upper bounds $c'_b(u)$ and $c'_r(u)$ such that $c_b(u) \leq c'_b(u) \leq c_b(u) + \sqrt{N}$ and $c_r(u) \leq c'_r(u) \leq c_r(u) + \sqrt{N}$, which can be done in linear space. By allocating $c'_b(u)$ space for buffer $R(u)$ and $c'_r(u)$ space for buffer $B(u)$, each buffer is big enough and we waste only $O(\sqrt{N})$ space per merger node. Since there are $O(\sqrt{N})$ merger nodes, the total space used by all buffers is therefore $O(N + T_s)$.

We discuss how to compute values $c'_b(u)$, as values $c'_r(u)$ can be computed similarly. To compute values $c'_b(u)$, we compute a $\sqrt{N}/2$-sample of the long red segments passing through each node u and determine for every long blue segment b how many segments in the sample it intersects. If this number is $h(b)$, then b intersects between $\sqrt{N}(h(b) - 1)/2$ and $\sqrt{N}(h(b) + 1)/2$ long red segments at node u. See Figure 5.7(c). We choose $c'_b(u)$ to be the maximum of $\sqrt{N}(h(b) + 1)/2$ taken over all long blue segments b at node u.

More precisely, we use two passes through the $\sqrt{N}$-merger after allocating a sample buffer $R_s(u)$ of size $2\sqrt{N}$ to each node. The first pass merges red segments by their intersections with left slab boundaries. At a node u, every

$\sqrt{N}/2$'th long segment is placed into $R_s(u)$. The second pass merges blue segments by their intersections with left slab boundaries. Before this pass, we set $c'_b(u) = 0$ for every node u. During the merge, when we process a long blue segment b, we determine the number $h_l(b)$ of segments in $R_s(u)$ that intersect $l(\sigma_{l(u)})$ below b, as well as the number $h_r(b)$ of segments in $R_s(u)$ that intersect $r(\sigma_{l(u)})$ below r. Let $h(b) = |h_r(b) - h_l(b)|$. If $\sqrt{N}(h(b)+1)/2 > c'_b(u)$, we set $c'_b(u) = \sqrt{N}(h(b)+1)/2$.

Since we allocate only $O(\sqrt{N})$ space to each merger node during the approximate counting of intersections, the space usage of this step is linear. Moreover, we merge red and blue segments once, and it can be shown that the computation of values $h_r(b)$ and $h_l(b)$ for all blue segments b passing through node u requires two scans of list $R_s(u)$ in total. Hence, this adds $O(N/B)$ to the merge cost, and we obtain the following lemma, which completes the proof of Theorem 5.3.

Lemma 5.3 *Long-long intersections can be reported using* $O(\mathrm{Sort}(N) + T_s/B)$ *memory transfers and linear space.*

Part II

Resilient Algorithms

Chapter 6

Background

In this chapter we give a short overview of resilient algorithms and data structures developed for the faulty memory RAM. A more comprehensive overview of the resilient algorithmics field can be found in a recent survey paper by Finocchi, Grandoni and Italiano [57].

Recall, from Chapter 1, that a sequence of elements is *faithfully ordered* if all the uncorrupted elements in the sequence appear in sorted order. An algorithm is *resilient* if it performs correctly on the set of uncorrupted elements. For instance, a resilient sorting algorithm produces a faithfully ordered sequence.

The complexity of resilient algorithms can be expressed in two slightly different ways. For example, a resilient sorting algorithm using $O(N \log N + \delta^2)$ time can also be described as a sorting algorithm that uses $O(N \log N)$ time while tolerating $O(\sqrt{N \log N})$ corruptions. While the former version is more general the latter emphasises the fact that one can use a resilient sorting algorithm for "free" asymptotically as long as the number of corruptions is $O(\sqrt{N \log N})$.

In this chapter we introduce a few resilient algorithms. We present a simple but important technique used to store variables reliably in the unreliable memory and then give an overview of an optimal resilient sorting algorithm. After this we will define the problems that are considered in this part of the dissertation and give a brief overview of the results we achieve.

In the following three chapters we present a number of resilient algorithms that have appeared in works coauthored by the author of this dissertation. The main focus of the chapter is on resilient dictionaries but we will also present a resilient priority queue and an I/O-efficient sorting algorithm. We will also review other resilient algorithms that are relevant for these results.

6.1 Reliable Values

An important concept used in the design of resilient algorithms is that of *reliable values*, which are variables that are stored in the unreliable memory such that they can still be reliably retrieved and updated. The algorithm that makes this possible is based on a simple algorithm solving the *majority problem*. It serves as a good first example of an algorithm that works correctly in the faulty memory RAM.

6.1.1 The majority problem

In the majority problem we are given a sequence of k memory cells and the promise that at least $\lceil \frac{k+1}{2} \rceil$ of the cells contain the same element x. The goal is to recover x from the k cells. In the standard comparison model there is a simple algorithm by Boyer and Moore [33] for extracting x using $O(1)$ space and $O(k)$ time. The algorithm keeps two variables in memory: a majority candidate a and a counter c. Initially a is set as the first element in the sequence and c is one. We now scan the elements once and perform one of the following three operations. If c is zero we pick the current element as the new candidate stored in a and set c to one. If the current element is equal to a we increment c by one, and if they differ we decrement c by one. At the end of this algorithm a is the majority element of the sequence.[1] The algorithm only work correctly if the set contains a majority element, and the original paper [33] suggests rescanning the input to confirm that a is indeed the majority if it is not known in advance whether such a majority exists.

We can use the algorithm above to reliably maintain a variable v. We simply store v using $2\delta + 1$ cells, each containing the value v. Since at most δ corruptions can take place, we know that corruptions can not change the fact that the majority of the $2\delta + 1$ cells contain v. Thus we can extract v by running the majority extraction algorithm while keeping the candidate element, a, and the counter v in the $O(1)$ safe memory cells. A variable v stored this way is denoted a *reliable variable*. It follows from the discussion above that a reliable variable can be maintained using $O(\delta)$ space and can be accessed and updated in $O(\delta)$ time.

6.2 Resilient Sorting

It was proven by Finocchi and Italiano [61] that resilient comparison-based sorting requires $\Omega(N \log N + \delta^2)$ comparisons and this lower bound was later matched in their paper with Grandoni [59]. The sorting algorithm is based on merge sort but uses a modified merging routine to achieve resilience.

6.2.1 Resilient merging

We now describe the resilient merging algorithm of [59] that merges two faithfully[2] ordered input sequences X and Y of total size N into a faithfully ordered sequence. It uses $O(N + \alpha\delta)$ comparisons, where α is the number of corrupted elements encountered. We refer to [59] for the full description and analysis.

The merging algorithm uses two different sub-routines: *PurifyingMerge* and *UnbalancedMerge*. The former uses $O(N + \alpha\delta)$ comparisons to merge X and Y into an faithfully ordered output sequence Z and a *fail buffer* F of size $O(\alpha)$

[1] The original paper describes this algorithm using a humorous analogy to an election floor where a fight breaks loose. During the fight delegates from different blocks knock each other down in pairs with placards. The delegate(s) that remain standing after this fight belong to the majority block.

[2] Recall that a sequence is faithfully ordered if the uncorruped elements in the sequence appear in sorted order.

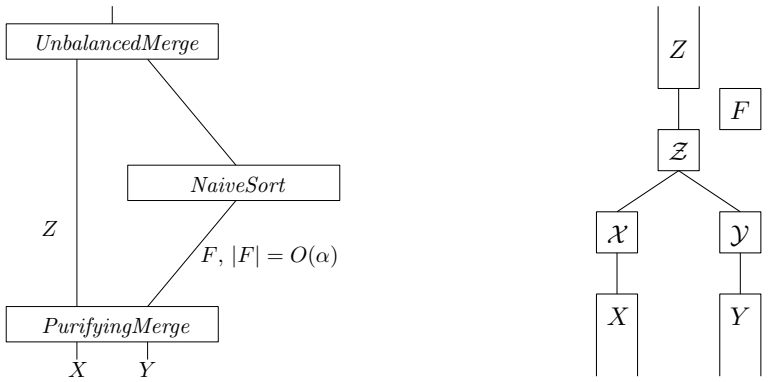


Figure 6.1: (a) The structure of the resilient merging algorithm. The input sequences are merged by the *PurifyingMerge* algorithm which creates a $O(\alpha)$ sized fail buffer which is then merged into the output sequence by the *UnbalancedMerge* algorithm. (b) Overview of the various buffers used in the *PurifyingMerge* algorithm.

containing elements that could not be merged into Z due to corruptions. The fail buffer is then sorted in $O(\alpha^2) = O(\alpha\delta)$ time by a naive sorting algorithm that repeatedly scans F and extracts the smallest remaining element. Finally the *UnbalancedMerge* algorithm, described in [61], is used to merge the sorted fail buffer into Z. It gets its name from the fact that the algorithm is most efficient when N_1 is much bigger than N_2. *UnbalancedMerge* performs $O(N_1 + (N_2 + \alpha)\delta)$ comparisons to merge a two sequences with N_1 and N_2 elements, respectively. In our case the unbalanced merge is completed in $O(|Z| + (|F| + \alpha)\delta) = O(N + \alpha\delta)$ time. Thus, the total cost of the resilient merging algorithm is $O(N + \alpha\delta)$. See Figure 6.1(a) for an overview of the structure of the merging algorithm.

We will now summarize the *PurifyingMerge* algorithm. The input to the algorithm are two faithfully ordered sequences X and Y and the output is a fail buffer of size $O(\alpha)$ containing some of the elements from X and Y as well as a new faithfully ordered sequence that contains the rest of the elements. The algorithm uses $O(n + \alpha\delta)$ comparisons. The merging takes place in *rounds* where each round uses auxiliary buffers $\mathcal{X}$,$\mathcal{Y}$ and $\mathcal{Z}$. The first two buffers contain the first $2\delta + 1$ elements of X and Y, respectively. The $\mathcal{Z}$ buffer contains the output elements of a single round and is of size δ, see Figure 6.1(b). We also maintain offsets into each of these buffers that track the progress of the algorithm, the initial offset for the $\mathcal{X}$ and $\mathcal{Y}$ buffers is $2\delta + 1$ and the offset for $\mathcal{Z}$ is δ. In each round we repeatedly consider the current elements of $\mathcal{X}$ and $\mathcal{Y}$ and copy the smallest of these to $\mathcal{Z}$, we also decrement the offset for the corresponding buffer, and for $\mathcal{Z}$. However, if the current element from $\mathcal{X}$ ($\mathcal{Y}$) is smaller than the last element extracted from $\mathcal{X}$ ($\mathcal{Y}$) we move both elements to the fail buffer and append two new elements to $\mathcal{X}$ ($\mathcal{Y}$) and restart the round (including emptying $\mathcal{Z}$ and resetting all offsets). A round ends when $\mathcal{X}$ or $\mathcal{Y}$ becomes empty (the indices for one of them moves past the end of the buffer) or when δ elements of output have been copied to $\mathcal{Z}$. When the round ends we check that the

outputted elements are still consistent with the remaining elements in $\mathcal{X}$ and $\mathcal{Y}$ and if so we move the elements in $\mathcal{Z}$ to the output buffer Z and a new round is started.

The crucial point in the analysis of *PurifyingMerge* is that a single round is only (re-)started if δ elements have been output to Z or if a corrupted element was found. In the latter case the corrupted element is provably one of the two elements moved to $\mathcal{Z}$ and thus, each corruption can only cause a round to restart once. This also proves that $|F| = O(\delta)$. Since a round costs $O(\delta)$ and outputs δ elements if it is not restarted, the total time for the merging algorithm is $O(N + \alpha'\delta)$, where α' is the number of corrupted elements encountered during the round.

Lemma 6.1 ([59]) PurifyingMerge *merges two sequences with a total of N elements in $O(N + \alpha\delta)$ time.*

Using *PurifyingMerge* we get a resilient sorting algorithm with the desired bound:

Theorem 6.1 ([59]) *The resilient sorting algorithm uses $O(N \log N + \alpha\delta)$ time.*

6.3 Contributions

In this section we will give a brief summary of the results presented in the following chapters where we present resilient data structures for priority queues and dynamic dictionaries, and we also combine the faulty memory RAM with the I/O-model and get algorithms that are both resilient and I/O-efficient. For the discussion and complete definition on this hybrid model we refer to Chapter 9.

6.3.1 Priority queue

In Chapter 7 we design and analyze a priority queue in the faulty-memory RAM model. A resilient dictionary is defined in the following way:

Definition 6.1 (Resilient Priority Queue) *A resilient priority queue maintains a set of elements under the operations* Insert *and* Deletemin*. An* Insert *adds an element and a* Deletemin *deletes and returns the minimum uncorrupted element or a corrupted one.*

Note that our definition of a resilient priority queue is consistent with the definition of resilient sorting as described in Section 6.2; given a sequence of N elements, inserting all of them into a resilient priority queue followed by N Deletemin operations yields a faithfully ordered sequence.

The resilient priority queue that we present uses $O(N)$ space for storing N elements and performs both Insert and Deletemin in $O(\log N + \delta)$ time amortized. Thus, it can tolerate $O(\log N)$ corruptions while matching the bounds of standard optimal comparison-based priority queues. We also prove that this is optimal under certain assumptions about how a structure uses the reliable memory.

At the time of publication, the priority queue was the only deterministic data structure able to tolerate $O(\log N)$ corruptions while still matching optimal bounds in the standard comparison model. However, the dynamic dictionary presented in Chapter 8 now has the same bounds. Although this dictionary can also be used as an priority queue, the structure presented in Chapter 7 is considerably simpler.

In Chapter 9 we present an I/O-efficient resilient priority queue for the case where δ is smaller than M^ε. The priority queue supports INSERT and DELETEMIN in optimal $O(\frac{1}{1-\varepsilon}(1/B)\log_{M/B}(N/M))$ I/Os amortized. This matches the bounds for non-resilient external memory priority queues.

6.3.2 Dictionaries

A resilient dictionary is defined in the following way:

Definition 6.2 (Resilient Dictionary) *A resilient dictionary maintains a set S while supporting membership queries. When given an element e, the membership query algorithm must return a positive answer if there exists an uncorrupted element $x = e$ in S. If there is no such element, corrupted or uncorrupted, the algorithm must return a negative answer. If there is a corrupted element $x' = e$, the answer is undefined.*

Finocchi and Italiano [61] prove that any resilient comparison-based searching algorithm can tolerate at most $O(\log N)$ corruptions if it uses $O(\log N)$ time and in Chapter 8 we present the first static dictionary that matches this lower bound. The dictionary uses just a sorted array and a simple searching algorithm based on binary search. Additionally, we present an even simpler randomized query algorithm that achieves the same $O(\log N + \delta)$ bound in expectancy. Finally, in the same chapter we show how we can use this dictionary to design a dynamic dictionary with the same $O(\log N + \delta)$ worst-case query bound and amortized $O(\log N + \delta)$ update bounds.

In Chapter 9 we adopt the dictionaries above to the I/O-model and present lower and upper bounds for I/O-efficient resilient dictionaries, see the chapter for details.

6.3.3 I/O-efficient resilient sorting

In Chapter 9 we also describe how one can adopt the resilient merging algorithm defined in this chapter to design resilient *multi-way* merging algorithm. We subsequently use this algorithm to design an optimal resilient sorting algorithm using $O(\frac{1}{1-\varepsilon}\text{Sort}(N))$ I/Os and $O(N \log N + \alpha\delta)$ time under the assumption that $\delta \leq M^\varepsilon$, for $0 \leq \varepsilon < 1$. This matches the optimal resilient and I/O-efficient comparison-based sorting algorithms for constant ε.

6.3.4 Resilient Counting

In a recently submitted paper [C4] cauthored by the author of this thesis, we consider the problem of maintaining many counters in the faulty memory RAM.

The paper is not presented in this dissertation. It is trivial to maintain $O(1)$ counters by simply keeping them in the reliable memory. The paper studies counters whose state is not stored in the reliable memory and must therefore be kept in unreliable memory. In one extreme, counters can implemented using reliable values by using $O(\delta)$ time per access. We show that by sacrifying the accuracy of the counters and allowing some controlled additive error, we can get faster update times, and we present upper and lower bound time-accuracy trade-offs for performing N increment operations.

Chapter 7

Resilient Priority Queue

The resilient priority queue that we present in this chapter uses $O(N)$ space for storing N elements and performs both INSERT and DELETEMIN in $O(\log N + \delta)$ time amortized. See Chapter 6 for the definition of the INSERT and DELETEMIN operations. The priority queue matches the bounds for an optimal comparison-based priority queue in the RAM model while tolerating $O(\log N)$ corruptions. It is a significant improvement over using the resilient search tree in [58] as a priority queue, since it uses $O(\log N + \delta^2)$ time amortized per operation and thus only tolerates $O(\sqrt{\log N})$ corruptions to preserve the $O(\log N)$ bound per operation. Our priority queue is the first resilient data structure allowing $O(\log N)$ corruptions, while still matching optimal bounds in the RAM model. Our priority queue does not store elements in reliable memory between operations, only structural information like pointers and indices. We prove that any comparison-based resilient priority queue behaving this way requires worst case $\Omega(\log N + \delta)$ time for either INSERT or DELETEMIN.

The resilient priority queue is based on the cache-oblivious priority queue by Arge et al. [13]. The main idea is to gather elements in large sorted groups of increasing size, such that expensive updates do not occur too often. The smaller groups contain the smaller elements, so they can be retrieved faster by DELETEMIN operations. We extensively use the resilient merging algorithm in [56] to move elements among the groups. Due to the large sizes of the groups, the extra work required to deal with corruptions in the merging algorithm becomes insignificant compared to the actual work done.

