Space Efficient Data Structures and External Terrain Algorithms

PhD Thesis
of
Jakob Truelsen

MADALGO, Department of Computer Science, Aarhus University

April 18, 2015

Abstract - English

This thesis deals with handling large data, and is split into two parts. In the first part we look at space efficient data structures. That is we try to handle as large as possible data sets with the available memory. Here we first look at the $(1+\varepsilon)$-approximate range mode problem. For this problem we are given an array A, and want to construct a data structure that supports approximate range mode queries. Such a query consists of indices i and j, and must return an element e from $A[i,j]$ such that the frequency of e in $A[i,j]$ is within a factor $(1+\varepsilon)$ of the most frequent element. We present a $(1+\varepsilon)$-approximate range mode structure using $\mathcal{O}(\frac{n}{\varepsilon})$ space, which supports queries in $\mathcal{O}(\log\frac{1}{\varepsilon})$ time.

Next we look at implicit data structures. An implicit structure consists of indivisible elements stored in an array, where any additional information must be encoded in the order of the elements. We consider two distinct models: In the *weak implicit model* we are allowed to store $\mathcal{O}(1)$ words of information between operations, while in the *strong implicit model* only the size n of the array may be stored. In the strong implicit model we construct a dictionary with the working set property supporting $\mathsf{insert}(e)$ and $\mathsf{delete}(e)$ in $\mathcal{O}(\log n)$ time, and $\mathsf{find}(e)$ in $\mathcal{O}(\log \ell)$ time, where ℓ is the number of distinct elements searched for since e was last searched for. We construct a static dictionary with the finger search property. This structure supports $\mathsf{find}(e)$, $\mathsf{predecessor}(e)$ and $\mathsf{successor}(e)$ in time $\mathcal{O}(\log d(e,f))$ and $\mathsf{change\text{-}finger}(e)$ in time $\mathcal{O}(n^{\varepsilon})$ for any $\varepsilon > 0$. Here $\mathsf{change\text{-}finger}(e)$ sets the current finger f to be the element e and $d(e,f)$ is the rank distance between the current finger f and e. We show that under some constraints, this structure is optimal in the strict implicit model, while in the weak model exponential search on a sorted array would have the same performance except $\mathsf{change\text{-}finger}(e)$ would take only $\mathcal{O}(\log n)$ time. Also in the strict implicit model, we construct a dynamic finger search dictionary achieving the same bounds as the static version mentioned above.

In the second part of this thesis we look at I/O algorithms, where we have given up on the idea of fitting all data into memory, and instead try to limit the number of disk accesses performed. We look at practically efficient terrain algorithms. First we construct an algorithm that, given a raster of size $\sqrt{N}\times\sqrt{N}$, can construct all $\sqrt{N}$ down scalings in $\mathcal{O}(\mathrm{scan}(N))$ I/Os in the cache oblivious model. The algorithm is shown to be efficient in practice.

Secondly we construct an algorithm that can simplify a planar decomposition, such as a contour map or a watershed map. Such a planar decomposition forms a planar graph with nodes and line segments between the nodes. This graph separates the different faces of the decomposition. The simplification consists of removing nodes of degree two. It maintains several key properties: first the input and output decompositions are homotopic, secondly any segment in the input is at most a distance ε_{xy} away from the segment that replaces it in the output. For the special case of contours we also make a similar guarantee in the z-axis. The algorithm uses $\mathcal{O}(\mathrm{sort}(N))$ I/Os under some realistic assumptions, and is shown to be efficient in practice.

Abstract - Danish

Denne afhandling omhandler håndtering af store datamængder, og består af to dele. Den første del handler om pladseffektive datastrukturer. Her ser vi første på det $(1+\varepsilon)$-approksimative intervaltypetalsproblem. I dette problem er vi givet et array A, og skal konstruere en datastruktur, som understøtter approksimative intervaltypetalsforespørgsler. En sådan forespørgsel består af indekser i og j, og skal returnere et element e fra $A[i,j]$, som har en hyppighed, der er højest en faktor $(1+\varepsilon)$ fra det mest hyppige element. Vi præsenterer en $(1+\varepsilon)$-approksimativ intervaltypetalsdatastruktur der bruger $\mathcal{O}(\frac{n}{\varepsilon})$ plads, og som understøtter forespørgsler i $\mathcal{O}(\log \frac{1}{\varepsilon})$ tid.

Derefter ser vi på implisite datastruktuere. En implicit datastruktur består af udelelige elementer, som er gemt i et array, hvor alt yderligere information skal kodes i rækkefølgen af elementerne. Vi studere to forskellige modeller: I den *svage implicite model* er det tilladt at vi gemmer $\mathcal{O}(1)$ ord af information mellem operationer, hvorimod man i den *stærke implicite model* kun må gemme størrelsen af arrayet n. I den stærke implicite model konstruerer vi en ordbog med arbejdssæts egenskaben, som understøtter $\mathsf{insert}(e)$ og $\mathsf{delete}(e)$ i $\mathcal{O}(\log n)$ tid, og $\mathsf{find}(e)$ i $\mathcal{O}(\log \ell)$ tid, hvor ℓ antallet af unikke elementer, der er søgt efter siden e sidst blev søgt efter. Vi konstruerer en statisk ordbog med fingersøgnings egenskaben. Denne struktur understytter $\mathsf{find}(e)$, $\mathsf{predecessor}(e)$ og $\mathsf{successor}(e)$ i tid $\mathcal{O}(\log d(e,f))$ og $\mathsf{change\text{-}finger}(e)$ i tid $\mathcal{O}(n^{\varepsilon})$ for et vilkårligt $\varepsilon > 0$. Her flytter $\mathsf{change\text{-}finger}(e)$ den nuværende finger f til elementet e, og $d(e,f)$ er rangafstanden mellem den nuværende finger f og e. Vi viser at denne struktur under visse betingelser er optimal i den stærke implicitte model, mens eksponentiel søgning i et sorteret array i den svage model vil have den samme udførselstid, undtagen for $\mathsf{change\text{-}finger}(e)$ som kun vil tage $\mathcal{O}(\log n)$ tid. I den stærke implicitte model konstruerer vi også en dynamisk fingersøgnings ordbog, der opnår de samme udførselstider som den statiske struktur.

I den anden del af denne afhandling ser vi på I/O algoritmer, her opgiver vi at få alt data til a være in rammen, og prøver istedet at begrænse antallet tilgange til disken. Specifikt ser vi på praktisk effektive terrænalgoritmer. Først konstruerer vi en algoritme, som givet en raster af størrelse $\sqrt{N} \times \sqrt{N}$, konstruerer alle $\sqrt{N}$ nedskaleringer i $\mathcal{O}(\text{scan}(N))$ I/Oer i den cache uvidende model. Denne algoritme vises at være effektiv i praksis.

Derefter konstruerer vi en algoritme, der kan simplificere planare opdelinger, som f.eks. kurvekort eller vandoplandskort. En sådan planar opdeling danner en planar graf med knuder og linjesegmenter mellem knuderne. Denne graf deler fladerne i opdelingen. Simplifikationen består i at fjerne knuder med grad to, hvor flere vigtige egenskaber overholdes. Input og output opdelingerne vil være homotopiske og ethvert segment i inputtet har højst afstand ε_{xy} til det segment, der erstatter det i outputtet. For det specielle tilfælde af kurvekort giver vi også en tilsvarende garanti for z-aksen. Algoritmen udfører $\mathcal{O}(\text{sort}(N))$ I/Oer under visse realistiske antagelser, og vi viser at den er effektiv i praksis.

Preface

I started my PhD thinking about looking into cache-oblivious data structures, and while I did produce some cache-oblivious structures along the way, this was more a by-product than a deliberate act. Instead, my work during my PhD has been in two main areas, namely that of space efficient data structures and practical terrain algorithms. Here the space efficient data structures seek to explore the theoretical limits of space usage, and are not useful in practice as is. The terrain algorithms on the other hand are of immediate use, while still theoretically sound.

This thesis reflects this split focus. Chapter 1 is comprised of a summary of my work. It contains descriptions of the various problems I have worked on along with previous work. It contains a rough description of the results of my co-authors and me, as well as information about recent developments in the fields. Chapters 2 and 3 contain some of the work on space efficient data structures, while chapters 4 and 5 contain some of the work on practical, efficient terrain algorithms. Each of these chapters are based on papers published during my PhD, and include text written by co-authors, as detailed below.

Range Mode Chapter 2 is based on "Cell Probe Lower Bounds and Approximations for Range Mode" [39], which is co-authored with Mark Greve, Allan Grønlund Jørgensen and Kasper Dalgaard Larsen. The paper was presented at ICALP 2010 by Kasper. Here I contributed mainly to the 3- and $(1 + \varepsilon)$-approximations.

Implicit Dictonaries Chapter 3 is based on "A Cache-Oblivious Implicit Dictionary with the Working Set Property" [19], which is co-authored with Gerth Stølting Brodal and Casper Kejlberg-Rasmussen. The paper was presented at ISAAC 2010 by me.

It is also based on "Finger Search in the Implicit Model" [20], which is co-authored with Gerth Stølting Brodal and Jesper Sindahl Nielsen. The paper was presented at ISAAC 2012 by Jesper.

Computing Multiresolution Rasters Chapter 4 is based on "An Optimal and Practical Cache-Oblivious Algorithm for Computing Multiresolution Rasters" [7], which is co-authored with Lars Arge, Gerth Stølting Brodal and Constantinos Tsirogiannis. The paper was presented at ESA 2013 by Constantinos.

Decomposition Simplification Chapter 5 is based on "Simplifying Massive Planar Subdivisions" [11] which is co-authored with Lars Arge and Jungwoo Yang. The paper was presented at ALENEX 2014 by Jungwoo. This work in an improvement

of "Simplifying Massive Contour Maps" [8], which is co-authored with Lars Arge, Lasse Deleuran, Thomas Mølhave and Morten Revsbæk. The paper was presented at ESA 2012 by me.

Other Work During my PhD, I also did other work not included in this thesis. Very early on in my PhD, I developed a pipelining framework for TPIE [1]. This has had a number of rewrites and is now very useful, for instance all the experimental results in this thesis are based on this framework. We are currently in the process of writing a paper detailing its construction.

I spend quite some time trying to use TPIE pipelining to construct an external algorithm for model checking with visiting a Czech PhD student Martin Šmérek, however this turned out to be infeasible. The problems were mainly that the pruning needed to make the problem handleable would occur later in an external algorithm than an internal one.

As already mentioned I co-authored the paper "Simplifying Massive Contour Maps" [8], on which "Simplifying Massive Planar Subdivisions" [11] is partly based. The former paper is not part of this thesis, since it is superseded by the latter.

I have co-authored the paper "Strictly Implicit Priority Queues" together with Gerth Stølting Brodal and Jesper Sindahl Nielsen. The paper is not part of this thesis and has yet to be published. In the paper we construct two priority queues in the strict implicit model, where between operations only an array of elements and the size n of the structure are stored. The first structure supports insert and delete-min in $\mathcal{O}(1)$ and $\mathcal{O}(\log n)$ time amortised respectively, however the delete-min operation performs only $\mathcal{O}(1)$ element moves. The second gives worst-case $\mathcal{O}(1)$ time insert and $\mathcal{O}(\log n)$ time delete-min. Both structures require that the elements in the structure are distinct. What we would really like is a priority queue with $\mathcal{O}(1)$ time insert and $\mathcal{O}(\log n)$ time delete-min which does $\mathcal{O}(1)$ moves, and which does not require distinct elements, however this seems very hard.

Contents

List of Figures

1

Introduction

This thesis is within the field of theoretical computer science. In this field we try to model various aspects of computers and computations using math. In this thesis we look at various computation problems whose solutions can be decomposed into data structures and algorithms.

An algorithm is a recipe for solving a given problem. A problem is some input and the solution is some output. One common problem could be sorting, where the input consists of a list of comparable items and the output is the same list of items now ordered from lowest to highest according to some total ordering. We typically say that the input is of size n. Among the important properties we care about for an algorithm are 1) Correctness: The output should look like we want. 2) Running time: How long time does it take for the algorithm to compute the solution on an input of size n. 3) Space usage: How much memory do we use while computing the solution.

An important component in most algorithms (and also in their own right) are data structures. Here we wish to store data in such a way that we can quickly answer questions about the data, and update the structure with new data. A well known data structure is the set. Here we can insert an element e into the set, or remove an element e. We can also ask if the element is already in the set. Typically we say that the number of elements in the data structure is n. Again we want each operation to be correct and fast. We also care about the amount of space used both between and during operations.

As long as there has been computers, there have been data sets just a bit to large to be handled in the memory of the computers. As memory capacity has grown with Moore's law (which predicts that the maximal transistor count of an integrated circuit will roughly double every two years), the size of the data we want to process has grown as well. This is sometimes in jest called Parkinson's law "work expands so as to fill the time available for its completion" or for computation "data expands to fill the space available for storage". In a sense it is natural that we want to process as much data as possible. This thesis deals with two different ways one can push the boundary on how much data can be processed. In the first part we look at space efficient data structures, where we try to decrease the amount of memory (RAM) needed in order to store a given amount of data, such that larger data sets may be processed. In the second part we look at external memory algorithms, where we store data on disk, instead of solely in memory, in such a way as to minimize the number of times we need to access the disk. Before going into details we will (in Section 1.1) describe the models of computations used in this thesis. Next in Section 1.2, and 1.3 we introduce various results in space efficient data structures.

Finally in Section 1.4, and 1.5 we introduce results within the area of external memory terrain algorithms.