Given two sequences X and Y, we let XY denote the *concatenation* of X and Y. A sequence X is *faithfully ordered* if its uncorrupted keys appear in non-decreasing order.

7.1 Fault tolerant priority queue

In this section we introduce the resilient priority queue. The elements are stored in faithfully ordered lists and are moved using two fundamental primitives, PUSH and PULL, based on faithful merging. We describe the structure of the priority queue in Section 7.1.1 and then introduce the PUSH and PULL primitives in Section 7.1.2. Finally, in Section 7.1.3, we describe the INSERT and DELETEMIN

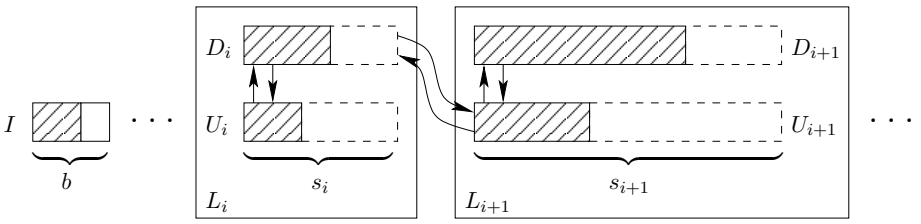


Figure 7.1: The structure of the priority queue. The buffers are stored in a doubly linked list using reliably stored pointers. Additionally, the size of each buffer is stored reliably.

operations.

7.1.1 Structure

The resilient priority queue consists of an insertion buffer I together with a number of layers $L_0, \ldots, L_k$, with $k = O(\log N)$. Each layer L_i contains an up-buffer U_i and a down-buffer D_i, represented as arrays. Intuitively, the up-buffers contain large elements that are on their way to the upper layers in the priority queue, whereas the down-buffers contain small elements, on their way to lower layers. The buffers in the priority queue are stored as a doubly linked list $U_0, D_0, \ldots, U_k, D_k$, see Figure 7.1. For each up and down buffer we reliably store the pointers to their adjacent buffers in the linked list and their size. In the reliable memory we store pointers to I, U_0 and D_0, together with $|I|$. Since the position of the first element in U_0 and D_0 is not always the first memory cell of the corresponding buffer, we also store the index of the first element in these buffers in reliable memory. The insertion buffer I contains up to $b = \delta + \log N + 1$ elements. For layer L_i we define the threshold s_i by $s_0 = 2 \cdot (\delta^2 + \log^2 N)$ and $s_i = 2s_{i-1} = 2^{i+1} \cdot (\delta^2 + \log^2 N)$, where N is the number of elements in the priority queue. We use these thresholds to decide whether an up buffer contains too many elements or whether a down buffer has too few. For the sake of simplicity, the up and down buffers are grown and shrunk as needed during the execution such that they don't use any extra space.

To structure the priority queue, we maintain the following invariants for the up and down buffers.

- *Order invariants:*
  1. All buffers are faithfully ordered.
  2. D_iD_{i+1} and D_iU_{i+1} are faithfully ordered, for $0 \leq i < k$.
- *Size invariants:*
  3. $s_i/2 \leq |D_i| \leq s_i$, for $0 \leq i < k$.
  4. $|U_i| \leq s_i/2$, for $0 \leq i < k$.

By maintaining all the up and down buffers faithfully ordered, it is possible to move elements between neighboring layers efficiently, using faithful merging. By invariant 2, all uncorrupted elements in D_i are smaller than all uncorrupted elements in both D_{i+1} and U_{i+1}. This ensures that small elements belong to the lower layers of the priority queue. We note that there is no assumed relationship between the elements in the up and down buffers in the same layer. Finally, the size invariants allow the sizes of the buffers to vary within a large range. This way, $\Omega(s_i)$ INSERT or DELETEMIN operations occur between two operations on the same buffer in L_i, yielding the desired amortized bounds.

Since the s_i values depend on N, whenever the size of the priority queue increases or decreases by $\Theta(N)$, we perform a global rebuilding. This rebuilding is done by collecting all elements, sorting them with the optimal resilient sorting algorithm from Chapter 6, and redistributing the output into the down buffers of all the layers starting with L_0. After the global rebuilding, the up buffers are empty and the down buffers full, except possibly the last down buffer.

7.1.2 Push and pull primitives

We now introduce the two fundamental primitives used by the priority queue. The PUSH primitive is invoked when an up buffer contains too many elements, breaking invariant 4. It "pushes" elements upwards, repairing the size invariants locally. The PULL operation is invoked when a down buffer contains too few elements, breaking invariant 3. It fills this down buffer by "pulling" elements from the layer above, again locally repairing the size invariants. Both operations faithfully merge consecutive buffers in the priority queue and redistribute the resulting sequence among the participating buffers. After merging, we deallocate the old buffers and allocate new arrays for the new buffers.

Push. The PUSH primitive is invoked when an up buffer U_i breaks invariant 4, i.e., when it contains more than $s_i/2$ elements. In this case we merge U_i, D_i and U_{i+1} into a sequence M using the resilient merging algorithm in [56]. We then distribute the elements in M by placing the first $|D_i| - \delta$ elements in a new buffer D'_i, and the remaining $|U_{i+1}| + |U_i| + \delta$ elements in a new buffer U'_{i+1}. After the merge, we create an empty buffer, U'_i, and deallocate the old buffers. If U'_{i+1} contains too many elements, breaking invariant 4, the PUSH primitive is invoked on U'_{i+1}. When L_i is the last layer, we fill D'_i with the first elements of M and create a new layer L_{i+1} placing the remaining elements of M into D'_{i+1} instead of U'_{i+1}. Since $|D'_i|$ is smaller than $|D_i|$, it could violate invariant 3. This situation is handled by using the PULL operation and is described after introducing PULL.

Unlike the priority queue in [13], the PUSH operation decreases the size of a down buffer. This is required to preserve invariant 2, in spite of corruptions. After a PUSH call, D'_i can contain elements from $U_i \cup U_{i+1}$. Since there is no assumed relationship between elements in $U_i \cup U_{i+1}$ and those in $D_{i+1} \cup U_{i+2}$, we need to ensure that each element in D'_i originating from $U_i \cup U_{i+1}$ is faithfully smaller than the elements in $D_{i+1} \cup U_{i+2}$. Assume the size of D_i is preserved, i.e. $|D'_i| = |D_i|$. Consider a corruption that alters an element in D_i to some

large value before the PUSH. This corrupted value could be placed in U'_{i+1} and, since $|D'_i| = |D_i|$, an element from $U_i \cup U_{i+1}$ must be placed in D'_i. This new element in D'_i potentially violates invariant 2.

Pull. The PULL operation is called on a down buffer D_i when it contains less than $s_i/2$ elements, breaking invariant 3. In this case, the buffers D_i, U_{i+1}, and D_{i+1} are merged into a sequence M using the resilient merging algorithm in [56]. The first s_i elements from M are written to a new buffer D'_i, and the next $|D_{i+1}| - (s_i - |D_i|) - \delta$ elements are written to D'_{i+1}. The remaining elements of M are written to U'_{i+1}. A PULL is invoked on D'_{i+1}, if it is too small.

Similar to the PUSH operation, the extra δ elements lost by D_{i+1} ensure that the order invariants hold in spite of possible corruptions. That is, a corruption of an element in $D_i \cup D_{i+1}$ to a very large value may cause an element from U_{i+1} to take the place of the corrupted element in D'_{i+1} and this element is possibly larger than some uncorrupted element in $D_{i+2} \cup U_{i+2}$.

After the merge, U'_{i+1} contains δ more elements than U_{i+1} had before the merge, and thus it is possible that it has too many elements, breaking invariant 4. We handle this situation as follows. Consider a maximal series of subsequent PULL invocations on down buffers $D_i, D_{i+1}, \ldots, D_j$, $0 \leq i < j < k$. After the first PULL call on D_i and before the call on D_{i+1} we store a pointer to D_i in the reliable memory. After all the PULL calls we investigate all the affected up buffers, by simply following the pointers between the buffers starting from D_i, and invoke the PUSH primitive wherever necessary. The case when PUSH operations cause down buffers to underflow is handled similarly.

7.1.3 Insert and deletemin

An element is inserted in the priority queue by simply appending it to the insertion buffer I. If I gets full, its elements are added to U_0 by first faithfully sorting I and then faithfully merging I and U_0. If U_0 breaks invariant 4, we invoke the PUSH primitive. If L_0 is the only layer of the priority queue and D_0 violates the size constraint, we faithfully merge the elements in I with D_0 instead.

To delete the minimum element in the priority queue, we first find the minimum of the first $\delta + 1$ values in D_0, the minimum of the first $\delta + 1$ values in U_0, and the minimum element in I. We then take the minimum of these three elements, delete it from the appropriate buffer and return it. After deleting the minimum, we right-shift all the elements in the affected buffer from the beginning up to the position of the minimum. This way we ensure that elements in any buffer are stored consecutively. If D_0 underflows, we invoke the PULL primitive on D_0, unless L_0 is the only layer in the priority queue. If U_0 or D_0 contains $\Theta(\log N + \delta)$ empty cells, we create a new buffer and copy the elements from the old buffer to the new one.

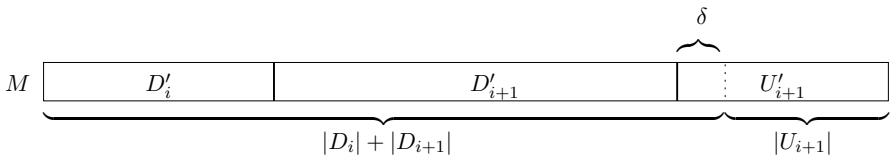


Figure 7.2: The distribution of M into buffers.

7.2 Analysis

In this section we analyze the resilient priority queue. We prove the correctness in Section 7.2.1 and analyze the time and space complexity in Section 7.2.2.

7.2.1 Correctness

To prove correctness of the resilient priority queue, we show that the DELETEMIN operation returns the minimum uncorrupted value or a corrupted value. We first prove that the order invariants are maintained by the PULL and PUSH operations.

Lemma 7.1 *The* PULL *and* PUSH *primitives preserve the order invariants.*

Proof. Recall that in a PULL invocation on buffer D_i, the buffers D_i, U_{i+1}, and D_{i+1} are faithfully merged into a sequence M. The elements in M are then distributed into three new buffers D'_i, U'_{i+1}, and D'_{i+1}, see Figure 7.2. To argue that the order invariants are satisfied we need to show that the elements of the down buffer on layer L_j, for $0 \leq j < k$, are faithfully smaller than the elements of the buffers on layer L_{j+1}, where k is the index of the last layer. The invariants hold trivially for unaffected buffers. The faithful merge guarantees that $D'_iD'_{i+1}$ as well as $D'_iU'_{i+1}$ are faithfully ordered, and thus the individual buffers are also faithfully ordered. Since invariant 2 holds for the original buffers all uncorrupted elements in D_{i+1} and U_{i+1} are larger than the uncorrupted elements in D_i, guaranteeing that $D_{i-1}D'_i$ is faithfully ordered. Finally, we now show that $D_{i+1}D_{i+2}$ and $D_{i+1}U_{i+2}$ are faithfully ordered.

Let m be the minimum uncorrupted element in $D_{i+2} \cup U_{i+2}$. We need to show that all uncorrupted elements in D'_{i+1} are smaller than m. If no uncorrupted element from U_{i+1} is placed in D'_{i+1}, the invariant holds by the order invariants before the operation. Otherwise, assume that an uncorrupted element $y \in U_{i+1}$ is moved to D'_{i+1}. Since $|U'_{i+1}| = |U_{i+1}| + \delta$ and y is moved to D'_{i+1}, at least $\delta + 1$ elements originating from $D_i \cup D_{i+1}$ are contained in U'_{i+1}. Since there can be at most δ corruptions, there exists at least one uncorrupted element, x, among these. By faithful merging, all uncorrupted elements in D'_{i+1} are smaller than x, which means that $y \leq x$. Since x originates from $D_i \cup D_{i+1}$, it is smaller than m. We obtain $y \leq m$.

A similar argument proves correctness of the PUSH operation. We conclude that both order invariants are preserved by PULL and PUSH operations. □

Having proved that the order invariants are maintained at all times, we now prove the correctness of the resilient priority queue.

Lemma 7.2 *The* DELETEMIN *operation returns the minimum uncorrupted value in the priority queue or a corrupted value.*

Proof. We recall that the DELETEMIN operation computes the minimum of the first $\delta + 1$ elements of U_0 and D_0. It compares these values with the minimum of I, found in a scan, and returns the smallest of these elements. Since U_0 and D_0 are faithfully ordered, the minimum of their first $\delta + 1$ elements is either the minimum uncorrupted value in these buffers, or a corrupted value even smaller. Furthermore, according to the order invariants, all the values in layers $L_1, \ldots, L_k$ are faithfully larger than the minimum in D_0. Therefore, the element reported by DELETEMIN is the minimum uncorrupted value or a corrupted value. □

7.2.2 Complexity

In this section we show that our resilient priority queue uses $O(N)$ space and that INSERT and DELETEMIN take $O(\log N + \delta)$ amortized time. We first prove that the PULL and PUSH primitives restore the size invariants.

Lemma 7.3 *If a size invariant is broken for a buffer in L_0, invoking* PULL *or* PUSH *on that buffer restores the invariants. Furthermore, during this operation* PULL *and* PUSH *are invoked on the same buffer at most once. No other invariants are broken before or after this operation.*

Proof. Assume that PUSH is invoked on U_0, and that it is called iteratively up to some layer L_l. By construction of PUSH, the size invariants for all the up buffers now hold. Since a PUSH steals δ elements from the down buffers, the layers $L_0, \ldots, L_l$ are traversed again and PULL is invoked on these as needed. The last of these PULL operations might proceed past layer L_l. Similarly, a PULL may cause an up buffer to overflow. However, since the cascading PUSH operations left $|U_i| = 0$ for $i \leq l$, any new PUSH are invoked on up buffers only on layer L_{l+1} or higher, thus PUSH is invoked on each buffer at most once. A similar argument works for the PULL operation. □

Lemma 7.4 *The resilient priority queue uses $O(N + \delta)$ space to store N elements.*

Proof. The insertion buffer always uses $O(\log N + \delta)$ space. We prove that the remaining layers use $O(N)$ space. For each layer we use $O(\delta)$ space for storing structural information reliably. In all layers, except the last one, the down buffer contains $\Omega(\delta^2)$ elements by invariant 3. This means that for each of these layers the elements stored in the down buffer dominate the space complexity. The structural information of the last layer requires additional $O(\delta)$ space. □

The space complexity of the priority queue can be reduced to $O(N)$ without affecting the time complexity, by storing the structural information of L_0 in safe memory, and by doubling or halving the insertion buffer during the lifetime of the algorithm such that it always uses $O(|I|)$ space.

Lemma 7.5 *Each* INSERT *and* DELETEMIN *takes* $O(\log N + \delta)$ *amortized time.*

Proof. We define the potential function:

$$\Phi = \sum_{i=1}^{k} (c_1 \cdot (\log N - i) \cdot |U_i| + c_2 \cdot i \cdot |D_i|)$$

We use Φ to analyze the amortized cost of a PUSH operation. In a PUSH operation on U_i, buffers U_i, D_i, and U_{i+1} are merged. The elements are then distributed into new buffers U_i', D_i', and U_{i+1}', such that $|U_i'| = 0$, $|D_i'| = |D_i| - \delta$, and $|U_{i+1}'| = |U_{i+1}| + |U_i| + \delta$. This gives the following change in potential $\Delta\Phi$:

$$\begin{aligned} \Delta\Phi &= -|U_i| \cdot c_1 \cdot (\log N - i) - \delta \cdot c_2 \cdot i + (|U_i| + \delta) \cdot c_1(\log N - (i+1)) \\ &= -c_1 \cdot |U_i| + \delta(-c_2 \cdot i + c_1 \cdot \log N - c_1 \cdot i - c_1) \ . \end{aligned}$$

Since the PUSH is invoked on U_i, invariant 4 is not valid for U_i and therefore $|U_i| \geq \frac{s_i}{2} = 2^i \ (\log^2 \ n \ + \ \delta^2)$. Thus:

$$\Delta\Phi \leq -c_1 \cdot |U_i| + c_1 \cdot \delta \cdot \log N \leq -c_1 \cdot 2^i \cdot (\log^2 N + \delta^2) + c_1 \cdot \delta \cdot \log N \leq -c_1 \cdot c' \cdot |U_i| \ , \tag{7.1}$$

for some constant $c' > 0$.

Since faithfully merging two sequences of size N takes $O(N + \delta^2)$ time [56], the time used for a PUSH on U_i is upper bounded by $c_m \cdot (|U_i| + |D_i| + |U_{i+1}| + \delta^2)$, where c_m depends on the resilient merge. This includes the time required for retrieving reliably stored variables. Adding the time and the change in potential we are able to get the amortized cost less than zero by tweaking c_1 based on equation (7.1). This is because $|U_i|$ is $\Omega(\delta^2)$ and at most a constant fraction smaller than the participants in the merge.

A similar analysis works for the PULL primitive. We now calculate the amortized cost of INSERT and DELETEMIN. We ignore any PUSH or PULL operations since their amortized costs are negative. The amortized time for inserting an element in I, sorting I, and merging it with U_0 is $O(\log N + \delta)$ per operation. The change in potential when adding elements to L_0 is $O(\log N)$ per element. The time needed to find the smallest element in a DELETEMIN is $O(\log N + \delta)$, and the change in potential when an element is deleted from L_0 is negative.