1.1 Models of computation

When analysing the work done by an algorithm in theoretical computer science it is important to describe the model of computation used during the analysis. That is a description of what a computation device can do and what resources are considered in the analysis. Without a precise definition of the model it is very hard to argue that you considered everything in your analysis. To make comparing algorithms by different authors easier we often analyse our algorithms in a predefined model also used by others. The different models used in this thesis are defined below.

1.1.1 Word RAM

The arguably most used model for upper bounds is the "word random access memory model" (often just called the RAM). In this model we model only the memory and the CPU of the computer. The memory of a computer consists of an array of words each w bits long. Arbitrary memory cells can be read at any time using an index (also know as a pointer). A data structure with n elements will use some $S(n)$ words of space, we assume that $\log(S(n)) \leq w$ (here and throughout the rest of the thesis log will denote the binary logarithm), such that a single word can encode a pointer to any other word. The CPU can perform operations on the words, the operations addition, subtraction, bitwise xor, bitwise and, bitwise or and bit shifts are always allowed. Most authors also allow multiplication and division, and some even allow all AC^0 operations. All operations are assumed to be unit cost such that the time of an algorithm is linear in the number of operations performed. This model is a fairly good approximation of a computer, the set of operations allowed matches very well with what a CPU supplies, however the model does not take the cache hierarchy into account, for instance both merge-sort and quick-sort take $\mathcal{O}(n \log n)$ time in this model, while quick-sort preforms significantly less cache faults [49].

1.1.2 Cell probe

As stated above the RAM model is good for analysing upper bound behaviour of algorithms since the set of CPU operations allowed matches fairly well with what a modern CPU can do. Proving lower bounds in the RAM model however is fairly hard. First there is not full agreement on what operations are allowed, and secondly even if there where, the set of operations is so large that arguing for even the most trivial problem is next to impossible. There are basically two ways to go about this. One could restrict the operations allowed, for instance by making elements indivisible and only allowing group operations, as done in the group model. Here it is often somewhat easier to argue about what an algorithm can do [36, 50], however these kinds of lower bounds do not apply to the general RAM, and we will not talk more about them in this thesis. Another way to go about this is to extend the set of operations allowed by the CPU. While this of course does not make it easier to prove lower bounds, it makes it easier to focus on what makes a given problem hard. In the cell probe model of Yao [70], this is exactly what is done. Here the memory is again divided into cells of w bits each, and the complexity of an algorithm is determined

by the number of these cells the algorithm accesses. That is, all operations done in the CPU are free, and we allow the CPU to do any kind of operation on its registers, even solving NP-hard or unsolvable problems. It is clear that a lower bound in this model also provides a lower bound in the RAM model, since any operation that touches a memory cell in the RAM model has cost at least one.

1.1.3 Implicit models

When designing data structures in the RAM model, an important parameter is the amount of space $S(n)$ used as a function of the number of elements the data structure contains. To minimize this there are two ways to go. In the succinct models, the distribution of the input elements is taken into account, and data structures can be created that require the minimal number of bits plus some lower order term [43, 44, 28, 59]. While this line of research is interesting and worthwhile we will in this thesis explore another path. We will consider the elements stored in the data structure as indivisible black-box elements that can only be compared or swapped, each occupying exactly one word of memory. It is clear that the minimal amount of words needed to store such a data structure is n, and in the implicit model this is exactly what we aim for. More precisely a data structure of n elements in the implicit model is an array with exactly n entries. Computation is done in a CPU with a constant number of registers with a word size of $\Theta(\log n)$ bits. Similar to the RAM model all operations on registers are unit cost. A register can either contain an integer, in which case we allow exactly the same operations as in the RAM model, or an element. Elements can be compared, read from the array, or written back to the array and nothing else.

There is no real agreement about how much memory may be used between operations on the structure, we therefor define two different implicit models. In the *weak implicit model* $\mathcal{O}(1)$ extra words of information may be stored between operations words [33, 34, 54]. In the *strong implicit model* **no** additional space is allowed, only the number of elements n is assumed to be implicitly maintained [15, 35].

1.1.4 Strict lower bound model

Similarly to the RAM model, proving lower bounds in the implicit model is not easy, since the complexity of operations allowed on the integer registers is too high. Given that the strict implicit model is a restriction of the weak implicit model, which is a restriction of the RAM model, which is a restriction of the cell probe model, we could just use the cell probe model to prove lower bounds. However as we will see in Chapter 3 some problems are harder in the strict implicit model than in the RAM model. We therefore need another model to prove lower bounds. The model we propose is the following. We will count only the array accesses (cell probes) and the element comparisons. To make it simpler for ourselves, we say that in one time step an algorithm is allowed to make at most one array access and one comparison. We do not bound the number of registers, so any two already loaded elements can be compared and any already loaded element may be written to the array in any given step. Since our model is used to prove lower bounds on the strict implicit model, we will make the requirement that the only thing stored in the registers of

the CPU when an algorithm (operation on the data structure) starts, is the number of element n and any possible arguments to the algorithm.

Consider a static data structure of size n, and an operation on this structure that takes only elements or keys as arguments (for instance $\mathsf{find}(k)$ on a dictionary). Then the first cell accessed during the operation must always be the same, independent of the content of the data structure. This is true since the content of any integer register is always the same initially (we do not allow randomization). Now after the first operation is performed an element has been loaded, and it might have been compared to some of the arguments. However the result of this comparison yields only a single bit of information, so for the next step at most one of two different possible cells may be probed (depending on the outcome of the comparison). Extending this we prove in Lemma 3.1 that for any algorithm A on a strict implicit data structure of size n that runs in time at most τ, whose arguments are keys or elements from the structure, there exists a set $\mathcal{X}_{\mathtt{A},n}$ of at most $\mathcal{O}(2^\tau)$ array entries, such that A touches only array entries from $\mathcal{X}_{\mathtt{A},n}$, no matter the arguments to A or the content of the data structure.

1.1.5 External memory

As said the RAM-model describes well the time complexity of algorithms on modern computation devices. However there are a number of differences between an actual computer and a RAM-model machine that causes some analysis to be misaligned with reality. The most pronounced is caching. In the RAM-model any operation is unit cost. However the real memory of a computer is divided into several caches, and performing operations on elements that reside in a cache closer to the CPU than one further away can be orders of magnitude faster.

For the machine apex[1], we have experimentally found the cache hierarchy to have the following properties:

Name	Size	Access time	Block size	Throughput
Register	3 KiB[2]	0.29 ns	8 B	26 GiB/s
L1	32 KiB[2]	0.57 ns	64 B	13 GiB/s
L2	256 KiB[2]	1.22 ns	64 B	13 GiB/s
L3	8192 KiB	3.75 ns	64 B	10 GiB/s
RAM	48 GiB	9.16 ns	4 KiB	8 GiB/s
Disk	21 TiB	9.4 ms	2 MiB	447 MiB/s

An important thing to notice here is that as we move away from the CPU the access time becomes drastically worse while the throughput does not drop nearly as much. This high throughput is obtained by transferring large blocks at the same time, for instance in order to get the 447 MiB/s throughput for the disk we need to transfer the data in continuous chunks of 2 Megabytes each.

This is reflectd in the I/O model (also called the External Memory Model) by Aggarwal and Vitter [5], where we consider the computer as consisting of an external disk of unbounded size, and an internal memory of size M. Data can be transferred between disk and memory in chunks of size B. Each such transfer is called an I/O

[1] The machine used for all experiments in this thesis. It has a 3.2GHz four-core Xeon CPU (W3565), 48GiB of memory and 20 3TiB of disks in two raids.

[2] The registers, L1 and L2 cache are per core, while the L3, RAM and disk are shared between CPU cores.

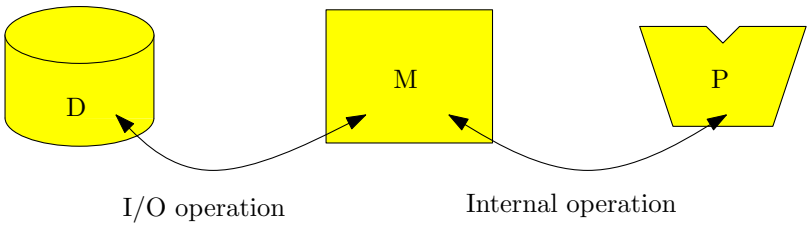


Figure 1.1: In the I/O model of computation data is transferred between the dist (D) and the main memory (M), in blocks of size B.

operation, and we measure the efficiency of an algorithm by the number of I/O operations it performs. See Figure 1.1.

To scan an array of N elements we need $\mathcal{O}(\text{scan}(N))$ I/O operations, where $\text{scan}(N) = N/B$, and to sort a set of N elements we need $\mathcal{O}(\text{sort}(N))$ I/O operations, where $\text{sort}(N) = N/B \log_{M/B} N/B$. The $\text{scan}(N)$ bound is achieved by loading data from a stream one block at a time, processing each item in the block one at a time, and writing the resulting block to another stream. The $\mathcal{O}(\text{sort}(N))$ bound can be achieved by external merge sort. First elements are read in M at a time and sorted internally, yielding N/M individually sorted chunks of size M. Next these chunks are merged in a M/B way merge sort. In each merge a block of size B of each of the M/B streams being merged is loaded in memory, so that the I/O complexity of each merge is $\mathcal{O}(\text{scan}(K))$, where K is the number of items being merged.

While the I/O model is described in terms of moves between the RAM and the disk, the model can also be used to describe cache faults between the CPU and L1 or between any other adjacent entries in the cache hierarchy.

1.1.6 Cache Oblivious

One problem with the External Memory Model is that it only works for one cache level at a time. While we could have different M's and B's for different cache levels, and designing the algorithm knowing these M's and B's, this quickly becomes very tedious. However if we could design an algorithm that does not know M and B and still do an analysis in terms of M and B with good I/O performance, then the algorithm would work well for all possible levels on all possible cache hierarchies all at once, even for crazy machines not yet invented. This model (with some more technical details to be satisfied) is know as the *cache-oblivious* model [37]. However there is one important detail to make this work. When creating an algorithm in the I/O model you have to carefully handle when exactly what blocks are transferred and when blocks are dropped from the memory (we call this a caching strategy). When you analyse your algorithm you prove that your caching strategy works. However in the cache-oblivious model the algorithm cannot know M and B so you cannot hope to implement a good caching strategy in terms of M and B in your algorithm. To get around this we assume that the computer uses an optimal caching strategy for each cache level, this can be justified by the fact that LRU is constant competitive. Now to do the analysis we just have to invent a caching strategy that we could have used, and prove that this strategy is good enough. If it is, then surly the optimal

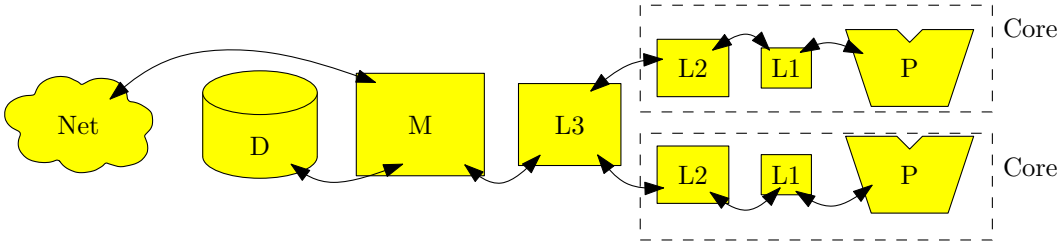


Figure 1.2: The cache levels of a normal two core computer.

strategy will be at least as good.

As in the I/O model scanning can trivially be done in $\mathcal{O}(\text{scan}(N))$ I/Os, and sorting can be done in $\mathcal{O}(\text{sort}(N))$ though this is somewhat more complicated [37].

The cache oblivious model is a relatively good fit for the cache layout of a normal computer as depicted in Figure 1.2.

1.2 Range mode

The first problem I looked at is the so called range mode problem. Given a multi-set of elements, an obvious question to ask is which element is the most common. This most common element is called the *mode* of the set. In the range mode problem we are given an sequence of elements, say $e_1, \ldots, e_n$. We want to construct a data structure, such that given two indices $i \leq j$ we can quickly find the mode of $e_i, \ldots, e_j$.

The mode is together with the median and mean a common statistical measure. Range mode has many useful application, for instance a sales company could have a list of sales ordered by date, and could be interested in what the most sold item is within a certain time span is.

A simplification we can make, is to assume that $e_1, \ldots, e_n$ are numbers in the range from 1 to n. We can make this simplification since we can map an arbitrary input into this range (e.g. using hashing) before constructing our data structure. This will take expected linear time (however we do not care much about the preprocessing times). Once the data structure for this simplified problem answers a range mode query we can then map the number back to the original element using an array lookup in constant time. This approach will only give us an $\mathcal{O}(n)$ space overhead.

As for most data structure problems, solutions for the range mode problem provide a tradeoff between query-time and storage-space. Here we have two obvious solutions at either end of the spectrum. First we could store the answer to all n^2 possible queries in a giant table, and look them up in constant time. In the other end of the spectrum we could walk trough the range, and use bucket counting to get an $\mathcal{O}(n)$ time query while using linear space.

In order to improve these trivial solutions, one can observe that the mode of $e_i, \ldots, e_j$ is either e_i or the mode of $e_{i+1}, \ldots, e_j$. Given an element e we can, in $\mathcal{O}(\log n)$ time, determine how many times it occurs in the range from i to j. This can be done by constructing a table T_e with the indices of all occurrences of e in e_1 to e_n in the preprocessing phase. Then when answering a query, doing a binary search for j and i in T_e, and subtracting the returned positions.