The cost of global rebuilding is dominated by the cost of sorting, which is $O(N \log N + \delta^2)$. There are $\Theta(N)$ operations between each rebuild, which leads to $O(\log N + \delta)$ time per operation, since $\delta \leq N$, and this concludes the proof. □

Theorem 7.1 *The resilient priority queue takes* $O(N)$ *space and uses amortized* $O(\log N + \delta)$ *time per operation.*

7.3 Lower bound

In this section we prove that any resilient priority queue takes $\Omega(\log N + \delta)$ time for either INSERT or DELETEMIN in the comparison model, under the assumption that no elements are stored in reliable memory between operations. This implies optimality of our resilient priority queue under these assumptions. We note that the reliable memory may contain any structural information, e.g. pointers, sizes, indices.

Theorem 7.2 *A resilient priority queue containing N elements, with $N > \delta$, uses $\Omega(\log N + \delta)$ comparisons to perform* INSERT *followed by* DELETEMIN.

Proof. Consider a priority queue Q with N elements, with $N > \delta$, that uses less than δ comparisons for an INSERT followed by a DELETEMIN. Also, Q does not store elements in reliable memory between operations. Assume that no corruptions have occurred so far. Without loss of generality we assume that all the elements in Q are distinct. We prove there exists a series of corruptions C, $|C| \leq \delta$, such that the result of an INSERT of an element e followed by a DELETEMIN returns the same element regardless of the choice of e.

Let $k < \delta$ be the number of comparisons performed by Q during the two operations. We force the result of each comparison to be the same regardless of e by suitable corruptions. In all the comparisons involving e, we ensure that e is the smallest. We do so by corrupting the value which e is compared against if necessary, by adding some positive constant $c \geq e$ to the other value. If two elements different than e are compared, we make sure the outcome is the same as if no corruptions had happened. If one of them was corrupted, adding c to the other one reestablishes their previous ordering. If both of them were corrupted by adding c, their ordering is unchanged and no corruptions are needed. Forcing any comparison to give the desired outcome requires at most one corruption, and therefore $|C| \leq k < \delta$.

We now consider the value e' returned by DELETEMIN on Q. If $e = e'$ then we choose e to be larger than some element $x \in Q$ not affected by a corruption in C. Such a value exists because the size of the priority queue is larger than δ. Since $e = e' > x$, Q returned an uncorrupted element that was not the minimum uncorrupted element in Q. If $e \neq e'$ we choose e to be smaller than any element in Q. With such a choice of e, no corruptions are required and the value returned by Q was not corrupted, but still larger than e. This proves Q is not resilient.

Adding the classical $\Omega(\log N)$ bound for priority queues in the comparison model the result follows. □

Chapter 8

Resilient Dictionaries

In this chapter we describe two optimal resilient static dictionaries, a randomized one and a deterministic one, as well as a dynamic dictionary.

Randomized static dictionary: We introduce a resilient randomized static dictionary that support searches in $O(\log N+\delta)$ time, matching the bounds for randomized searching in [56]. We note however that our dictionary is somewhat simpler and uses only $O(\log \delta)$ worst case random bits, whereas the algorithm in [56] uses expected $O(\log \delta \cdot \log N)$ random bits. On the downside, our dictionary assumes that the corruptions are performed by a non-adaptive adversary, i.e. an adversary that does not perform corruptions based on the behavior of the algorithm.

Deterministic static dictionary: We give the first optimal resilient static deterministic dictionary. It supports searches in a sorted array in $O(\log N+\delta)$ time in the worst case, matching the lower bounds from [60]. Unlike its randomized counterpart, the deterministic dictionary does not make any assumptions regarding the way in which corruptions are performed.

Dynamic dictionary: We introduce a deterministic dynamic dictionary that significantly improves over the resilient search trees by Finocchi et al. [58]. It supports searches in $O(\log N + \delta)$ in the worst case, and insertions and deletions in $O(\log N + \delta)$ time amortized. Also, it supports range queries in $O(\log N + \delta + k)$ time, where k is the output size.

8.1 Optimal randomized static dictionary

In this section we introduce a simple randomized resilient search algorithm. It searches for a given element in a sorted array using worst case $O(\log \delta)$ random bits and expected time $O(\log N+\delta)$, assuming that corruptions are performed by a non-adaptive adversary. The running time matches the algorithm by Finocchi et al. [56], which, however, uses expected $O(\log N \cdot \log \delta)$ random bits. The main idea of our algorithm is to implicitly divide the sorted input array in 2δ disjoint sorted sequences $S_0, \ldots, S_{2\delta-1}$, each of size at most $\lceil N/2\delta \rceil$. The j'th element of S_i, $S_i[j]$, is the element at position $\mathrm{pos}_i(j) = 2\delta j + i$ in the input array. Intuitively, this divides the input array into $\lceil N/2\delta \rceil$ consecutive *blocks*

of size 2δ, where $S_i[j]$ is the i'th element of the j'th block. Note that, since 2δ disjoint sequences are defined from the input array and at most δ corruptions are possible, at least half of the sorted sequences $S_0, \ldots, S_{2\delta-1}$ do not contain any corrupted elements.

The algorithm generates a random number $k \in \{0, \ldots, 2\delta - 1\}$ and performs an iterative binary search on S_k. We store in safe memory k, the search key e, and the left and right indices, l and r, used by the binary search. The binary search terminates when l and r are adjacent in S_k, and therefore 2δ elements apart in the input array, since $\text{pos}_k(r) - \text{pos}_k(l) = 2\delta$ when $r = l + 1$. If the binary search was not mislead by corruptions, then the location of e is between $\text{pos}_k(l)$ and $\text{pos}_k(r)$ in the input array. To check whether the search was mislead, we perform the following verification procedure. Consider the neighborhoods N_l and N_r, containing the $2\delta + 1$ elements in the input array situated to the left of $\text{pos}_k(l)$ and to the right of $\text{pos}_k(r)$ respectively. We compute the number $s_l = |\{z \in N_l \mid z \leq e\}|$ of elements in N_l that are smaller than e in $O(\delta)$ time by scanning N_l. Similarly, we compute the number s_r of elements in N_r that are larger than e. If $s_l \geq \delta + 1$ and $s_r \geq \delta + 1$, and the search key is not encountered in N_l or N_r, we decide whether it lies in the array or not by scanning the $2\delta - 1$ elements between $\text{pos}_k(l)$ and $\text{pos}_k(r)$. If s_l or s_r is smaller than $\delta + 1$, a corruption has misguided the search. In this case, a new k is randomly selected and the binary search is restarted.

Theorem 8.1 *The randomized dictionary supports searches in $O(\log N + \delta)$ expected time and uses $O(\log \delta)$ expected random bits.*

Proof. We first prove the correctness of the algorithm. Assume that $s_l \geq \delta + 1$ and $e \notin N_l$. Since only δ corruptions are possible, there exists an uncorrupted element in N_l strictly smaller than e. Because the input array is sorted, no uncorrupted elements to the left of $\text{pos}_k(l)$ in the input array are equal to e. By a similar argument, if $s_r \geq \delta + 1$ and $e \notin N_r$, then no uncorrupted elements to the right of $\text{pos}_k(r)$ in the input array are equal to e. If no corrupted elements are encountered during the binary search, all the uncorrupted elements of N_l are smaller than e, and therefore $s_l \geq \delta + 1$. Similarly, we have $s_r \geq \delta + 1$, and the algorithm terminates after scanning the elements between l and r.

We now analyze the running time. Each iteration generates a random number $k \in \{0, \ldots, 2\delta - 1\}$, using $O(\log \delta)$ random bits. The sorted sequences induced by different k's are disjoint, thus at most δ of them may contain corruptions. Since there are 2δ sorted sequences, the probability of selecting a value k that leads to a corruption-free sequence is at least $1/2$, and therefore the expected number of iterations is at most two. Each iteration uses $O(\log N)$ time for the binary search and $O(\delta)$ time for the verification. We conclude that a search uses expected $O(\log \delta)$ random bits and $O(\log N + \delta)$ expected time. □

We note that for each iteration an adaptive adversary can learn about the subsequence S_k on which we perform the binary search by investigating the elements accessed. Subsequently a single corruption suffices to force the search path to end far enough from its correct position such that the verification fails. In this situation, the algorithm performs $O(\delta)$ iterations and therefore

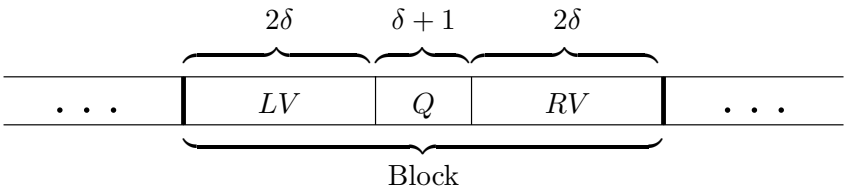


Figure 8.1: The structure of a block. The left and right verification segments, LV and RV, contain 2δ elements each, and the query segment Q contains $\delta + 1$ elements.

$O(\delta(\log N + \delta))$ time regardless of the random choices of subsequences on which to perform the binary search.

We obtain a worst case bound of $O(\log \delta)$ random bits by using a standard derandomization technique. In the i'th iteration we perform the binary search on sequence $S_{h(i)}$, for $h(i) = (r_0 + ir_1 + i^2 r_2 + i^3 r_3) \bmod k$, where k is a prime number with $2\delta \leq k < 4\delta$, and r_i are chosen uniformly at random in $\{0, \ldots, k-1\}$. By construction $h(i)$ is a 4-wise independent hash function [76], which suffices to obtain an expected constant number of iterations for our algorithm [95].

8.2 Optimal static dictionary

In this section we close the gap between lower and upper bounds for deterministic resilient searching algorithms. We present a resilient algorithm that searches for an element in a sorted array in $O(\log N + \delta)$ time in the worst case, which is optimal [60]. It is an improvement of the previously published best deterministic dictionary, which supports searches in $O(\log N + \delta^{1+\varepsilon})$ time [56]. We reuse the idea presented in the design of the randomized algorithm and define disjoint sorted sequences to be used by a binary search algorithm. Similarly to the randomized algorithm, we design a verification procedure to check the result of the binary search. We design the adapted binary search and the verification procedure such that we are guaranteed to advance only one level in the binary search for each corrupted element misleading the search. We count the number of detected corruptions and adjust our algorithm accordingly to ensure that no element is used more than once, excepting a final scan performed only once on two adjacent blocks. The total time used for verification is $O(\delta)$.

We divide the input array into implicit blocks. Each block consists of $5\delta + 1$ consecutive elements of the input and is structured in three segments: the *left verification segment*, LV, consists of the first 2δ elements, the next $\delta + 1$ elements form the *query segment*, Q, and the *right verification segment*, RV, consists of the last 2δ elements of the block, see Figure 8.1. The left and right verification segments, LV and RV, are used only by the verification procedure. The elements in the query segment are used to define the sorted sequences $S_0, \ldots, S_\delta$, similarly to the randomized dictionary previously introduced. The j'th element of sequence S_i, $S_i[j]$, is the i'th element of the query segment of the j'th block, and is located at position $\text{pos}_i(j) = (5\delta + 1)j + 2\delta + i$ in the

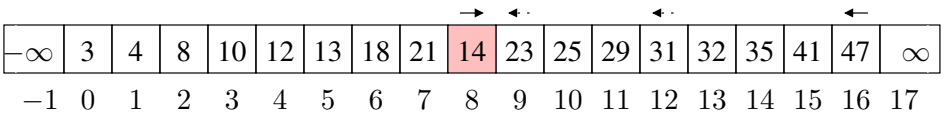


Figure 8.2: Example of binary search on a sequence S_k, for the search key 21. The arrows show the direction of the search. The emphasized element is corrupted.

input array.

We store a value $k \in \{0, \ldots, \delta\}$ in safe memory identifying the sequence S_k on which we currently perform the binary search. Also, k identifies the number of corruptions detected. Whenever we detect a corruption, we change the sequence on which we perform the search by incrementing k. Since there are $\delta + 1$ disjoint sequences, there exists at least one sequence without any corruptions.

Binary search. The binary search is performed on the elements of S_k. Similarly to the randomized algorithm, we store in safe memory the search key, e, and the left and right sequence indices, l and r, used by the binary search. Initially, $l = -1$ is the position of an implicit $-\infty$ element. Similarly, r is the position of an implicit ∞ to the right of the last element. Since each element in S_k belongs to a distinct block, l and r also identify two blocks, B_l and B_r.

Each step in the binary search compares the search key e against the element at position $i = \lfloor (l + r)/2 \rfloor$ in S_k. Assume without loss of generality that this element is smaller than e. We set l to i and decrement r by one. We then compare e with $S_k[r]$. If this element is larger than e, the search continues. Otherwise, if no corruptions have occurred, the position of the search element is in block B_r or B_{r+1} in the input array. When two adjacent elements are identified as in the case just described, or when l and r become adjacent, we invoke a verification procedure on the corresponding blocks. See also Figure 8.2.

The verification procedure determines whether the two adjacent blocks, denoted B_i and B_{i+1}, are correctly identified. If the verification succeeds, the binary search is completed, and all the elements in the two corresponding adjacent blocks, B_i and B_{i+1} are scanned. The search returns true if e is found during the scan, and false otherwise. If the verification fails, the search may have been mislead by corruptions and we backtrack it two steps. To facilitate backtracking, we store two word-sized bit-vectors, d and f in safe memory. The i'th bit of d indicates the direction of the search and the i'th bit of f indicates whether there was a rounding in computing the middle element in the i'th step of the binary search respectively. We can easily compute the values of l and r in the previous step of the binary search by retrieving the relevant bits of d and f. If the verification fails, it detects at least one corruption and therefore k is incremented, thus the search continues on a different sequence S_k.

Verification phase. Verification is performed on two adjacent blocks, B_i and B_{i+1}. It either determines that e lies in B_i or B_{i+1} or detects corruptions. The verification is an iterative algorithm maintaining a value which expresses the confidence that the search key resides in B_i or B_{i+1}. We compute the *left confidence*, c_l, which is a value that quantifies the confidence that e is in B_i or to the right of it. Intuitively, an element in LV_i smaller than e is consistent with the thesis that e is in B_i or to the right of it. However an element in LV_i larger than e is inconsistent. Similarly, we compute the *right confidence*, c_r, to express the confidence that e is in B_{i+1} or to the left of it.

We compute c_l by scanning a sub-interval of the left verification segment, LV_i, of B_i. Similarly, the right confidence is computed by scanning the right verification segment, RV_{i+1}, of B_{i+1}. Initially, we set $c_l = 1$ and $c_r = 1$. We scan LV_i from right to left starting at the element at index $v_l = 2\delta - 2k$ in LV_i. Intuitively, by the choice of v_l we ensure that no element in LV_i is accessed more than once. Similarly, we scan RV_{i+1} from left to right beginning with the element at position $v_r = 2k$. In an iteration we compare $LV_i[v_l]$ and $RV_{i+1}[v_r]$ against e. If $LV_i[v_l] \leq e$, c_l is increased by one, otherwise it is decreased by one and k is increased by one. Similarly, if $RV_{i+1}[v_r] \geq e$, c_r is increased; otherwise, we decrease c_r and increase k. The verification procedure stops when $\min(c_r, c_l)$ equals $\delta - k + 1$ or 0. The verification succeeds in the former case, and fails in the latter. See also Figure 8.3.

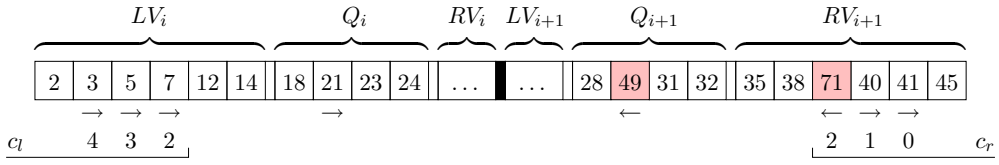


Figure 8.3: A verification step for $\delta = 3$, with $k = 1$ initially. The search key is 45. The verification algorithms stops with $c_r = 0$, reporting failure. The emphasized elements are corrupted.

Theorem 8.2 *The resilient algorithm searches for an element in a sorted array in $O(\log N + \delta)$ time.*

Proof. We first prove that when c_l or c_r decrease during verification, a corruption has been detected. We increase c_l when an element smaller than e is encountered in LV_i, and decrease it otherwise. Intuitively, c_l can been seen as the size of a stack S. When we encounter an element smaller than e, we treat it as if it was pushed, and as if a pop occurred otherwise. Initially, the element g from the query segment of B_i used by the binary search is pushed in S. Since g was used to define the left boundary in the binary search, $g < e$ at that time. Each time an element $LV_i[v] < e$ is popped from the stack, it is *matched* with the current element $LV_i[v_l]$. Since $LV_i[v] < e < LV_i[v_l]$ and $v_l < v$, at least one of $LV_i[v_l]$ and $LV_i[v]$ is corrupted, and therefore each match corresponds to detecting at least one corruption. It follows that if $2t - 1$ elements are scanned on either side during a failed verification, then at least t corruptions are detected.