This immediately gives a solution with linear space and $\mathcal{O}(\sqrt{n}\log n)$ query time (see Figure 1.3). Simply divide $e_1, \ldots, e_n$ into $\sqrt{n}$ slabs and store the answer for

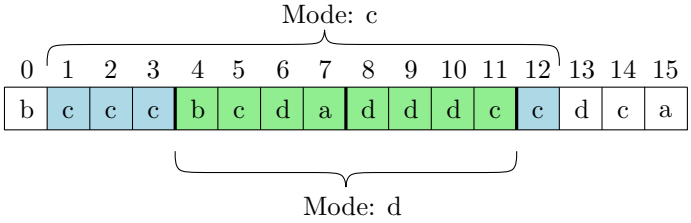


Figure 1.3: Example of mode computation. Here the mode from 1 to 12 is computed by first looking up the mode of the largest contained multislab 4 to 11 (in green) which is d, and comparing its frequency with the frequency of all the remaning elements (in blue). Here the frequency of d in the range from 1 to 12 is 4 while the frequence of c in the same range is 6, and the mode is therefore c.

Reference	**Space**	**Query**	**Conditions**
Krizanc *et al.* [48]	$\mathcal{O}(n^2 \log\log n/\log n)$	$\mathcal{O}(1)$	
Petersen [61]	$\mathcal{O}(n^2/\log n)$	$\mathcal{O}(1)$	
Petersen *et al.* [62]	$\mathcal{O}(n^2 \log\log n/\log^2 n)$	$\mathcal{O}(1)$	
Krizanc *et al.* [48]	$\mathcal{O}(n^{2-2\varepsilon})$	$\mathcal{O}(n^\varepsilon \log n)$	$0 < \varepsilon \leq \frac{1}{2}$
Petersen [61]	$\mathcal{O}(n^{2-2\varepsilon})$	$\mathcal{O}(n^\varepsilon)$	$0 < \varepsilon \leq \frac{1}{2}$
Krizanc *et al.* [47]	$\mathcal{O}(n)$	$\mathcal{O}(\sqrt{n}\log\log n)$	
Chan *et al.* [26]	$\mathcal{O}(n)$	$\mathcal{O}(\sqrt{n/\log n})$	

Table 1.1: The operation time, and space overhead of important structures for the range mode problem.

every multislab (i.e. store the answer for every query starting and ending at slab boundaries). To answer a query i, j, find the mode of the largest contained multislab, and compare the frequency of this element to the at most $2\sqrt{n}$ elements not covered by the multislab. The element that has the largest frequency in the range from i to j, is the mode for the range i to j.

Using these insights several data structures have been developed, and log factors shaved off; see Table 1.1 for an overview of the main results. All these query times are quite high, however in [26], Chan *et al.* presented strong evidence that a query time significantly below $\sqrt{n}$, for linear space, cannot be achieved by purely combinatorial techniques, since boolean matrix multiplication of two $\sqrt{n} \times \sqrt{n}$ matrices reduces to n range mode queries in an array of size $\mathcal{O}(n)$.

Due to the relatively long query time for linear spaced data structures we decided to look into approximations of the mode. In the c-approximate range mode problem, a query is given indices i and j, and we must return an element with a frequency at least $\frac{1}{c}$ times that of the mode. This problem was first considered by Bose *et al.* [18]. With constant query time, they solve 2-approximate range mode with $\mathcal{O}(n \log n)$ space, 3-approximate range mode with $\mathcal{O}(n \log\log n)$ space, and 4-approximate range mode with linear space. For $(1+\varepsilon)$-approximate range mode, they describe a data structure that uses $\mathcal{O}(\frac{n}{\varepsilon})$ space and answers queries in $\mathcal{O}(\log\log_{(1+\varepsilon)} n) = \mathcal{O}(\log\log n + \log\frac{1}{\varepsilon})$ time. In Chapter 2 we present a simple data structure for the 3-approximate range mode with linear space, and constant query time. We then use this to construct a structure for $(1+\varepsilon)$-approximate range mode. This structure uses $\mathcal{O}(\frac{n}{\varepsilon})$ space and answers queries in $\mathcal{O}(\log\frac{1}{\varepsilon})$ time. This gives a

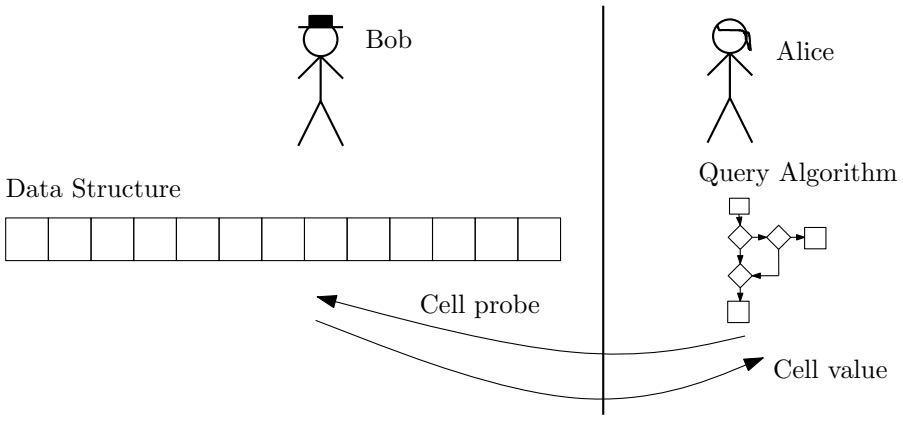


Figure 1.4: Range mode reduction.

linear space, constant time data structure for c-approximate range mode for every constant $c > 1$.

In Chapter 2 we also give a lower bound for the range mode problem in the cell probe model. We prove that any data structure that uses S cells and supports range mode queries must have a query time of $\Omega(\frac{\log n}{\log(Sw/n)})$. This means that any data structure that uses $\mathcal{O}(n \log^{\mathcal{O}(1)} n)$ words of space needs $\Omega(\log n / \log \log n)$ time to answer a range mode query, assuming $w = \log^{\mathcal{O}(1)} n$. Similarly, any data structure that supports range mode queries in constant time needs $n^{1+\Omega(1)}$ space.

The lower bound for linear space is somewhat far from the upper bounds of the algorithms, and the conditional lower bound provided in [26], however with current techniques for cell probe lower bounds proving a query time above $\log n$ does not seem feasibly for any data structure problem, let alone range mode.

For the lower bound we use techniques by Pătraşcu and Thorup [60, 58]. Actually our construction proves the same lower bound for queries on the form: Is there an element with frequency at least (or precisely) k in $A[i, j]$, where k is given at query time? In the scenario where k is fixed for all queries it is quite easy to give a linear space data structure with constant query time for determining whether there is an element with frequency at least k. We will store an array X of length n, where $X[i]$ stores the lowest index j such that the frequency in the of the mode of $A[i, j]$ is at least k. The answer to a query (i, j) we just check if $X[i] \leq j$.