We now argue that no single corrupted cell is counted twice. A corruption is detected if and only if two elements are matched during verification. Thus it suffices to argue that no element participates in more than one matching. We first analyze corruptions occurring in the left and right verification segments. Since the verification starts at index $2(\delta - k)$ in the left verification segment and k is increased when a corruption is detected, no element is accessed twice, and therefore not matched twice either. A similar argument holds for the right verification segment. Each failed verification increments k, thus no element from a query segment is read more than once. In each step of the binary search both the left and the right indices are updated. Whenever we backtrack the binary search, the last two updates of l and r are reverted. Therefore, if the same block is used in a subsequent verification, a new element from the query segment is read, and this new element is the one initially on the stack. We conclude that elements in the query segments, which are initially placed on the stack, are never matched twice either.

To argue correctness we prove that if a verification is successful, and e is not found in the scan of the two blocks, then no uncorrupted element equal to e exists in the input. If a verification succeeds and e is not found in either block, then $c_l \geq \delta - k + 1$. Since only $\delta - k$ more corruptions are possible, there is at least one uncorrupted element in LV_i smaller than e and thus there can be no uncorrupted elements equal to e to the left of B_i in the input array. By a similar argument, if $c_r \geq \delta - k + 1$, then all uncorrupted elements to the right of B_{i+1} in the input array are larger than e.

We now analyze the running time. We charge each backtracking of the binary search to the verification procedure that triggered it. Therefore, the total time of the algorithm is $O(\log N)$ plus the time required by verifications. To bound the time used for all verification steps we use the fact that if $O(f)$ time is used for a verification step, then $\Omega(f)$ corruptions are detected or the algorithm ends. At most $O(\delta)$ time is used in the last verification for scanning the two blocks. □

8.3 Dynamic dictionary

In this section we describe a linear space resilient deterministic dynamic dictionary supporting searches in optimal $O(\log N + \delta)$ worst case time and range queries in optimal $O(\log N + \delta + k)$ worst case time, where k is the size of the output. The amortized update cost is $O(\log N + \delta)$.

Structure. The sorted sequence of elements is partitioned into a sequence of *leaf structures*, each storing $\Theta(\delta \log N)$ elements. For each leaf structure we select a guiding element, and we place these $O(N/(\delta \log N))$ guiding elements in the leaves of a reliably stored binary search tree. Each guiding element is chosen such that it is larger than all uncorrupted elements in the corresponding leaf structure.

For this reliable *top tree* T, we use the (non-resilient) binary search tree in [35], which consists of $h = \log |T| + O(1)$ levels when containing $|T|$ elements.

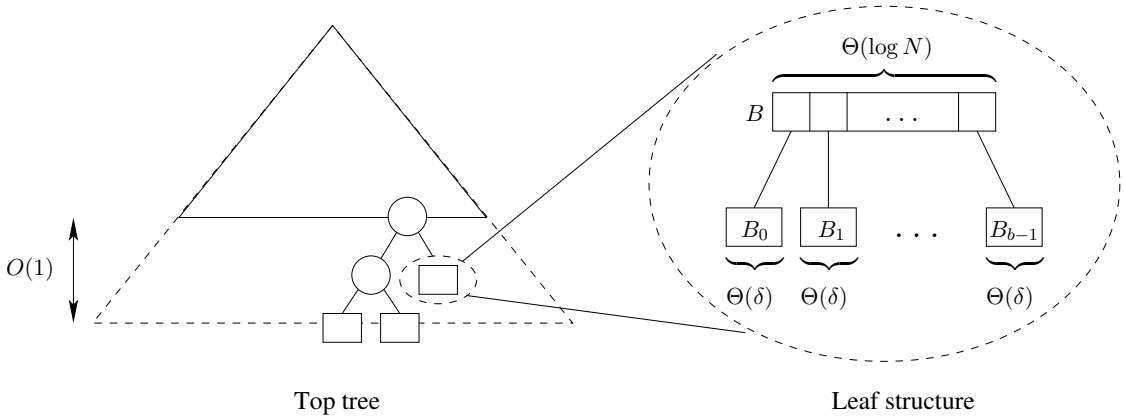


Figure 8.4: The structure of the dynamic dictionary.

In the full version [36] it is shown that the tree can be maintained such that the first $h-2$ levels are complete. We lay the tree in memory in left-to-right breadth first order, as specified in [35]. It uses linear space, and an update costs amortized $O(\log^2 |T|)$ time. A global rebuilding is performed when $|T|$ changes by a constant factor.

All the elements and pointers in the top tree are stored reliably, using replication. Since a reliable value takes $O(\delta)$ space, $O(\delta|T|)$ space is used for the entire structure. The time used for storing and retrieving a reliable value is $O(\delta)$, and therefore the additional work required to handle the reliably stored values increases the amortized update cost to $O(\delta \log^2 |T|)$ time.

The leaf structure consists of a top bucket B and b buckets, $B_0, \ldots, B_{b-1}$, where $\log N \leq b \leq 4 \log N$. Each bucket B_i contains between δ and 6δ input elements, stored consecutively in an array of size 6δ, and uncorrupted elements in B_i are smaller than uncorrupted elements in B_{i+1}. For each bucket B_i, the top bucket B associates a guiding element larger than all elements in B_i, a pointer to B_i, and the size of B_i, all stored reliably. Since storing a value reliably uses $O(\delta)$ space, the total space used by the top bucket is $O(\delta \log N)$. The guiding elements of B are stored as a sorted array to enable fast searches using the deterministic resilient search algorithm from Section 8.2.

Lemma 8.1 *The dynamic dictionary uses $O(N)$ space to store N elements.*

Proof. Since a leaf structure stores $\Theta(\delta \log N)$ input elements, the top tree contains $O(N/(\delta \log N))$ nodes, using $O(\delta|T|) = O(\delta N/(\delta \log N)) = o(N)$ space. Each of the $O(N/(\delta \log N))$ leaf structures uses $O(\delta \log N)$ space and therefore the total space used for leaf structures is $O(N)$. □

Searching. The search operation consists of two steps. It first locates a leaf in the top tree T, and then searches the corresponding leaf structure. Let h denote the height of T. If $h \leq 3$, we perform a standard tree search from the root of T using the reliably stored guiding elements and pointers. Otherwise, we

locate two internal nodes, v_1 and v_2, with guiding elements g_1 and g_2, such that $g_1 < e \leq g_2$, where e is the search key. Since $h-2$ is the last complete level of T, level $\ell = h-3$ is complete and contains only internal nodes. The breadth first layout of T ensures that elements of level ℓ are stored consecutively in memory. The search operation locates v_1 and v_2 using the deterministic resilient search algorithm from Section 8.2 on the array defined by level ℓ. The search only considers the $2\delta + 1$ cells in each node containing guiding elements and ignores memory used for auxiliary information, e.g. sizes and pointers. Although they are stored using replication, the guiding elements are considered as $2\delta+1$ regular elements in the search. Since the space used by the auxiliary information is the same for all nodes, these gaps in the memory layout of level ℓ are easily excluded from the search. We modify the resilient searching algorithm previously introduced such that it reports two consecutive blocks with the property that if the search key is in the structure, it is contained in one of them. The reported two blocks, each of size $5\delta + 1$, span $O(1)$ nodes of level ℓ and the guiding elements of these are queried reliably to locate v_1 and v_2. The appropriate leaf can be in either of the subtrees rooted at v_1 and v_2, and we perform a standard tree search in both using the reliably stored guiding elements and pointers. Searching for an element in a leaf structure is performed by using the resilient search algorithm from Section 8.2 on the top bucket, B, similar to the way v_1 and v_2 were found in T. The corresponding reliably stored pointer is then followed to a bucket B_i, which is scanned.

Range queries can be performed by scanning the level ℓ, starting at v, and reporting relevant elements in the leaves below it.

Lemma 8.2 *The search operation of the dynamic dictionary uses $O(\log N + \delta)$ worst case time. A range query reporting k elements is performed in worst case $O(\log N + \delta + k)$ time.*

Proof. The initial search in the top tree takes $O(\log N + \delta)$ worst case time by Theorem 8.2. Traversing the $O(1)$ levels to a leaf takes time $O(\delta)$. Searching in the top bucket of the leaf structures uses $O(\log\log N + \delta)$ time, again using Theorem 8.2. The final scan of a bucket takes time $O(\delta)$.

In a range query, the elements reported in any leaf completely contained in the query range pay for the $O(\delta \log N)$ time used for going through the bottom part of the top tree and scanning the top bucket. The search pays for the rightmost traversed leaf. □

Updates. Efficiently updating the structure is performed using standard bucketing techniques. To insert an element into the dictionary, we first perform a search to locate the appropriate bucket B_i in a leaf structure, and then the element is appended to B_i and the size of B_i in the top bucket is updated. When the size of B_i increases to 6δ, we split it into two buckets, B_s and B_g, of almost equal sizes. We compute a guiding element that splits B_i in $O(\delta^2)$ time by repeatedly scanning B_i and extracting the minimum element. The element m returned by the last iteration is kept in safe memory. In each iteration, we select a new m which is the minimum element in B_i larger than the current

m. Since at most δ corruptions can occur, B_i contains at least 2δ uncorrupted elements smaller than m and 2δ uncorrupted elements larger, after $|B_i|/2 = 3\delta$ iterations. The elements from B_i smaller than m are stored in B_s, and the remaining ones are stored in B_g. The guiding element for B_s is m, while B_g preserves the guiding element of B_i. The new split element is reliably inserted in the top bucket using an insertion sort step, by scanning and shifting the elements in B from right to left, and placing the new element at its appropriate position. Similarly, when the size of the top bucket becomes $4 \log N$, it is split in two new leaf structures. The first leaf structure consists of the first $2 \log N$ bottom buckets, and the second leaf structure contains the rest. The second leaf structure is associated with the original guiding element, and the guiding element of the new leaf structure is the the last guiding element in its top bucket. This new guiding element is inserted into the top tree.

Deletions are handled similarly by first searching for the element and then removing it from the appropriate bucket. When an element is deleted from a bucket, we ensure that the elements in the affected bucket are stored consecutively by swapping the deleted element with the last element. If the affected bucket holds fewer than δ elements after the deletion, it is merged with a neighboring bucket. If the resulting bucket contains more than 6δ elements, it is split as described above. If the top bucket contains less than $\log N$ guiding elements, it is merged with a neighboring leaf structure which is found using a search. Following this, the original leaf is deleted from the top tree.

Lemma 8.3 *The insert and delete operations of the dynamic dictionary take $O(\log N + \delta)$ amortized time each.*

Proof. An update in the top tree takes $O(\delta \log^2 N)$ time and requires $\Omega(\delta \log N)$ updates in the leaf structures. Thus each update costs amortized $O(\log N)$ time for operations in the top tree. Splitting and merging a bucket of a leaf structure takes time $O(\delta \log N)$ for updates to the top bucket and $O(\delta^2)$ time for computing a split element for a bucket. A bucket is split or merged every $\Omega(\delta)$ operations resulting in an amortized update cost of $O(\log N + \delta)$. Appending or removing a single element to a bucket takes worst case time $O(\delta)$ for updating the size. Adding the $O(\log N + \delta)$ cost of the initial search concludes the proof. □

Theorem 8.3 *The resilient dynamic dictionary structure uses $O(N)$ space while supporting searches in $O(\log N + \delta)$ time worst case with an amortized update cost of $O(\log N + \delta)$. Range queries with an output size of k is performed in worst case $O(\log N + \delta + k)$ time.*

Chapter 9

I/O-Efficient Resilient Algorithms

In this chapter we study algorithms and data structures for external memory in the presence of an unreliable internal and external memory.

Memory corruptions are of particular concern for applications dealing with massive amounts of data since such applications typically run for a very long time, and are thus more likely to encounter memory cells containing corrupted data.

Current resilient algorithms do not scale past the internal memory of a computer and thus, it is currently not possible to work with large sets of data I/O-efficiently while maintaining resiliency to memory corruptions. Since both models become increasingly interesting as the amount of data increases, it is natural to consider whether it is possible to achieve resilient algorithms that use the disk optimally. Very recently, this was also proposed as an interesting direction of research by Finocchi et al. [57, 61].

9.1 Our Contribution

The work in this chapter combines the faulty memory RAM and the external memory model in the natural way. The model has three levels of memory: a disk, an internal memory of size M, and $O(1)$ CPU registers. All computation takes place on elements placed in the registers. The content of any cell on disk or in internal memory can be corrupted at any time, but at most δ corruptions can occur. Moving elements between memory and registers takes constant time and transferring a chunk of B consecutive elements between disk and memory costs one *I/O*. Transfers between the different levels are atomic, no data can be corrupted while it is being copied. Correctness of an algorithm is proved with the assumption that an adaptive adversary may perform corruptions during execution. For randomized algorithms we assume that the random bits are hidden from the adversary. In two natural variants of our model it is assumed that corruptions take place only on disk, or only in memory.

In this chapter, we present I/O-efficient solutions to all problems that, to the best of our knowledge, have previously been considered in the faulty memory RAM. It is not clear that resilient algorithms can be optimal both in time and in I/O-complexity. Most techniques for designing I/O-efficient algorithms naturally

	I/O Complexity	Assumptions	I/O Tolerance (max δ)	Time Tolerance (max δ)
Det. Dict.	$O\left(\frac{1}{\varepsilon}\log_B N + \frac{\delta}{B^{1-\varepsilon}}\right)$	$\frac{1}{\log B} < \varepsilon < 1$	$O(B^{1-\varepsilon}\log_B N)$	$O(\log N)$
Ran. Dict.	$O(\log_B N + \frac{\delta}{B})$	Memory Safe	$O(B\log_B N)$	$O(\log N)$
P. Queue	$O(\frac{1}{1-\varepsilon}\frac{1}{B}\log_{M/B}(N/M))$	$\delta \leq M^{\varepsilon},\ \varepsilon < 1$	$O(M^{\varepsilon})$	$O(\log N)$
Sorting	$O(\frac{1}{1-\varepsilon}\mathrm{Sort}(N))$	$\delta \leq M^{\varepsilon},\ \varepsilon < 1$	$O(M^{\varepsilon})$	$O(\sqrt{N\log N})$

Table 9.1: The first column shows the I/O upper bounds presented in this chapter with the assumptions shown in the second column. The third and fourth column shows how many corruptions the algorithms can tolerate while still matching the optimal algorithms in the I/O and comparison model respectively. Note that the restriction imposed by the time bounds are orders of magnitude stronger than the ones imposed by the I/O bounds for realistic values of M, B and N.

try to arrange data on disk such that few blocks need to be read in order to extract the information needed, whereas resilient algorithms try to put little emphasis on individual, potentially corrupted, memory cells.

In the I/O model, a comparison-based dictionary with optimal queries can be achieved with a B-tree [29], which supports queries and updates in $O(\log_B N)$ I/Os. It is also known that any resilient comparison-based search algorithm must examine $\Omega(\log N + \delta)$ memory cells [61]. Combining these we get a simple lower bound of $\Omega\left(\log_B N + \frac{\delta}{B}\right)$ I/Os, and $\Omega(\log N + \delta)$ time for a resilient comparison-based I/O-efficient static ditionary. In Section 9.2 we prove a stronger lower bound of $\Omega\left(\frac{1}{\varepsilon}\log_B N + \frac{\delta}{B^{1-\varepsilon}}\right)$ I/Os for a search, for all $\log_B N \leq \delta \leq B\log N$ and ε given by the equation $\delta = \frac{B^{1-\varepsilon}}{\varepsilon}\log_B N$. In the case where $\delta = \Theta(\frac{B}{\log B}\log_{\log B} N)$, setting $\varepsilon = \frac{\log\log B}{\log B}$ gives a lower bound of $\Omega(\log_{\log B} N + \frac{\delta}{B}\log B)$ which is $\omega(\log_B N + \frac{\delta}{B})$. We come to the interesting conclusion that no deterministic resilient dictionary can obtain an I/O bound of $O(\log_B N + \frac{\delta}{B})$ without some assumptions on δ. The lower bound is valid for randomized algorithms as long as the internal memory is unreliable. For deterministic algorithms, the lower bound also holds if the internal memory is reliable and corruptions only occur on disk.

In Section 9.3 we construct a resilient dictionary supporting searches using expected $O\left(\log_B N + \frac{\delta}{B}\right)$ I/Os and $O(\log N + \delta)$ time for any δ if corruptions occur exclusively on disk. Thus, we have an interesting separation between the I/O complexity of resilient randomized and resilient deterministic searching algorithms. This also proves that it is important whether it is the disk or the internal memory that is unreliable.

In Section 9.4 we present an optimal resilient static dictionary supporting queries in $O\left(\frac{1}{c}\log_B N + \frac{\alpha}{B^{1-c}} + \frac{\delta}{B}\right)$ I/Os and $O(\log N + \delta)$ time when $\log N \leq \delta \leq B\log N$ and $\frac{1}{\log B} \leq c \leq 1$. Queries use $O(\log_B N + \frac{\delta}{B})$ I/Os and $O(\log n + \delta)$ time for $\delta \leq \log N$ and $\delta > B\log N$. Additionally, in Section 9.5, we construct randomized and deterministic dynamic dictionaries with optimal query bounds using our static dictionaries.

Finally, in Section 9.6 we describe a resilient multi-way merging algorithm. We use this algorithm to design an optimal resilient sorting algorithm using $O(\frac{1}{1-\varepsilon}\text{Sort}(N))$ I/Os and $O(N \log N + \alpha\delta)$ time under the assumption that $\delta \leq M^\varepsilon$, for $0 \leq \varepsilon < 1$. The multi-way merging algorithm is also used to design a resilient priority queue for the case $\delta \leq M^\varepsilon$, where $0 \leq \varepsilon < 1$. Our priority queue supports INSERT and DELETEMIN in optimal $O(\frac{1}{1-\varepsilon}(1/B)\log_{M/B}(N/M))$ I/Os amortized, matching the bounds for non-resilient external memory priority queues. The amortized time bound for both operations is $O(\log N + \delta)$ matching the time bounds of the optimal resilient priority queue presented in Chapter 7.

Table 9.1 shows an overview of the upper bounds in this chapter. The two last columns in the table shows how many corruptions our algorithms can tolerate while still achieving optimal bounds in the I/O model and comparison model respectively. Note that the bounds on δ required to get optimal time are orders of magnitude smaller than the bounds required to get optimal I/O performance for realistic values of N, M and B. We conclude that it is possible, under realistic assumptions, to get resilient algorithms that are optimal in both the I/O-model and the comparison model without restricting δ more than what was required to obtain optimal time bounds in the faulty memory RAM.

9.2 Lower Bound for Dictionaries

Any resilient searching algorithm must examine $\Omega(\log N + \delta)$ memory cells in the comparison model [61]. The $\Omega(\log N)$ term follows from the comparison model lower bound for searching. It is well-known that comparison-based searching in the I/O model requires expected $\Omega(\log_B N)$ I/Os. Since any resilient searching algorithm must read at least $\Omega(\delta)$ elements to ensure at least some non-corrupted information is the basis for the output, we get the following trivial lower bound.

Lemma 9.1 *For any comparison-based randomized resilient dictionary the average-case expected search cost is* $\Omega\left(\log_B N + \frac{\delta}{B}\right)$ *I/Os.*

In this section we prove a stronger lower bound on the worst-case number of I/Os required for any deterministic resilient static dictionary in the comparison model. We do not make any assumptions on the data structure used by the dictionary, nor on the space it uses. Additionally, we do not bound the amount of computation time used in a query and we assume that the total order of all elements stored in the dictionary are known by the algorithm initially. During the search for an element e, an algorithm gains information by performing block I/Os, each I/O reading B elements from disk. Before a block of B elements is read into memory the adversary can corrupt the elements in the block. The adversary is allowed to corrupt up to δ elements during the query operation, but does not have to reveal when it chooses to do so. Also, the adversary adaptively decides what the rank of the search element has among the N dictionary elements. Of course, the rank must be consistent with the previous uncorrupted elements read by the algorithm.

Theorem 9.1 *Given N and δ, any deterministic resilient static dictionary requires worst-case $\Omega\left(\frac{1}{\varepsilon}\log_B N\right)$ I/Os for a search, for all ε where $\frac{1}{\log B} \leq \varepsilon \leq 1$ and $\delta \geq \frac{1}{\varepsilon}B^{1-\varepsilon}\log_B N$.*

Proof. We design an adversary that uses corruptions to control how much information any correct query algorithm gains from each block transfer.

Let ε be a constant such that $\frac{1}{\log B} \leq \varepsilon \leq 1$. The strategy of the adversary is as follows. For each I/O, the adversary narrows the *candidate interval* where e can be contained in by a factor B^ε. Initially, the candidate interval consists of all N elements. For each I/O, the adversary implicitly divides the sorted set of elements in the candidate interval into B^ε slabs of equal size. Since the search algorithm only reads B elements in an I/O, there must be at least one slab containing at most $B^{1-\varepsilon}$ of these elements. The adversary corrupts these elements, such that they do not reveal any information, and decides that the search element resides in this slab. The remaining elements transferred are not corrupted and are automatically consistent with the interval chosen for e. The game is then played recursively on the elements of the selected slab, until all elements in the final candidate interval have been examined.

For each I/O, the candidate interval decreases by a factor B^ε. The algorithm has no information regarding elements in the slab except for the corrupted elements from the I/Os performed so far. After k I/Os the candidate interval has size $\frac{N}{(B^\varepsilon)^k}$ and the adversary has introduced at most $kB^{1-\varepsilon}$ corruptions. The game continues as long as there is at least one uncorrupted element among the elements remaining in the candidate interval, which the adversary can choose as the search element. All corrupted elements may reside in the current candidate interval, and the game ends when the size of the candidate interval, $\frac{N}{(B^\varepsilon)^k}$, becomes smaller than or equal to the total number of introduced corruptions, $kB^{1-\varepsilon}$. It follows that at least $\Omega\left(\log_{B^\varepsilon}\frac{N}{B^{1-\varepsilon}}\right) = \Omega\left(\frac{1}{\varepsilon}\log_B N\right)$ I/Os are required. The adversary introduces at most $B^{1-\varepsilon}$ corruptions in each step. If ε satisfies $\frac{1}{\varepsilon}B^{1-\varepsilon}\log_B N \leq \delta$, then the adversary can play the game for at least $\frac{1}{\varepsilon}\log_B N$ rounds and the theorem follows. □

For deterministic algorithms it does not matter whether elements can be corrupted on disk or in internal memory. Since the adversary is adaptive it knows which block of elements an algorithm will read into internal memory next, and may choose to corrupt the elements on disk just before they are loaded into memory, or corrupt the elements in internal memory just after they have been written there. In randomized algorithms where the adversary does not know the algorithm's random choices it cannot determine which block of elements will be fetched from disk before the transfer has started. Therefore, the adversary can follow the strategy above only if it can corrupt elements in internal memory.

By setting $\delta = \frac{1}{\varepsilon}B^{1-\varepsilon}\log_B N$ in Theorem 9.1, we get the following corollary.

Corollary 9.1 *Any deterministic resilient static dictionary requires worst-case $\Omega(\frac{1}{\varepsilon}\log_B N) = \Omega(\frac{\delta}{B^{1-\varepsilon}})$ I/Os for a search, where $\delta \in [\log_B N, B\log N]$, and ε given by $\delta = \frac{1}{\varepsilon}B^{1-\varepsilon}\log_B N$.*

The trivial I/O lower bound for a resilient searching algorithm is $\Omega\left(\log_B N + \frac{\delta}{B}\right)$. Setting $\varepsilon = \frac{\log\log B}{\log B}$ in Theorem 9.1 shows that this is not optimal.

Corollary 9.2 *For $\delta = \frac{B}{\log B}\log_{\log B} N$ any deterministic resilient static dictionary requires worst-case $\Omega(\frac{\log B}{\log\log B}(\log_B N + \frac{\delta}{B}))$ I/Os for a search.*

9.3 Randomized Static Dictionary

In this section we describe a simple I/O-efficient randomized static dictionary, that is resilient to corruptions on the disk. Corruptions in memory are not allowed, thus the adversarial lower bound in Theorem 9.1 does not apply. The dictionary supports queries using expected $O\left(\log_B N + \frac{\delta}{B}\right)$ I/Os and $O(\log N + \delta)$ time. The algorithm is similar to the randomized binary search algorithm in [61]. Remember that, if only elements on disk can be corrupted, the lower bound from Theorem 9.1 also holds for deterministic algorithms. This means that deterministic and randomized algorithms are separated by the result in this section.

The idea is to store the N elements in the dictionary in sorted order in an array S and to build 2δ B-trees [29], denoted $T_1, \ldots, T_{2\delta}$, of size $\lfloor \frac{N}{2\delta} \rfloor$. The i'th B-tree T_i stores the $2\delta j + i$'th element in S for $j = 0, \ldots, \lfloor \frac{N}{2\delta} \rfloor - 1$. Each node in each tree is represented by a faithfully ordered array of $\Theta(B)$ search keys. The nodes of the B-tree are laid out in left to right breadth first order, to avoid storing pointers, i.e. the c'th child of the node at index k has index $Bk + c - (B-1)$.

The search for an element e proceeds as follows. A random number $r_1 \in \{1, \ldots, 2\delta\}$ is generated, and the root block of T_{r_1} is fetched into the internal memory. In this block, a binary search is performed among the search keys resulting in an index, i, of the child where the search should continue. A new random number $r_2 \in \{1, \ldots, 2\delta\}$ is generated, and the i'th child of the root in tree T_{r_2} is fetched and the algorithm proceeds iteratively as above. The search terminates when a leaf is reached and two keys $S[2\delta j + i]$ and $S[2\delta(j+1)+i]$ have been identified such that $S[2\delta j + i] \leq e < S[2\delta(j+1) + i]$. If the binary search was not mislead by corruptions of elements, then e is located in the subarray $S[2\delta j + i, \ldots, 2\delta(j+1) + i]$. To check whether the search was mislead, the following *verification procedure* is performed. Consider the neighborhoods $L = S[2\delta(j-1)+i-1, \ldots, 2\delta j+i-1]$ and $R = S[2\delta(j+1)+i+1, \ldots, 2\delta(j+2)+i+1]$, containing the $2\delta + 1$ elements in S situated to the left of $S[2\delta j + i]$ and to the right of $S[2\delta(j+1) + i]$ respectively. The number $s_L = |\{z \in L \mid z \leq e\}|$ of elements in L that are smaller than e is computed by scanning L. Similarly, the number s_R of elements in R that are larger than e is computed. If $s_L \geq \delta + 1$ and $s_R \geq \delta + 1$, and the search key is not encountered in L or R, we decide whether it is contained in the dictionary or not by scanning the subarray $S[2\delta j, \ldots, 2\delta(j+1)]$. If s_L or s_R is smaller than $\delta + 1$, at least one corruption has misguided the search. In this case, the search algorithm is restarted.

Theorem 9.2 *The data structure described is a linear space randomized dictionary supporting searches in expected $O\left(\log_B N + \frac{\delta}{B}\right)$ I/Os and $O(\log N + \delta)$ time assuming that memory cells are incorruptible and block transfers are atomic.*

Proof. The proof roughly follows the proof of [56]. First, we prove correctness

of the algorithm. Assume that $s_L \geq \delta + 1$ and $e \notin L$. Since only δ corruptions are possible, there exists at least one uncorrupted element in L smaller than e. Because S is sorted, no uncorrupted elements to the left of $S[2\delta j]$ in S can be equal to e. By a similar argument, if $s_R \geq \delta + 1$ and $e \notin R$, then no uncorrupted elements to the right of $S[2\delta(j+1)]$ in S are equal to e. If no corruptions are encountered during the B-tree search, all the uncorrupted elements of L are less than or equal to e, and therefore $s_L \geq \delta + 1$. Similarly, we have $s_R \geq \delta + 1$, and the algorithm terminates after scanning $S[2\delta j, \ldots, 2\delta(j+1)]$.

In each step, the algorithm chooses a random B-tree among the 2δ B-trees, and loads the next node from the randomly chosen B-tree to guide the search. The adversary cannot know which B-tree that is used in any iteration, since he does not know the random bits and block reads are atomic. Let β_i be the number of nodes containing corruptions on level i in each of the B-trees. Then, the probability that the node used in iteration i contains corruptions is at most $\frac{\beta_i}{2\delta}$ and thus the probability that the algorithm does not use any blocks containing corrupted elements is $\prod_{i=1}^{\log_B N} \left(1 - \frac{\beta_i}{2\delta}\right)$ which is at least $\frac{1}{2}$ [C3]. It follows that the expected number of restarts caused by misguided searches due to faults is at most 2. □

If memory cells were corruptible the atomic transfer assumption would be of little use. The adversary could simply corrupt the elements in the internal memory after the block transfer completes, decreasing the benefit of the randomization.

9.4 Optimal Deterministic Static Dictionary

In this section we present a linear space deterministic resilient static dictionary. Let c be a constant such that $\frac{1}{\log B} \leq c \leq 1$. The dictionary supports queries in $O\left(\frac{1}{c}\log_B N + \frac{\alpha}{B^{1-c}} + \frac{\delta}{B}\right)$ I/Os and $O(\log N + \delta)$ time. In Section 9.2 we proved a lower bound on the I/O complexity of resilient dictionaries, and by choosing c in the above bound to minimize the expression for $\alpha = \delta$, this bound matches the lower bound. Thus, this dictionary is optimal.

Our data structure is based on the B-tree and the resilient binary search algorithm from Chapter 8. In a standard B-tree search one corrupted element can misguide the algorithm, forcing at least one I/O in the *wrong* part of the tree. To circumvent this problem, each guiding element in each internal node is determined by taking majority of B^{1-c} copies. This gives a trade-off between the number of corruptions required to misguide a search, and the fan-out of the tree, which becomes B^c. Additionally, each node stores $2\delta + 1$ copies of the minimum and maximum element contained in the subtree, such that the search algorithm can reliably check whether it is on the correct path in each step. We ensure that the query algorithm avoids reading the same corrupted element twice by ensuring that any element is read at most once. The exact layout of the tree and the details of the search operation are as follows.

Structure: Let S be the set of elements contained in the dictionary and let N denote the size of S. The dictionary is a B^c-ary search tree T built on $\frac{N}{\delta}$

leaves. The elements of S are distributed to the leaves in faithful order such that each leaf contains δ elements. Each leaf is represented by a *guiding element* which is smaller than the smallest uncorrupted element in the leaf and larger than the largest uncorrupted element in the preceding leaf. The top tree is built using these guiding elements. The tree is stored in a breadth-first left-to-right layout on disk, such that no pointers are required.

Each internal node u in T stores three types of elements; guiding elements, minimum elements, and maximum elements, stored consecutively on disk. The guiding elements are stored in $\lceil (2\delta+1)/B^{1-c} \rceil$ identical blocks. Each block contains B^{1-c} copies of each of the B^c guiding elements in sorted order such that the first B^{1-c} elements are copies of the smallest guiding element. This means that each guiding element is stored $2\delta+1$ times and can be retrieved reliably. The minimum elements are $2\delta+1$ copies of the guiding element for the leftmost leaf in the subtree defined by u, stored consecutively in $\lceil \frac{2\delta+1}{B} \rceil$ blocks. Similarly the maximum elements are $2\delta+1$ copies of the guiding element for the leaf following the rightmost leaf in the subtree defined by u, stored consecutively in $\lceil \frac{2\delta+1}{B} \rceil$ blocks. Additionally, minimum and maximum elements are stored with each leaf. The minimum are 4δ copies of the guiding element representing the leaf, stored consecutively in $\frac{4\delta}{B}$ blocks, and the maximum elements are 4δ copies of the guiding element representing the subsequent leaf, stored consecutively in $\frac{4\delta}{B}$ blocks. These are used to verify that the algorithm found the only leaf that may store an uncorrupted element matching the search element.

Query: A query operation for an element q, uses an index k that indicates how many chunks of B^{1-c} elements the algorithm has discarded during the search, initially $k=0$. Intuitively, a chunk is discarded if the algorithm detects that $\Omega(B^{1-c})$ of its elements are corrupted. The query operation traverses the tree top-down, storing in safe memory the index k, and $O(1)$ extra variables required to traverse the tree using the knowledge of its layout on disk. In an internal node u, the algorithm starts by checking whether u is on the correct path in the tree using the copies of the minimum and maximum elements stored in u. This is done by scanning B^{1-c} of the $2\delta+1$ copies of the minimum element starting with the kB^{1-c}'th copy, counting how many of these that are larger than q. If $B^{1-c}/2$ or more copies of the minimum element are larger than q the block is discarded by incrementing k and the search is restarted (backtracked) at node v, where $v=u$ if u is root of the tree and the parent of u otherwise. The maximum elements are checked similarly. If the algorithm backtracks, k is increased ensuring that the same element is never read more than once.

If the checks succeed the k'th block storing copies of the B^c guiding elements of u is scanned from left to right. The majority value of each of the B^{1-c} copies of each guiding element is extracted in sorted order using the majority algorithm [33] and compared to q, until a retrieved guiding element larger than q is found or the entire block is read. The traversal then continues to the corresponding child. If any invocation of the majority algorithm fails to select a value, or two fetched guiding elements are out of order, the block is discarded as above by increasing k and backtracking the search to the parent node.

Upon reaching a leaf, the algorithm verifies whether the search found the correct leaf. This is achieved by running a variant of the verification procedure

designed for the same purpose in Chapter 8. Counters c_l and c_r, which are initially 1, are stored in safe memory. Then the copies of the minimum and maximum element are scanned in chunks of B^{1-c} elements, starting from the $2kB^{1-c}$'th element. If the majority of elements in a chunk of B^{1-c} copies of the minimum element are smaller than the search element, c_l is increased by 1. Otherwise, c_l is decreased and k increased by one. The copies of maximum elements are treated similarly. Note that every decrement of c_l or c_r signals that at least $\frac{B^{1-c}}{2}$ corruptions have been found. Thus, c_l represents the number of chunks scanned that has not yet been contradicted, where the majority of copies indicates that the search element is in the current leaf or in leafs to the right. Similar for c_r. If $\min\{c_l, c_r\}$ reaches 0, we backtrack to the parent of the leaf as above. If $\min\{c_l, c_r\}\frac{B^{1-c}}{2}$ gets larger than $\delta - k(\frac{B^{1-c}}{2}) + 1$ the verification succeeds. The algorithm finishes by scanning the δ elements stored in the leaf, returning whether it finds q or not.

Lemma 9.2 *The data structure is a linear space resilient dictionary supporting queries in* $O\left(\frac{1}{c}\log_B N + \frac{\alpha}{B^{1-c}} + \frac{\delta}{B}\right)$ *I/Os, for any* $1/\log B \leq c \leq 1$.

Proof. A value is only erroneously retrieved when at least $\frac{B^{1-c}}{2}$ of the copies used to determine it are corrupted. If k is incremented because of a failed check in an internal node u, then at least one value in the parent of u was erroneously retrieved or at least $\frac{B^{1-c}}{2}$ copies of the minimum or maximum value read at u were corrupted. If k is increased during a verification of a leaf, the majority of elements in one chunk of B^{1-c} copies of the minimum (maximum) element was larger (smaller) than the search element and in another the majority was smaller (larger) than the search element. Therefore, k is only increased when $\frac{B^{1-c}}{2}$ corruptions are detected. Since k increases before any backtracking is performed, the algorithm never reads the same element twice, proving that all corruptions counted are distinct.