The lower bound is achieved by reducing a communication complexity problem know as *blocked lopsided set disjointness* [58] to a sequence of range mode queries. This is done by having Bob create a range mode data structure, and Alice simulating queries on it by sending memory requests to Bob (see Figure 1.4).

1.3 Implicit dictionaries

A problem I have spent quite a lot of time on, is the problem of creating space efficient dictionaries with query dependent properties such as the working-set property or the finger-search property.

A dictionary is a data structure, that stores elements, with associated keys. A dictionary should support some subset of the operations listed in Table 1.2. The first dictionary with decent performance to be discovered was the AVL tree[2], here

Operation	Description
Insert(v)	Insert the element v into the dictionary.
Find(k)	Find and return the element v associated with the key k.
Remove(k)	Remove the element with the associated key k.
Predecessor(k)	Find and return the element v that has the largest associated key that is less than k.
Successor(k)	Find and return the element v that has the smallest associated key that is greater than k.

Table 1.2: Common operations on a dictionary.

all operations described in Table 1.2 run in time $\mathcal{O}(\log n)$ where n is the number of elements currently in the structure. The structure uses $\mathcal{O}(n)$ space. Later other more popular structures such as the red-black tree[13], achieving the same bounds where described.

1.3.1 Access sensitive properties

One interesting dictionary is the splay-tree [65], which supports all the operations in Table 1.2 in time $\mathcal{O}(\log n)$ amortised. While this on the surface sounds worse than that of a red-black tree, the space overhead of the structure is smaller since one does not have to store any rebalancing information (like colour or subtree size). Also the structure is easier to implement since you basically just have to implement the splay operation.

The splay-tree also has other surprising properties. If you perform a sequence of m searches for keys $k_1, \ldots, k_m$, the total time is bounded by $\mathcal{O}(n \log n + \sum_{i=1}^{m} \log \ell_i)$ where ℓ_i is the minimal j such that $k_{i-j} = k_i$, that is the number of searches since k_i was last searched for. This property is called the working set property. The idea is that elements you have accessed recently should be fast to access again.

Another property of the splay tree is the dynamic finger property. If we perform a sequence of m searches for keys $k_1, \ldots, k_m$, we can bound the total time by $\mathcal{O}(n \log n + \sum_{i=1}^{m-1} \log d(k_i, k_{i+1}))$ where $d(k, k',)$ is the rank-distance between k and k'. That is if you search for something that is close to what you just searched for it will potentially be much faster that searching for something far away. For instance if you do a predecessor search for all elements in increasing order x times, the time required is $\mathcal{O}(n \log n + xn)$ instead of the expected $\mathcal{O}(xn \log n)$.

Yet another property of the splay tree is the static finger property, where we pick some element f in the tree. When performing a sequence of m searches for keys $k_1, \ldots, k_m$, we get a total time of $\mathcal{O}(n \log n + \sum_{i=1}^{m} \log d(k_i, f))$, where again $d(k, f)$ is the rank-distance between the element with key k and f.

Given a red-black tree, it is easy to construct a dictionary that achieves the working-set property in a worst case sense such that searching for a key k takes time $\mathcal{O}(\log \ell)$ where ℓ is the number of distinct elements searched for since k was last searched for. The structure, described by Iacono [42], consists of $\log \log n$ red-black trees and dequeues of double exponentially increasing size. In the i'th level we have a red-black tree with 2^{2^i} elements, and a dequeue with the same elements (see Figure 1.5). The idea is that if you concatenate the dequeues from 0 to $\log \log n$ then the elements will appear in order of decreasing access time, such that an element

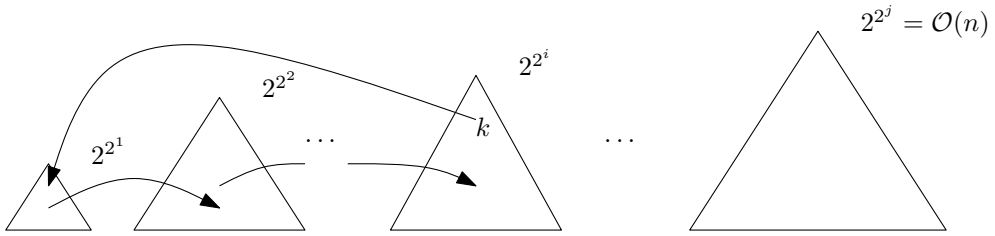


Figure 1.5: Double exponential layout of the Iacono structure.

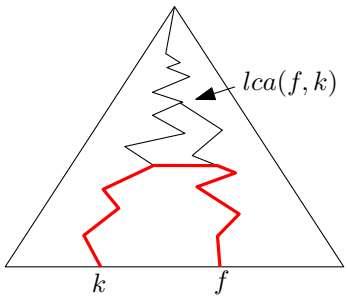


Figure 1.6: Finger search for k from f. The red path is the search path, going up to the level where the ancestors of k and f are adjacent, instead of all the way to the lowest common ancestor.

that has been access more recently will appear before all elements accessed less recently. When searching for an element k we will search the red-black trees in order, until we find the element. If we assume that it was last accessed ℓ searches ago, it must be in one of the first $\log\log\ell$ levels, so the total search time becomes $\mathcal{O}(\sum_{i=0}^{\log\log\ell}\log 2^{2^i}) = \mathcal{O}(\log\ell)$. In the same time it is fairly easy to maintain the order properties under insert, delete, and search. This is done by bumping the oldest element from one level into the next level. All of these other operations will maintain a worst cast time of $\mathcal{O}(\log n)$.

In a similar way it is also quite easy to create a dictionary that has the static finger search property for a given fixed finger f in the worst case sense. Again construct a sequence of $\log\log n$ double exponentially increasing red-black trees. This time keeping elements closer to f in the lower levels. Here the search time will become $\mathcal{O}(\log d(k, f,))$, while all other operations maintain a worst cast time of $\mathcal{O}(\log n)$.

Another way of achieving a worst case finger search property for a dynamic set of fingers of constant size is to use a level linked tree. Here the nodes in the search tree is augmented with pointers to the left and the right cousin. Assume we use a search tree where elements are only stored in leaves, then for a finger f we maintain a pointer to the leaf containing it. When searching for some key k we start in the leaf containing f, and walk up looking at the left and right cousins until we find a sub-tree containing k. This takes $\mathcal{O}(\log d(k, f,))$ worst case time, while the insert and delete time can be maintained at $\mathcal{O}(\log n)$. See Figure 1.6.

1.3.2 Decreasing space usage

While the solutions mentioned above have nice asymptotic running time and space usage, the constant in the space usage can in fact be quite high, and an obvious question to ask is therefore how low can the constant in space usage get? As mentioned in Section 1.1.3 there are two fundamental different ways of going about this. We can assume a specific kind of elements, and provide a space efficient encoding based on the properties of the individual elements and the distribution of the elements, which is done in the succinct models, or we can assume that elements are black box comparable elements, which we will do here. If an element is black box and occupies k bits, then clearly any data structure containing n elements will occupy at least kn bits, and this is indeed what we are going to aim for.