The algorithm finishes when $\min\{c_l, c_r\}\frac{B^{1-c}}{2} \geq \delta - k\frac{B}{2} + 1$ during the verification of a leaf. Since the adversary has at most $\delta - k\frac{B}{2}$ corruptions left, in at least one chunk of B^{1-c} copies of the minimum element read during the verification, more than half of the elements are uncorrupted. Since the majority of copies in this block are smaller than the search element, no uncorrupted elements matching the search key can be in leafs to the left. Similar for the blocks containing copies of the maximum, proving that the correct leaf is found.

To bound the I/O complexity, we count how many nodes of the tree the algorithm visits, that are **not** on the correct root to leaf path. If a search is guided in the wrong direction (away from the correct root to leaf path), the majority of B^{1-c} copies of a guiding element in the relevant block are corrupted. For each additional step performed by the algorithm after a *wrong* turn, either the minimum or the maximum chunk scanned must contain $\frac{B^{1-c}}{2}$ corruptions.

In the verification step, each time a minimum and a maximum block is scanned either k or $\min\{c_l, c_r\}$ is increased. Therefore, if $2t - 1$ I/Os were performed by a failed verification k increased by t, meaning that $\frac{tB^{1-c}}{2}$ corruptions were detected. We conclude that the algorithm uses $O(\log_{B^c} N + \frac{\alpha}{B^{1-c}} + \frac{\delta}{B}) = O(\frac{1}{c}\log_B N + \frac{\alpha}{B^{1-c}} + \frac{\delta}{B})$ I/Os. □

To obtain optimal time bounds for the dictionary, we use the resilient binary search algorithm from Chapter 8 on each block, instead of scanning it. If more than $\frac{B^{1-c}}{2}$ corruptions are discovered during the search of a block, it is discarded as above. Otherwise, $\frac{B^{1-c}}{2}$ supporting elements are found on both sides of an element, and the algorithm continues to the corresponding child as before. This reduces the time used per node to $O(\log B + B^{1-c})$. Verification takes $O(\delta)$ time in total.

Lemma 9.3 *For any $\frac{1}{\log B} \leq c \leq 1$, queries use $O((B^{1-c} + \log B)(\frac{1}{c}\log_B N + \frac{\alpha}{B^{1-c}}) + \delta)$ time.*

Corollary 9.3 *If $\delta > B \log N$, queries use $O(\frac{\delta}{B})$ I/Os and $O(\delta)$ time.*

Proof. Follows from Lemma 9.2 and 9.3 by setting $c = \frac{1}{\log B}$ in Lemma 9.2 and 9.3, i.e. T is a binary tree. □

Corollary 9.4 *If $\delta < \log N$, queries use $O(\log_B N)$ I/Os and $O(\log N)$ time.*

Proof. Follows from Lemma 9.2 and 9.3 by setting $c = 1 - \frac{\log\log B}{\log B}$ i.e. T has degree $\frac{B}{\log B}$. □

Corollary 9.5 *If $\log N \leq \delta \leq B \log N$ for any $\frac{1}{\log B} \leq c \leq 1$, queries use $O(\frac{1}{c}\log_B N + \frac{\alpha}{B^{1-c}} + \frac{\delta}{B})$ I/Os and $O(\log N + \delta)$ time.*

Proof. Follows from Lemma 9.2 and 9.3 by selecting $c \in [\frac{1}{\log B}, 1 - \frac{\log\log B}{\log B}]$ such that $\frac{1}{c}\log_B N = \frac{\delta}{B^{1-c}}$. □

9.5 Dynamic Dictionaries

In this section we present a dynamic I/O-efficient resilient dictionary based on the techniques in used for the dynamic dictionary in Chapter 8 and the static dictionary presented in Section 9.4. The dynamic dictionary supports queries and updates in $O(\frac{1}{c}\log_B N + \frac{\alpha}{B^{1-c}} + \frac{\delta}{B})$ I/Os and $O(\log N + \delta)$ time, worst-case and amortized respectively for any fixed constant c in the range $\frac{1}{\log B} \leq c \leq 1$.

9.5.1 Structure

The data structure consists of a search tree, T, constructed on a set of $\Theta(N/(\delta \log^3 N))$ leaf structures, each containing $\Theta(\delta \log^3 N)$ elements.

The top-level search tree T is based on the binary trees of Brodal et al. [35], with all elements and pointers replicated $2\delta + 1$ times. We maintain T such that all levels of the tree, except possibly the last $O(1)$ levels, are complete [35]. Additionally, we maintain an auxiliary static dictionary D_T, described in Section 9.4, containing the replicated guiding elements stored in the lowest level of T that is guaranteed to be complete.

A leaf structure contains $\Theta(\delta \log^3 N)$ elements distributed, in sorted order, into $\Theta(\log^3 N)$ buffers of size $\Theta(\delta)$. The buffers are the leaves of a three level search tree L_T with fan-out $\Theta(\log N)$. Similar to T the components of L_T are stored reliably. Finally, the $2\delta + 1$ copies of each of the $\Theta(\log N)$ elements in each internal node of L_T are stored in a static dictionary (Section 9.4). Note that the elements in the buffers stored in the leaves of L_T, however, are *not* replicated.

9.5.2 Operations

A query operation for an element q initially queries the dictionary D_T. The constant sized subtrees of T rooted at the nodes returned by this query are traversed by reliably determining the replicated pointers and guiding elements. This traversal identifies a leaf structure. The corresponding tree L_T is traversed starting from the root, using the static dictionaries stored in the internal nodes of L_T to guide the search, and retrieving the replicated pointers to traverse the tree reliably. Upon reaching a leaf in L_T, its associated buffer is scanned and the query returns true if q is found, otherwise it returns false.

To delete an element e from the dictionary it is first removed from the buffer that contains it. If the buffer becomes too small it is merged with a neighboring buffer, and the representative of the merged buffer becomes the largest of the two guiding elements associated with the respective buffers before the merge. The smaller guiding element is then removed from all ancestors of the merged leafs in L_T by completely rebuilding them. If the leaf structure L now contains too few elements it is merged with a neighboring leaf structure in T and rebuild. In this case the top tree T is also updated using the algorithm of [35] and the reliably stored elements and structural information. This update may alter the set of elements stored in the lowest complete level of T. Each new element in the lowest complete level overwrites the element it replaced in the static dictionary D_T. An insert is handled similarly.

Theorem 9.3 *The data structure described is a deterministic dynamic resilient dictionary supporting searches and updates in* $O(\frac{1}{c}\log_B N + \frac{\alpha}{B^{1-c}} + \frac{\delta}{B})$ *I/Os and* $O(\log N + \delta)$ *time, worst-case and amortized respectively.*

Proof. The top tree T uses $O(|T|\delta/B) = O(N/B)$ blocks and the size of each leaf structure is dominated by the elements stored in the leaf buffers.

In a query, one static dictionary storing $O(\frac{N}{\log^3 N})$ elements and three static dictionaries storing $O(\delta \log N)$ elements are searched, $O(1)$ replicated values are retrieved reliably, and a buffer storing $O(\delta)$ elements is scanned.

Completely rebuilding a static dictionary from a sorted set of N replicated elements takes $O(\delta N/B)$ I/Os and $O(\delta N)$ time. Therefore, rebuilding three nodes in L_T during an update takes $O(\delta \log N)$ time and $O(\frac{\delta}{B}\log N)$ I/Os. Since the size of the buffers in the leaves can vary by a constant factor, this is only needed every $\Theta(\delta)$ updates, meaning that the amortized cost of updating L_T is $O(\frac{\log N}{B})$ I/Os and $O(\log N)$ time. Rebuilding an entire leaf structure from a sorted set of buffers takes $O(\frac{\delta}{B}\log^3 N)$ I/Os and $O(\delta \log^3 N)$ time and this only

happens every $\Theta(\delta \log^3 N)$ updates. Thus, the amortized cost of updating a leaf structure is $O(\frac{1}{B})$ I/Os and $O(1)$ time.

The update time for the top tree T as presented in [35] is $O(\log^2 N)$. Since all components of T are stored reliably, updating T takes $O(\delta \log^2 N)$ time and $O(\lceil \frac{\delta}{B} \rceil \log^2 N)$ I/Os. When T is updated, all elements that have been replaced at the lowest guaranteed complete level of T are overwritten by the value replacing them in this level in the auxiliary static dictionary D_T. In the static dictionary from Section 9.4, an element can exist replicated $O(1)$ times on each level of the tree. Therefore, it takes $O(\delta \frac{1}{c} \log_B N)$ time and $O(\lceil \frac{\delta}{B^{1-c}} \rceil \frac{1}{c} \log_B N)$ I/Os to update such a value, and amortized $O(\log^2 N)$ elements are changed in each update of T. Again, since the size of the leaf structures can vary within a constant fraction, updates to T are only needed every $\Theta(\delta \log^3 N)$ update, thus the amortized update cost for T is $O(1)$ I/Os and $O(1)$ time. □

9.6 Sorting

In this section we present a resilient multi-way merging algorithm and use it to design a resilient I/O-efficient sorting algorithm. It is also used in the next section to design a resilient I/O-efficient priority queue. First we show how to merge γ faithfully ordered lists of total size x when $\gamma \leq \min\{\frac{M}{B}, \frac{M}{\delta}\}$.

9.6.1 Multi-way merging:

Initially, the algorithm constructs a perfectly balanced binary tree, T, in memory on top of the γ buffers being merged. Each edge of the binary tree is equipped with a buffer of size $5\delta+1$. Each internal node $u \in T$ stores the state of a running instance of the *PurifyingMerge* resilient binary merging algorithm described in Chapter 6 that works in rounds. Racall that, in each round $O(\delta)$ elements from both input buffers are read and the next δ elements in the faithful order are output. If corrupted elements are found, these are moved to a fail buffer and the round is restarted. The algorithm merges elements from the buffers on u's left child edge and right child edge into the buffer of u's parent edge. The states and sizes of all buffers are stored as reliable variables. The entire tree including all buffers and state variables are stored in internal memory, along with one block from each of the γ input streams and one block for the output stream of the root. Instead of storing a fail buffer for each instance of *PurifyingMerge*, a global shared fail buffer F is stored containing all detected corrupted elements.

Let b_l and b_r be the buffers on the edges to the left and right child respectively and let b denote the buffer on the edge from u to its parent. If u is the root, b is the output buffer. The elements are merged using the *fill* operation, which operates on u, as follows. First, it checks whether b_l and b_r contain at least $4\delta + 1$ elements, and if not they are filled recursively. Then, the stored instance of the *PurifyingMerge* algorithm is resumed by running a round of the algorithm outputting the next δ elements to its output stream. The multi-way merging algorithm is initiated by invoking *fill* on the root of T which runs until all

elements have been output. Then, the elements moved to F during the *fill* are merged into the output using *NaiveSort* and *UnbalancedMerge* as in [61].

Lemma 9.4 *Merging $\gamma = \min\{\frac{M}{B}, \frac{M}{\delta}\}$ buffers of total size $x \geq M$ uses $O(x/B)$ I/Os and $O(x \log \gamma + \alpha\delta)$ time.*

Proof. The correctness follows from Lemma 1 in [61]. The size of T is $O(\gamma(\delta + B)) = O(\min\{\frac{M}{B}, \frac{M}{\delta}\}(\delta + B)) = O(M)$. We use γ I/Os to load the first block in each leaf of T and $O(x/B)$ I/Os for reading the entire input and writing the output. The final merge with F takes $O(x/B)$ I/Os. Since T fits completely in memory we perform no other I/Os.

Merging two buffers of total size n using *PurifyingMerge* takes $O(n + \alpha\delta)$ time where α is the number of detected corruptions in the input buffers. Since detected corruptions are moved to the global fail buffer each corruption is only charged once. Each element passes through $\log \gamma$ nodes of T and the final merge using *NaiveSort* and *UnbalancedMerge* takes $O(x + \alpha\delta)$ time. □

9.6.2 Sorting

Assuming $\delta \leq M^\varepsilon$ for $0 \leq \varepsilon < 1$, we can use the multi-way merging algorithm to implement the standard external memory $M^{1-\varepsilon}$-way merge sort from [7] matching the optimal external memory sorting bound for constant ε.

Theorem 9.4 *Our resilient sorting algorithm uses $O(\frac{1}{1-\varepsilon} Sort(N))$ I/Os and $O(N \log N + \alpha\delta)$ time assuming $\delta \leq M^\varepsilon$.*

9.7 Priority queue

In this section we describe a comparison-based resilient priority queue which is optimal with respect to both time and I/O performance assuming that $\delta \leq M^\varepsilon$. An optimal I/O-efficient priority queue uses $\Theta(1/B \log_{M/B}(N/M))$ I/Os amortized per operation [7]. An $\Omega(\log N + \delta)$ time lower bound for comparison-based resilient priority queues was proved in Chapter 7.

Our priority queue is based on an amortized version of the worst-case optimal external memory priority queue of [37] and uses our new multi-way merging algorithm to move elements between disk and internal memory.

The priority queue consists of a part on disk, denoted $\mathcal{L}$, and three structures, D, I and F, in internal memory. We maintain that D stores the smallest elements in the priority queue and that I stores newly inserted elements. We maintain that D has more than $\delta + 1$ elements and that both I and D have at most $M + \delta + 1$ elements, the former ensures that there is at least one uncorrupted element in D. Finally, the buffer F, of size 2δ, stores possibly corrupted elements.

The structure on disk, $\mathcal{L}$ is a linked list of levels. Each level consists of a number of faithfully ordered sequences, represented by linked lists of blocks of size $\max\{B, \delta\}$, stored in a linked list. Let N_ℓ denote the number of buffers at level ℓ, and let $\mathcal{L}_\ell^i$, $0 \leq i \leq N_\ell$ denote the i'th buffer of level ℓ. The elements at level ℓ is $\mathcal{L}_\ell = \mathcal{L}_\ell^0 \cup \cdots \cup \mathcal{L}_\ell^{N_i}$. We define the parameter $\gamma = \min(M/B, M/\delta)$

and maintain that $N_\ell < \gamma$ and that $|\mathcal{L}_\ell| < \gamma^{\ell+1}M$. Let h denote the index of the highest level, the first level is level 0. The pointers in the linked lists as well as buffer offsets and sizes are stored as reliable variables. We describe two operations, PULL and PUSH on $\mathcal{L}$ that extracts and adds M elements to $\mathcal{L}$ respectively.

We will need the following result on selection in the faulty memory RAM: We say that element x in a buffer X has *faithful-rank* r if there is at least $r - \alpha - \delta$ uncorrupted elements in X smaller than x and at most $|X| - r + \alpha + \delta$ uncorrupted elements of X larger than x.

Lemma 9.5 *Given an integer k and a faithfully ordered sequence S, $|S| < M$, an element with a faithful-rank k in S can be selected in $O((\alpha+1)\delta)$ time. The element returned has index $i \in \{k, \ldots, k+\alpha\}$ in S.*

Proof. The algorithm maintains an index k', initialized to k in safe memory. Initially $S[k']$ i copied into a safe memory variable. The algorithm then checks whether the majority of the $2\delta+1$ elements immediately to the left of $S[k']$ are smaller than $S[k']$, and whether the majority of the $2\delta+1$ immediately to the right are larger than $S[k']$. If this is the case $S[k']$ is returned. Otherwise, the algorithm increments the value of k' by one and restarts. The complexity of the algorithm depends on the number of iterations. An iteration fails to select an element if the element at $S[k']$ is corrupted to a value larger than $\delta+1$ of the $2\delta+1$ preceding elements or a value smaller than the $\delta+1$ of the $2\delta+1$ succeeding elements. If $S[k']$ is uncorrupted the algorithm always terminates and returns $S[k']$ and thus at most α iterations are performed. These arguments also prove correctness of the algorithm. □

9.7.1 Operations on $\mathcal{L}$

This section describes the basic operations used to manipulate $\mathcal{L}$. Let $merger(\ell, x)$ denote an invocation of the multi-way merging algorithm that extracts x elements from level ℓ. Let T be the binary tree with N_ℓ leaves used in the merging algorithm.

If $x < |\mathcal{L}_\ell|$ we do not output all the elements of $\mathcal{L}_\ell$. In that case we run the merger without fetching new elements from the leaves of T once x elements have been output. This empties all the buffers of T. The extra elements are gathered into a buffer E of size at most $6M$. We prepend E to a buffer of level ℓ making sure this buffer remains faithfully ordered. If there is a buffer, $\mathcal{L}_\ell^j$ with $|\mathcal{L}_\ell^j| < 6\delta$ we simply append this buffer to E and sort the result using the resilient sorting algorithm. If no such buffer exist we use the selection algorithm from Lemma 9.5 with $k = 2\delta+2$ for all N_ℓ buffers of level ℓ in turn - remembering the maximum element returned so far in the reliable memory. The algorithm from Lemma 9.5 works by repeatedly trying different elements and checking whether or not it is consistently ordered with the $2\delta+1$ elements to its left and the $2\delta+1$ elements to its right. If not it tries the neighboring elements and continues in this fashion. We modify the algorithm such that a candidate element that doesn't work are moved to our global fail buffer F and

the front of the the list is compacted. Since each round and the compaction take $O(\delta)$ time, the entire operation takes $O(\alpha'\delta)$ time where α' is the number of corrupted elements we sample. By moving corrupted elements to F, we ensure that we never spend δ time in the selection algorithm for the same corruption more than once.