Much work has been going into improving the space overhead of dictionaries. An overview of this work can be found in Table 1.3.

Structure **Reference**	**Additional Space**	**Operation Time**
AVL Tree [2]	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$
Munro and Suwanda [54]	$\mathcal{O}(1)$	$\mathcal{O}(n^{1/3} \log n)$
Frederickson [35]	$\mathcal{O}(1)$	$\mathcal{O}(n^{\epsilon})$ for $\varepsilon > 0$
Implicit AVL Tree [53]	$\mathcal{O}(1)$	$\mathcal{O}(\log^2 n)$
Implicit BTree [34]	$\mathcal{O}(1)$	$\mathcal{O}(\log^2 n / \log \log n)$
FG Dictionary [33]	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$

Table 1.3: Space usage of various dictionaries.

In Chapter 3 we present dictionaries that are both space efficient and have the working-set or finger-search properties. The main tricks in doing this is combining the implicit dictionary of Franceschini and Grossi [33] with the dictionaries of double exponentially increasing size of Iacono [42].

We present an implicit dictionary with the working set property. It uses no additional space, it supports Insert, Delete, Predecessor, and Successor in time $\mathcal{O}(\log n)$, and Find in time $\mathcal{O}(\log \ell)$. Since the dictionary is based on the Franceschini-Grossi Dictionary which is also cache oblivious efficient, the log's can be changed to a $\log_B$ in a cache oblivious analysis of our structure. This structure uses the same bumping strategy as in [42], however instead of keeping track of the age of elements explicitly, we divide the elements within each level into three generations.

After the publication of [19] on which Chapter 3 is partly based, Brodal and Kejlberg-Rasmussen [22] improved upon this result so that also the Predecessor and Successor operations run in time $\mathcal{O}(\log \ell)$.

We present an implicit dictionary with the static finger search property. It again uses no additional space, while it supports Insert, Delete in time $\mathcal{O}(\log n)$, Predecessor, Successor and Find in time $\mathcal{O}(\log t)$, where t is the rank distance between the element returned and the current finger, and change-finger in time $\mathcal{O}(n^{\varepsilon})$ for any $0 < \varepsilon \leq 1$. The change-finger operation changes the finger from its current element f to some new element f'.

In Lemma 3.1 we state that for any algorithm A on a strict implicit data structure of size n that runs in time at most τ, whose arguments are keys or elements from the structure, there exists a set $\mathcal{X}_{\mathtt{A},n}$ of at most $\mathcal{O}(2^{\tau})$ array entries, such that A

touches only array entries from $\mathcal{X}_{\mathsf{A},n}$, no matter the arguments to A or the content of the data structure.

Now assume we want to create a static dictionary with the static finger search property. Here in the weak model, we can just store all elements in a sorted array, and additionally store the index of the finger element, now finger searching can trivially be done in $\mathcal{O}(\log t)$ time using exponential search, and the finger can easily be changed in $\mathcal{O}(\log t)$ time. Nevertheless we show using Lemma 3.1 that in the strong implicit model where we are not allowed to store the index of the finger, changing the finger must take $\Omega(n^\varepsilon)$ time if we want a search time of $\mathcal{O}(\log t)$, or indeed anything less than $\log n$.

Lemma 3.1 gives us a general technique to prove lower bounds for data structures in the implicit model. The technique is only useful in the strict implicit model. As we show, it can be used to prove some polynomial lower bounds, however we have only been able to apply it where at least one of the operations of the structure run in time $\log n$ or below. If all operations run in $\mathcal{O}(\log n)$ time, the lemma just implies that all operations may touch every cell, which does not seem useful. The lemma is based on a very simple counting technique, so its novelty can be discussed. I have not seen similar techniques used in conjunction with the implicit model before, however this is most likely do to the obscurity of the strict implicit model.

The above results show that much of what can be done in the weak implicit model with $\mathcal{O}(1)$ registers between operations, can also be done without these $\mathcal{O}(1)$ registers. While for some problems having the $\mathcal{O}(1)$ extra registers is a must. The only real advantage the strong model has over the weak is composability. If you construct a data structure using more than $\mathcal{O}(1)$ weak implicit data structures, you get a data structure that is not even weakly implicit. While if you compose any number of strong implicit structures the result will also be strongly implicit. However this advantage does not appear in many situations.

Many data structures defined in the weak implicit model can be converted into strong implicit structures. If all operations take at least $\Omega(\log n)$ time, the conversion can be done trivially. For instance given a weak implicit dictionary where all operations take $\Omega(\log n)$ time, we can construct a strong implicit dictionary that supports the same operations within the same time bound. To do this we keep the first $C \log n$ elements outside of the main structure, for some constant C. Between operations on the dictionaries the $\mathcal{O}(1)$ words of memory can be encoded in the order of these $C \log n$ elements, this data can be recovered before doing an operation, and re-encoded after the operation, with only an additional $\mathcal{O}(\log n)$ time overhead. To perform a find operation, we first recover the $\mathcal{O}(1)$ words from the $C \log n$. If the element we are looking for is within the first $C \log n$ elements, we return it, otherwise we query the main structure and return the result. Similar tricks can be done for insert, delete, successor, and predecessor. The same transformation can also be applied to a priority queue where the operations take $\Omega(\log n)$ time.

This implies that the strong implicit model only makes sense when at least one operation runs in $o(\log n)$ time, and since the model as defined here is comparison based, this mostly happens with a fine grained analyses where the query time is dependent on something other than n.

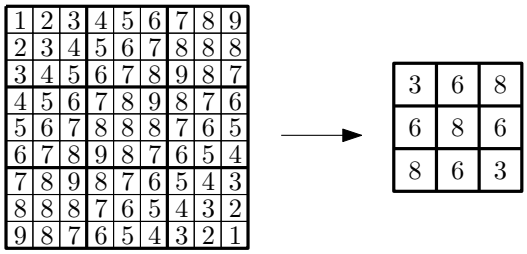


Figure 1.7: Example of down scaling a 9x9 raster by a factor of 3 to a 3x3 raster.

1.4 Multiresolution rasters

At ESA 2013 Constantinos Tsirogiannis presented an algorithm for a problem known as the "multi resolution raster" problem [9]. While this algorithm is good in theory, we afterwards still felt that this problem could get a cleaner and more practical efficient solution.

The problem can be defined as follows. The input is a raster R which consists of $\sqrt{N} \times \sqrt{N}$ cells, that for instance could be an elevation map of a terrain. We want to compute rasters R_s, for all integers $1 < s \leq \sqrt{N}$, where R_s is a s down-sampling of R. That is R_s has size $\lceil \sqrt{N}/s \rceil \times \lceil \sqrt{N}/s \rceil$ and each cell in R_s stores the average value of the corresponding $s \times s$ cells of R. See Figure 1.7 for an example.