If the maximum element is from buffer $\mathcal{L}_\ell^j$ we take the first 6δ elements of $\mathcal{L}_\ell^j$ and merge them with E, using the complete binary merging algorithm [56] and prepend the resulting buffer to $\mathcal{L}_\ell^j$. Finally, the extracted elements are returned.

Lemma 9.6 *An invocation of* merger(ℓ,x) *for $x \geq M$ returns the x faithfully smallest elements from $\mathcal{L}_\ell$ using $O(x/B)$ I/Os and $O(x \log N_\ell + \alpha\delta)$ time and leaves all buffers of $\mathcal{L}_\ell$ faithfully ordered.*

Proof. The I/O-complexity follows from Lemma 9.4. Extracting the x elements from the merger and emptying the remaining elements into E uses $O(x \log N_\ell + \alpha\delta)$ time by Lemma 9.4. The $O(N_\ell)$ invocations of the selection algorithm and performing compactions use $O(N_\ell\delta + \alpha'\delta) = O(M + \alpha'\delta) = O(x + \alpha'\delta)$ time where α' is the number of corrupted elements chosen as selection candidates.

We empty the merger into a buffer E and locate a buffer m in which E added to. Finding this list involves using selection N_ℓ times and merging the first 6δ elements of the buffer with the elements of E, for a total cost of $O(N_\ell\delta + \delta\alpha + M) = O(M + \delta\alpha) = O(x + \delta\alpha)$.

We must also prove that the faithful ordering of the buffer $\mathcal{L}_\ell^j$ in which we insert the elements of E is maintained. Recall that $\mathcal{L}_\ell^j$ was the buffer in which the selection algorithm from Lemma 9.5 returned the largest element r when invoked with $k = 2\delta + 2$. Let y be the maximum uncorrupted element of E, we now prove that $y \leq r$.

Let $\mathcal{L}_\ell^p$ be the buffer from which y originated and let c be the element returned by the selection algorithm for $\mathcal{L}_\ell^p$. Since c has faithful-rank $2\delta + 2$ there must be at least one uncorrupted element in $\mathcal{L}_\ell^p$ smaller than c, this element is also smaller than y and thus $y \leq c$. Since r was the maximum among all the selections we know that $r \geq c$ and thus we conclude that $y \leq c \leq r$. □

Pushing elements to $\mathcal{L}$

The Push operation inserts a buffer, A, of size M into $\mathcal{L}$. It does so by simply adding it as a new list of level $i = 0$ and incrementing N_i by one. If $N_i = \gamma$ all elements of level i are merged into a new list, $A' = \mathit{merger}(i, |\mathcal{L}_i|)$ which is then recursively inserted into level $i + 1$, level i is now empty.

Lemma 9.7 *The* Push *operation adds M elements to $\mathcal{L}$ and maintains that $N_\ell < \gamma$ and $|\mathcal{L}_\ell| < \gamma^{\ell+1}M$ for $0 \leq \ell \leq h$ and that all buffers in $\mathcal{L}$ remain faithfully ordered. For each visited level, ℓ, it uses $O(|\mathcal{L}_\ell| \log \gamma + \alpha\delta + N_\ell\delta)$ time and $O(|\mathcal{L}_\ell|/B)$ I/Os while pushing $|\mathcal{L}_\ell|$ elements to the next level.*

Proof. By construction, the number of buffers in each level is maintained by Push. We now prove that $|\mathcal{L}_\ell| < \gamma^{\ell+1}$ for all levels ℓ after the operation.

This claim is proved by induction. Initially, all lists on level 0 has size M, since they are inserted when the I buffer is full and we push M elements into $\mathcal{L}$. Now assume that the claim holds for level i. PUSH is invoked on level i, only when $N_i = \gamma$. Since all buffers have size γ^i and there are γ of them, the size of $merger(i, |\mathcal{L}_i|)$ is at most $|\mathcal{L}_i| = \gamma^{i+1}$ and thus, the list inserted on level $i+1$ is not to large. Elements moved to the fail buffer only makes levels smaller. The correctness of the multi-way merger proves that the buffers of $\mathcal{L}$ remain faithfully ordered. □

Pulling elements from $\mathcal{L}$

The PULL operation iterates from level $i = 0$ to level h and maintains a faithfully ordered buffer S of size $M+\delta$ which contains elements that are faithfully smaller than elements in $\mathcal{L}_0 \cup \cdots \cup \mathcal{L}_{i-1}$. Furthermore, an index $\ell \leq i$ is maintained with the property that $|\mathcal{L}_j|$ is unchanged for $j \leq i, j \neq \ell$, and $\mathcal{L}_\ell$ contains $M+\delta$ fewer elements than before the PULL.

Initially $S = merger(0, M+\delta)$, $i = 1$ and $\ell = 0$. Each iteration proceeds as follows. First the $M+\delta$ faithfully smallest elements from level i are extracted by invoking $A = merger(i, M+\delta)$. The complete binary merging algorithm from [56] is then invoked on A and S to produce a buffer C of size $2(M+\delta)$. We now split C into two halves $C_1 = C[1, \ldots, M+\delta]$ and $C_2 = C[M+\delta+1, \ldots, 2(M+\delta)]$. We set $S = C_1$. We place C_2 in level ℓ or level i. To do this we perform the same operation we used to put E back into $\mathcal{L}$ in the multi-way merging algorithm. The only difference is that we look at all buffers on level i and on level ℓ. If C_2 is placed on level ℓ we set $\ell = i$ for the next iteration. When all levels have been visited, the first M elements of A are merged with D. The remaining elements are inserted into I.

Lemma 9.8 *The* PULL *returns $M+\delta$ elements in faithful order and the M first are the faithfully smallest elements of $\mathcal{L}$. All buffers in $\mathcal{L}$ remain sorted. Ignoring the $O(\alpha)$ elements moved to the fail buffer, the size of all but one level remains the same, and the last level contains $M+\delta$ fewer elements. A* PULL *uses $O(h\frac{M}{B})$ I/Os and $O(hM \log N_\ell + \alpha\delta)$ time.*

Proof. We first prove, that the M smallest uncorrupted elements of $\mathcal{L}$ are extracted, we then prove that buffers in $\mathcal{L}$ remain sorted, and finally we prove the time and I/O bounds.

1. To prove (1) we basically need to prove that the following invariant holds: After performing an iteration of the PULL algorithm at level i the $M-\delta-\alpha$ smallest uncorrupted elements of A are smaller than any uncorrupted element in $\mathcal{L}_0 \cup \cdots \cup \mathcal{L}_i$. Since $|A| = M+\delta$, this invariant implies that the M first elements of A are smaller than any uncorrupted elements in $\mathcal{L}_0 \cup \cdots \cup \mathcal{L}_i$.

   Before proving the invariant we need to prove that the smallest $M+\delta-\alpha$ uncorrupted elements of C are in C_1 which become the new S. Let X denote the set containing the $M+\delta-\alpha$ smallest uncorrupted elements in

C and let $m = \max\{x \in X\}$. Since C is faithfully ordered, all elements of X appear faithfully ordered in C and there are no uncorrupted elements larger than m in positions before m by definition of X. Thus, the only elements not in X that can be stored in C before m are corrupted elements. There are at most α of these and thus, the index of m is smaller than $M + \delta - \alpha + \alpha = M + \delta$, implying that all of X is in $C_1 = C[1, \ldots, M + \delta]$.

We prove that the invariant holds by induction. It holds in the base case by Lemma 9.6. For the general case, assume that we have just completed the iteration at level i and let α' denote the number of corruptions before the most recent iteration, and $\alpha \geq \alpha'$ the number of corruption after the iteration. Let X' be the $M + \delta - \alpha$ smallest elements after the iteration and X the $M + \delta - \alpha'$ smallest elements before. We need to prove that any element in X' is smaller than any uncorrupted element in $\mathcal{L}_0, \ldots, \mathcal{L}_i$ under the assumption that any element from X is smaller than any uncorrupted element in $\mathcal{L}_0, \ldots, \mathcal{L}_{i-1}$. Since all the elements of X' are uncorrupted, it is enough to prove that the largest element, $m \in X'$ is smaller than all uncorrupted elements in $\mathcal{L}_0 \cup \cdots \cup \mathcal{L}_i$.

We split the proof in two cases.

- $m \in X' - X$: In this case m originates from $A \subset \mathcal{L}_i$. By the correctness of the merger, m is smaller than everything remaining in $\mathcal{L}_i$. m is also, by definition of X', smaller than everything in $A - X'$. We need to prove that m is also smaller than everything in $\mathcal{L}_0 \cup \cdots \cup \mathcal{L}_{i-1}$. Assume there is an element $x \in X$ larger than m. If so, we have that $m \leq x \leq y$ for any uncorrupted element $y \in \mathcal{L}_0 \cup \cdots \cup \mathcal{L}_{i-1}$, by the induction hypothesis. We now prove that it is impossible for m to be larger than all elements of X, we do this by contradiction and assume that m is bigger than all $x \in X$. Thus, for m to be the $M + \delta - \alpha'$ largest uncorrupted in C_1 it must be smaller than at least $|X| - |X'| + 1 = \alpha' - \alpha + 1$ elements from X. But since m is uncorrupted and larger than all the elements of X which are uncorrupted, each of the elements from X that are not in X' must be corrupted instead. Thus $\alpha' - \alpha + 1$ elements needs to have been corrupted in S or corrupted during the merge, which is impossible by definition of α and α'.
- $m \in X \cap X'$: By the induction hypothesis, x is smaller than all uncorrupted elements of $\mathcal{L}_0 \cup \cdots \cup \mathcal{L}_{i-1}$. It remains to be shown that m is also smaller than all elements of $\mathcal{L}_i$. Since m was from X, and thus *not* from A, we know that the number of elements in S, but not in X', is $|S - (X' \cap S)| \geq |S| - |X'| + 1 = \alpha' + 1$ and by definition of α', there is at least one uncorrupted element of S not in X' and the smallest of these elements are smaller than x by definition of X', but larger than all remaining uncorrupted elements in $S - S \cup X'$ and $\mathcal{L}_i$ by Lemma 9.6.

2. We need to prove that all lists remain sorted after Pull. The only change to the lists happen when elements are removed from the lists by the merger,

and when the elements left in the merger are re-inserted in level ℓ or level i. Removing elements from a buffer does not change the order of the remaining elements. The second part was proved in Lemma 9.6.

3. The complexity of the PULL operation is upper bounded by the invocations of $merger(\ell, M + \delta)$ for $0 \leq \ell \leq h$.

□

9.7.2 Operations on internal buffers

The internal data structures, D and I, are implemented using the optimal internal memory resilient dynamic dictionaries presented in Chapter 8. Recall that these dictionaries maintain n elements under updates and searches in amortized $O(\log n + \delta)$ time per operation and use $O(n)$ space.

To insert x into the priority queue, we simply insert it into I. If I grows to size M we push M elements into $\mathcal{L}$. We do this by using the resilient binary merging algorithm to merge I and D into a new buffer E. We now re-insert the first $|D| - \delta$ elements of E into a D, the next δ are put in I and the remaining M elements are pushed to $\mathcal{L}$ using the PUSH operation. If $|D| < \delta + 1$ now it is filled by a PULL operation.

A DELETEMIN finds the minimum element in F, in the first $\delta + 1$ elements of D and of the first $\min(\delta + 1, |I|)$ elements of I and returns the minimum of these three, the element is deleted from its buffer. To find the $\delta + 1$ smallest elements from D and I respectively, we need to extract the $\delta + 1$ minimum elements from the resilient dictionaries in Chapter 8. This dictionary stores the elements in buffers of size $\Theta(\delta)$ and one can move between successor and predecessor buffers in $O(\delta)$ time. If a DELETEMIN causes D to become smaller than $\delta + 1$, we *pull* $M + \delta$ from $\mathcal{L}$ and fill D with the M first elements. The last δ elements are put into I. If this causes I to become larger than M we empty I into $\mathcal{L}$ as above.

Since h, the number of levels of $\mathcal{L}$, might not decrease even if DELETEMIN is invoked many times, we use global rebuilding and rebuild the entire data structure every $\Theta(N)$ operations.

Theorem 9.5 *Our data structure is a linear space resilient priority queue supporting operations* INSERT *and* DELETEMIN *in amortized* $O(\frac{1}{1-\varepsilon}(1/B)\log_{M/B}(N/M))$ *I/Os and* $O(\log N + \delta)$ *time, assuming* $\delta \leq M^{\varepsilon}$ *for* $0 \leq \varepsilon < 1$.

Proof. To bound the cost of DELETEMIN and INSERT we bound the number of levels in $\mathcal{L}$. Since a level $\mathcal{L}_\ell$ is pushed to higher levels only if it is full and since we use global rebuilding, the highest level in the priority queue is $h = O(\log_\gamma(N/M))$. Both PUSH and PULL use $O(1/B)$ I/Os and $O(\log \gamma)$ time per element they touch on each level. Therefore, the total time, including the operations on I and D is $O(h \log \gamma + \log M + \delta) = O(\log N + \delta)$ per element. Furthermore, we use a total of $O(\alpha\delta)$ time to cope with corrupted elements. Thus, the total time used to perform N operations is $O(N((log_\gamma(N/M))(\log \gamma) + \log M + \delta) + \alpha\delta) = O(N \log N + \delta N + \alpha\delta)$, or amortized $O(\log N + \alpha\delta/N + \delta) = O(\log N + \delta)$ per operation.

The total I/O cost per element is $O(\frac{1}{B}h) = O(\frac{1}{B}\log_\gamma(N/M))$. Thus, the I/O complexity for N operations is $O(N(\frac{1}{B})\log_\gamma(N/M)) = O(\frac{1}{1-\varepsilon}\text{Sort}(N))$, or $O(\frac{1}{1-\varepsilon}(1/B)\log_{M/B}(N/M))$ amortized per element. □

Bibliography

Papers included in this dissertation

[D1] P. K. Agarwal, L. Arge, T. Mølhave, and B. Sadri. I/O-efficient algorithms for computing contours on a terrain. In *SCG '08: Proceedings of the twenty-fourth annual symposium on Computational geometry*, pages 129–138, New York, NY, USA, 2008. ACM.

[D2] L. Arge, T. Mølhave, and N. Zeh. Cache-oblivious red-blue line segment intersection. In *ESA '08: Proceedings of the 16th annual European symposium on Algorithms*, pages 88–99, Berlin, Heidelberg, 2008. Springer-Verlag.

[D3] G. S. Brodal, R. Fagerberg, A. G. Jørgensen, G. Moruz, and T. Mølhave. Optimal resilient dynamic dictionaries. Technical Report DAIMI PB-585, Department of Computer Science, Aarhus University, November 2007.

[D4] G. S. Brodal, A. G. Jørgensen, and T. Mølhave. Fault tolerant external memory algorithms. In *WADS '09: Proceedings of the 11th Algorithms and Data Structures Symposium.*, volume 5664 of *Lecture Notes in Computer Science*, pages 411–422, Berlin, Heidelberg, 2009. Springer-Verlag.

[D5] A. Danner, T. Mølhave, K. Yi, P. K. Agarwal, L. Arge, and H. Mitasova. TerraStream: from elevation data to watershed hierarchies. In *GIS '07: Proceedings of the 15th annual ACM international symposium on Advances in geographic information systems*, pages 1–8, New York, NY, USA, 2007. ACM.

[D6] A. G. Jørgensen, G. Moruz, and T. Mølhave. Priority queues resilient to memory faults. In *WADS '07: Proceedings of the 10th International Workshop on Algorithms and Data Structures*, volume 4619 of *Lecture Notes in Computer Science*, pages 127–138, Berlin, Heidelberg, 2007. Springer-Verlag.

Other papers coauthered by the author of this dissertation

[C1] L. Arge, L. Deleuran, K. D. Larsen, T. Mølhave, and M. Revsbæk. Practical and efficient algorithms for computing massive contour maps. In preparation.

[C2] L. Arge and T. Mølhave. GIS ved MADALGO. *Geoforum Danmark*, 104, May 2009.

[C3] G. S. Brodal, R. Fagerberg, I. Finocchi, F. Grandoni, G. Italiano, A. G. Jørgensen, G. Moruz, and T. Mølhave. Optimal resilient dynamic dictionaries. In *ESA '07: Proceedings of the 15th annual European symposium on Algorithms*, volume 4708 of *Lecture Notes in Computer Science*, pages 347–358. Springer-Verlag, Berlin, Heidelberg, 2007.

[C4] G. S. Brodal, A. G. Jørgensen, and T. Mølhave. Counting in the presence of memory faults. Submitted, 2009.

[C5] J. M. Eshøj, P. K. Bøcher, J.-C. Svenning, T. Mølhave, and L. Arge. Impacts of 21st century sea-level rise on a major city (Aarhus, Denmark) – an assessment based on fine-resolution digital topography and a new flooding algorithm. Submitted, 2009.

Other references

[1] M. Abadi, M. Budiu, Úlfar Erlingsson, and J. Ligatti. Control-flow integrity. In *Proc. 12th ACM conference on Computer and communications security*, pages 340–353, 2005.

[2] P. K. Agarwal, L. Arge, and A. Danner. From point cloud to grid DEM: A scalable approach. In A. Riedl, W. Kainz, and G. Elmes, editors, *Progress in Spatial Data Handling. 12th International Symposium on Spatial Data Handling*, pages 771–788. Springer-Verlag, 2006.

[3] P. K. Agarwal, L. Arge, T. M. Murali, K. R. Varadarajan, and J. S. Vitter. I/O-efficient algorithms for contour-line extraction and planar graph blocking. In *Proc. 9th ACM-SIAM Sympos. Discrete Algorithms*, pages 117–126, 1998.

[4] 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.

[5] 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.

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

[7] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31:1116–1127, 1988.

[8] D. Ajwani, A. Beckmann, R. Jacob, U. Meyer, and G. Moruz. On computational models for flash memory devices. In J. Vahrenhold, editor, *Proc. of the 8th International Symposium on Experimental Algorithms, SEA 2009*, volume 5526 of *Lecture Notes in Computer Science*. Springer, 2009.

[9] D. Ajwani, I. Malinger, U. Meyer, and S. Toledo. Characterizing the performance of flash memory storage devices and its impact on algorithm design. In *Proc. of the 7th International Workshop on Experimental Algorithms, WEA 2008*, volume 5038 of *Lecture Notes in Computer Science*. Springer, 2008.

[10] 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.

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

[12] L. Arge, M. A. Bender, E. D. Demaine, B. Holland-Minkley, and J. Ian Munro. An optimal cache-oblivious priority queue and its application to graph algorithms. *SIAM Journal on Computing*, 36(6):1672–1695, 2007.

[13] L. Arge, M. A. Bender, E. D. Demaine, B. Holland-Minkley, and J. I. Munro. Cache-oblivious priority queue and graph algorithm applications. In *Proc. 34th Annual ACM Symposium on Theory of Computing*, pages 268–276, 2002.

[14] L. Arge, H. Blunck, and A. H. Jensen. I/O-efficient nearest neighbor algorithms for grid dem quality evaluation. In preparation, 2009.

[15] L. Arge, G. S. Brodal, and R. Fagerberg. Cache-oblivious data structures. In D. Mehta and S. Sahni, editors, *Handbook of Data Structures and Applications*, chapter 34, page 27. CRC Press, 2005.

[16] 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.

[17] L. Arge, A. Danner, H. Haverkort, and N. Zeh. I/O-efficient hierarchical watershed decomposition of grid terrain models. In A. Riedl, W. Kainz, and G. Elmes, editors, *Progress in Spatial Data Handling. 12th International Symposium on Spatial Data Handling*, pages 825–844. Springer-Verlag, 2006.

[18] 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.

[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 and M. Revsbæk. I/O-efficient contour tree simplification. Submitted, 2009.

[21] 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.

[22] 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.

[23] L. Arge, D. E. Vengroff, and J. S. Vitter. External-memory algorithms for processing line segments in geographic information systems. *Proc. European Symposium on Algorithms*, pages 295–310, 1995.

[24] *LAS Specification Version 1.3 – R10.* The American Society for Photogrammetry & Remote Sensing (ASPRS), August 2009.

[25] Y. Aumann and M. A. Bender. Fault tolerant data structures. In *Proc. 37th Annual Symposium on Foundations of Computer Science*, pages 580–589, Washington, DC, USA, 1996. IEEE Computer Society.

[26] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan. The sorcerer's guide to fault attacks. *Proceedings of the IEEE*, 94(2):370–382, 2006.

[27] G. D. Battista and R. Tamassia. Algorithms for plane representations of acyclic digraphs. *Theor. Comput. Sci.*, 61(2-3):175–198, 1988.

[28] R. Baumann. Soft errors in advanced computer systems. *IEEE Design and Test of Computers*, 22(3):258–266, 2005.

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

[30] M. A. Bender, R. Cole, and R. Raman. Exponential structures for cache-oblivious algorithms. In *Proc. International Colloquium on Automata, Languages, and Programming*, pages 195–207, 2002.

[31] D. Boneh, R. A. DeMillo, and R. J. Lipton. On the importance of checking cryptographic protocols for faults. In *Eurocrypt*, pages 37–51, 1997.

[32] R. S. Borgstrom and S. R. Kosaraju. Comparison-based search in the presence of errors. In *Proc.25th Annual ACM symposium on Theory of Computing*, pages 130–136, 1993.

[33] R. S. Boyer and J. S. Moore. MJRTY: A fast majority vote algorithm. In *Automated Reasoning: Essays in Honor of Woody Bledsoe*, pages 105–118, 1991.

[34] G. S. Brodal and R. Fagerberg. Cache oblivious distribution sweeping. In *Proc. International Colloquium on Automata, Languages, and Programming*, pages 426–438, 2002.

[35] G. S. Brodal, R. Fagerberg, and R. Jacob. Cache-oblivious search trees via binary trees of small height. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 39–48, 2002.

[36] G. S. Brodal, R. Fagerberg, and R. Jacob. Cache-oblivious search trees via trees of small height. Technical Report ALCOMFT-TR-02-53, ALCOM-FT, May 2002.

[37] G. S. Brodal and J. Katajainen. Worst-case efficient external-memory priority queues. In *Proc. 6th Scandinavian Workshop on Algorithm Theory*, volume 1432 of *Lecture Notes in Computer Science*, pages 107–118. Springer Verlag, Berlin, 1998.

[38] H. Carr, J. Snoeyink, and U. Axen. Computing contour trees in all dimensions. *Computational Geometry: Theory and Applications*, 24:75–94, 2003.

[39] L. P. Chew. Constrained delaunay triangulations. *Algorithmica*, 4:97–108, 1989.

[40] 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.

[41] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proc. 6th ACM-SIAM Sympos. Discrete Algorithms*, pages 139–149, 1995.

[42] Y.-J. Chiang and C. T. Silva. I/O optimal isosurface extraction. In *Proc. IEEE Visualization*, pages 293–300, 1997.

[43] B. S. Chlebus, A. Gambin, and P. Indyk. Shared-memory simulations on a faulty-memory dmm. In *Proc. 23rd International Colloquium on Automata, Languages and Programming*, pages 586–597, 1996.

[44] B. S. Chlebus, L. Gasieniec, and A. Pelc. Deterministic computations on a pram with static processor and memory faults. *Fundamenta Informaticae*, 55(3-4):285–306, 2003.

[45] C. Constantinescu. Trends and challenges in VLSI circuit reliability. *IEEE micro*, 23(4):14–19, 2003.

[46] A. Danner. *I/O Efficient Algorithms and Applications in Geographic Information Systems*. PhD thesis, Department of Computer Science, Duke University, 2006.

[47] M. de Berg, P. Bose, K. Dobrindt, M. J. van Kreveld, M. H. Overmars, M. de Groot, T. Roos, J. Snoeyink, and S. Yu. The complexity of rivers in triangulated terrains. In *Proc. of the 8th Canadian Conference on Computational Geometry*, pages 325–330, 1996.

[48] M. de Berg, O. Cheong, H. Haverkort, J.-G. Lim, and L. Toma. The complexity of flow on fat terrains and its I/O-efficient computation. *Computational Geometry*, In Press, 2009.

[49] B. Demsky and M. C. Rinard. Goal-directed reasoning for specification-based data structure repair. *IEEE Transactions on Software Engineering*, 32(12):931–951, 2006.

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

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

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

[53] H. Edelsbrunner, D. Letscher, and A. Zomorodian. Topological persistence and simplification. *Discrete and Computational Geometry*, 28(4):511–533, 2002.

[54] B. Elkarablieh, I. Garcia, Y. L. Suen, and S. Khurshid. Assertion-based repair of complex data structures. In *Proc. 22nd IEEE/ACM International Conference on Automated Software Engineering*, pages 64–73, 2007.

[55] I. Fary. On straight lines representation of planar graphs. *Acta Sci. Math. Szeged*, 11:229–233, 1948.

[56] I. Finocchi, F. Grandoni, and G. F. Italiano. Optimal resilient sorting and searching in the presence of memory faults. In *Proc. 33rd International Colloquium on Automata, Languages and Programming*, volume 4051 of *Lecture Notes in Computer Science*, pages 286–298. Springer, 2006.

[57] I. Finocchi, F. Grandoni, and G. F. Italiano. Designing reliable algorithms in unreliable memories. *Computer Science Review*, 1(2):77–87, 2007.

[58] I. Finocchi, F. Grandoni, and G. F. Italiano. Resilient search trees. In *Proc. 18th ACM-SIAM Symposium on Discrete Algorithms*, pages 547–554, 2007.

[59] I. Finocchi, F. Grandoni, and G. F. Italiano. Optimal resilient sorting and searching in the presence of dynamic memory faults. *Theoretical Computer Science*, 2009. To appear.

[60] I. Finocchi and G. F. Italiano. Sorting and searching in the presence of memory faults (without redundancy). In *Proc. 36th Annual ACM Symposium on Theory of Computing*, pages 101–110, New York, NY, USA, 2004. ACM Press.

[61] I. Finocchi and G. F. Italiano. Sorting and searching in faulty memories. *Algorithmica*, 52(3):309–332, 2008.

[62] G. Franceschini. Proximity mergesort: Optimal in-place sorting in the cache-oblivious model. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 291–299. SIAM, 2004.

[63] G. Franceschini and R. Grossi. Optimal cache-oblivious implicit dictionaries. In *Proceedings of the 30th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 316–331, 2003.

[64] 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.

[65] 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.

[66] 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.

[67] S. Govindavajhala and A. W. Appel. Using memory errors to attack a virtual machine. In *IEEE Symposium on Security and Privacy*, pages 154–165, 2003.

[68] J. Gudmundsson, M. H. Hammar, and M. J. van Kreveld. Higher order delaunay triangulations. *Computational Geometry Theory & Applications*, 23(1):85–98, Jul 2002.

[69] H. Haverkort and J. Janssen. Simple I/O-efficient flow accumulation on grid terrains. Abstract collection Workshop on Massive Data Algorithms, Aarhus, 2009.

[70] D. Howe, M. Costanzo, P. Fey, T. Gojobori, L. Hannick, W. Hide, D. P. Hill, R. Kania, M. Schaeffer, S. St Pierre, S. Twigger, O. White, and S. Yon Rhee. Big data: The future of biocuration. *Nature*, 455(7209):47–50, September 2008.

[71] K. H. Huang and J. A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Transactions on Computers*, 33:518–528, 1984.

[72] Intel. Intel® Core™ i7 Processor Extreme Edition Series and Intel® Core™ i7 Processor Datasheet - Volume 1. Technical report, June 2009.

[73] M. Isenburg, Y. Liu, J. Shewchuk, and J. Snoeyink. Streaming computation of Delaunay triangulations. In *Proceedings of SIGGRAPH*, 2006.

[74] M. Isenburg, Y. Liu, J. Shewchuk, J. Snoeyink, and T. Thirion. Generating raster DEM from mass points via TIN streaming. In M. Raubal, H. Miller, A. Frank, and M. Goodchild, editors, *Geographic Information Science - Fourth International Conference, GIScience 2006, Münster, Germany, September 2006.*, 2006.

[75] 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.

[76] A. Joffe. On a set of almost deterministic k-independent random variables. *Annals of Probability*, 2(1):161–162, 1974.

[77] D. Kelly. Fundamentals of planar ordered sets. *Discrete Math.*, 63(2-3):197–216, 1987.

[78] S. Kutten and D. Peleg. Tight fault locality. *SIAM Journal on Computing*, 30(1):247–268, 2000.

[79] N. L. Lea. An aspect driven kinematic routing algorithm. In A. J. Parsons and A. D. Abrahams, editors, *Overland Flow: Hydraulics and Erosion Mechanics*. Chapman & Hall, New York, 1992.

[80] T. Leighton and Y. Ma. Tight bounds on the size of fault-tolerant merging and sorting networks with destructive faults. *SIAM Journal on Computing*, 29(1):258–273, 2000.

[81] A. Lempel, S. Even, and I. Cederbaum. An algorithm for planarity testing of graphs. In *Theory of Graphs: Internat. Symposium (Rome 1966)*, pages 215–232, New York, 1967. Gordon and Breach.

[82] W. Lorensen and H. Cline. Marching cubes: a high resolution 3d surface construction algorithm. *Comput. Graph.*, 21(4):163–170, 1987.

[83] P. Lyman and H. R. Varian. How much information? 2003.

[84] C. Mallet and F. Bretar. Full-waveform topographic lidar: State-of-the-art. *International Journal of Photogrammetry and Remote Sensing*, 64:1–16, January 2009.

[85] 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.

[86] 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.

[87] E. Moet, M. van Kreveld, and A. F. van der Stappen. On realistic terrains. *Computational Geometry*, 41(1-2):48 – 67, 2008. Special Issue on the 22nd European Workshop on Computational Geometry (EuroCG), 22nd European Workshop on Computational Geometry.

[88] J. I. Munro and S. S. Rao. Succint representation of data structures. In D. Mehta and S. Sahni, editors, *Handbook of Data Structures and Applications*, chapter 37. CRC Press, 2005.

[89] NASA and Japan (METI). NASA, Japan release most complete topographic map of earth, June 2009.

[90] Community cleverness required. *Nature*, 455, 9 2008.

[91] M. H. Nodine, M. T. Goodrich, and J. S. Vitter. Blocking for external graph searching. *Algorithmica*, 16(2):181–214, 1996.

[92] G. Novark, E. D. Berger, and B. G. Zorn. Exterminator: Automatically correcting memory errors with high probability. *Communications of the ACM*, 51(12):87–95, 2008.

[93] 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.

[94] U. F. Petrillo, I. Finocchi, and G. F. Italiano. The price of resiliency: a case study on sorting with memory faults. *Algorithmica*, 53(4):597–620, 2009.

[95] S. Pettie and V. Ramachandran. Minimizing randomness in minimum spanning tree, parallel connectivity, and set maxima algorithms. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 713–722, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics.

[96] D. K. Pradhan. *Fault-tolerant computer system design*. Prentice-Hall, Inc., 1996.

[97] B. Ravikumar. A fault-tolerant merge sorting algorithm. In *Proc. 8th Annual International Conference on Computing and Combinatorics*, pages 440–447, 2002.

[98] G. Reeb. Sur les points singuliers d'une forme de Pfaff complétement intégrable ou d'une fonction numérique. *Comptes Rendus de L'Académie ses Sciences*, pages 847–849, 1946.

[99] M. Z. Rela, H. Madeira, and J. G. Silva. Experimental evaluation of the fail-silent behaviour in programs with consistency checks. In *Proc. 26th Annual International Symposium on Fault-Tolerant Computing*, pages 394–403, 1996.

[100] M. Revsbæk. I/O efficient algorithms for batched union-find with dynamic set properties and its applications to hydrological conditioning. Master's thesis, Aarhus University, Denmark, 2007.

[101] R. Sibson. *Interpolating multivariate data*, chapter A Brief Description of Natural Neighbor Interpolation, pages 21–36. John Wiley & Sons, 1981.

[102] C. Silva, Y. Chiang, J. El-Sana, and P. Lindstrom. Out-of-core algorithms for scientific visualization and computer graphics. In *Visualization'02*, 2002. Course Notes for Tutorial 4.

[103] R. A. Skelton. Cartography. *History of Technology*, 6:612–614, 1958.

[104] S. P. Skorobogatov and R. J. Anderson. Optical fault induction attacks. In *Proc. 4th International Workshop on Cryptographic Hardware and Embedded Systems*, pages 2–12, 2002.

[105] 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.

[106] R. Tamassia and F. P. Preparata. Dynamic maintenance of planar digraphs, with applications. *Algorithmica*, 5(4):509–527, 1990.

[107] R. Tamassia and I. Tollis. A unified approach to visibility representations of planar graphs. *Discrete & Computational Geometry*, 1:321–341, 1986.

[108] R. Tamassia and J. S. Vitter. Optimal parallel algorithms for transitive closure and point location in planar structures. In *SPAA '89: Proceedings of the first annual ACM symposium on Parallel algorithms and architectures*, pages 399–408, New York, NY, USA, 1989. ACM Press.

[109] 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.

[110] Tezzaron Semiconductor. Soft errors in electronic memory - a white paper. http://www.tezzaron.com/about/papers/papers.html, 2004.

[111] A. Tribe. Automated recognition of valley lines and drainage networks from grid digital elevation models: a review and a new method. *Journal of Hydrology*, 139:263–293, 1992.

[112] 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.

[113] R. van der Pas. Memory hierarchy in cache-based systems. Technical report, Sun Microsystems, Sant a Clara, California, U.S.A., 2002.

[114] M. van Kreveld, R. van Oostrum, C. Bajaj, V. Pascucci, and D. Schikore. Contour trees and small seed sets for isosurface traversal. In *Proc. ACM Annual Symposium on Computational Geometry*, pages 212–219, 1997.

[115] J. J. van Zyl. The shuttle radar topography mission (srtm): a breakthrough in remote sensing of topography. *Acta Astronautica*, 48(5-12):559 – 565, 2001.

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

[117] 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.

[118] J. S. Vitter. *Algorithms and Data Streams for External Memory*. Foundations and Trends in Theoretical Computer Science. now Publishers, Hanover, MA, 2008.

[119] J. P. Wilson and J. C. Gallant. *Terrain Analysis : Principles and Applications*. John Wiley & Sons, New York, NY, 2000.

[120] K.-C. Wu and D. Marculescu. Soft error rate reduction using redundancy addition and removal. In *Proc. 2008 conference on Asia and South Pacific design automation*, pages 559–564, 2008.

[121] J. Xu, S. Chen, Z. Kalbarczyk, and R. K. Iyer. An experimental study of security vulnerabilities caused by errors. In *Proc.International Conference on Dependable Systems and Networks*, pages 421–430, 2001.

[122] S. S. Yau and F.-C. Chen. An approach to concurrent control flow checking. *IEEE Transactions on Software Engineering*, SE-6(2):126–137, 1980.