Constructing all these scale instances is important for several different applications. In general it is often not obvious at which scales different properties of a terrain emerges. If you zoom out too far you will miss details. If you zoom in too close one feature can be distributed over a large area. For instance Fisher *et al.* [32] construct rasters at many different scales to perform landform classification. Woodcock and Strahler [69] construct many different scale instances in order to estimate the average tree canopi size from a grayscale forest image.

It is not immediately obvious how large the total size of the output rasters are, however assuming we do not output the original raster the total size is:

$$\sum_{i=2}^{\sqrt{N}} \frac{N}{i^2} \leq N(-1 + \sum_{i=1}^{\infty} \frac{1}{i^2}) \leq 0.645N.$$

This problem has a fairly trivial $\mathcal{O}(\text{sort}(N))$ solution, that was first detailed in [9]. First a matrix of prefix sums P is computed, such that any i, j in P contains the sum of every entry i', j' from R where $i' \leq i$ and $j' \leq j$. This way the average value that we need to compute for some entry i, j in an R_s, can be computed from four prefix values, as $R_s[i,j] := (P[si, sj] - P[si, s(j-1)] - P[s(i-1), sj] + P[s(i-1), s(j-1)]) \frac{1}{s^2}$. P can be computed with a scan over R. This immediately gives an internal memory algorithm. Now we iterate over all cells t in all the outputs, and for each consider the four entries $p_1, \ldots, p_4$ from P needed to compute the value of t, while doing this we create a stream S with the pairs (p_i, s). We sort S by p. We now iterate over S and P simultaneously, for every entry (p, s) in S we write a new entry $(p, s, P[p])$ into a stream S'. We sort S' by t. We now iterate over the cells of the output rasters t and for each cell we compute the average value based on the four values forwarded to t through S'.

In [9] a $\mathcal{O}(\text{scan}(N))$ algorithm for the problem was also presented, the algorithm is quite simple; first compute P, then scan over P to compute the B largest out-

puts, $R_2, \ldots, R_B$ all at once. Then produce the smaller outputs $R_{B+1}, \ldots, R_{\sqrt{N}}$, by performing 4 I/Os per cell, to directly read the required entries from P. In total only $\mathcal{O}(\text{scan}(N))$ I/Os are performed, under some assumptions. First it makes the tall cache assumption, that is $M > B^2$, however in order to achieve good I/O performance we often pick B to be of the order of a few megabytes, meaning that we would need terabytes of memory. Secondly it writes to B files simultaneously, however the performance of most operating systems degrades rapidly when writing to more than a few hundred files at a time.

In Chapter 4 we present an $\mathcal{O}(\text{scan}(N))$ algorithm for the problem. This algorithm only assumes that a constant number of files can be used at the same time, and it does not need the tall cache assumption. Furthermore the algorithm is cache-oblivious and experimentally shown to be practically efficient. The algorithm from Chapter 4 was compared with the $\mathcal{O}(\text{sort}(N))$ algorithm from [9] by performing runs on a large dataset. Here the former took 2 hours and 15 minutes while the latter took 13 hours and 14 minutes.

The algorithm works by deriving smaller rasters from larger ones. The main observation is that instead of creating the 1/18 scale raster by processing the full input raster, it can be created from the 1/9 scale raster. By always deriving a scale raster from the smallest possible scale raster and by deriving multiple rasters at the same time an efficient algorithm can be constructed.

1.5 Decomposition simplification

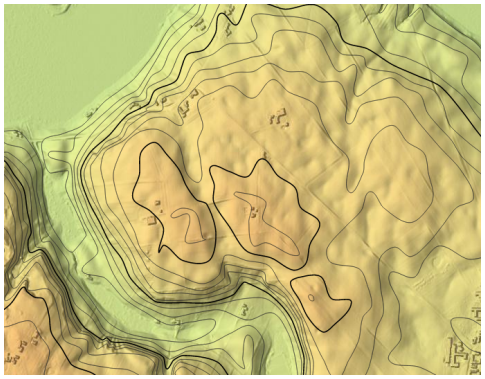
The problem on which I have used the most time during my PhD studies, is the problem of simplifying large planar decompositions.

A planar decomposition is the division of the plane (often the surface of the earth) into disjoint faces. There are many different kinds of planar decompositions that have practical value, some common ones are depicted in Figure 1.8.

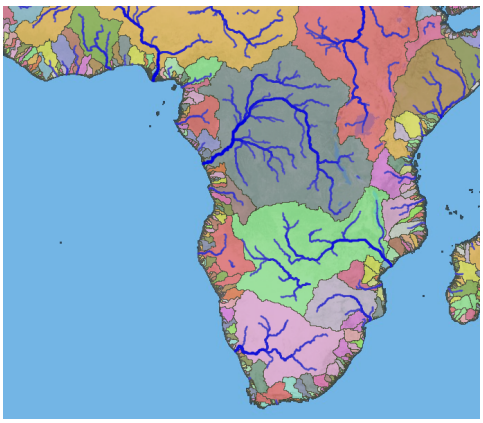
Input We will now go into detail with some of the planar decomposition instances and how they are computed.

A contour map is a vector description of the elevation of a terrain. A fixed set of heights are selected (for instance every multiple of 2 meters), and all points on the terrain with one of the selected heights are extracted, if the terrain is reasonable well behaved, the extracted points will form a number of curves. These curves can be used for all kinds of things ranging from navigation to environmental impact assessments. Before the introduction of *Light Detection and Ranging* (LIDAR) contour curves where drawn by hand, sparse height samples of the terrain where collected by surveyors, these points where positioned on a large sheet of paper, and a cartographer would draw nice smooth curves between the points. With LIDAR technology and digital computers, a more precise approach is possible. A detailed height model is collected by flying a plane at low altitude (on a nice summer day, with clear skies). On the plane a LIDAR device is attached which uses a number of lasers to accurately measure points on the ground. Currently a new model of Denmark is being flow, where on average more than 5 points are collected for every square meter of surface. A surface model can then be constructed from this massive point cloud. A common way of doing this is to project the points into the XY plain, compute a Delaunay triangulation, and then rising this triangulation back up to 3D. The surface model

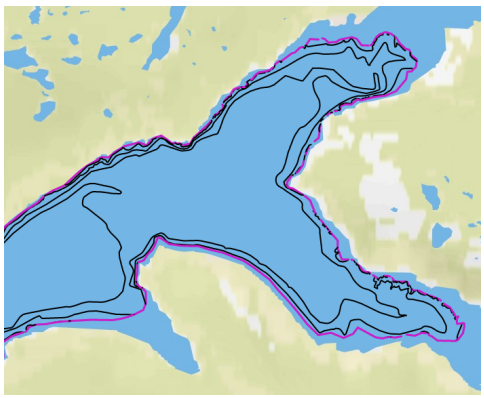
(a) **Cadastre**: A cadastral map is the decomposition of the plane into land parcels.



(b) **Contours**: A contour map is the decomposition of the plane into regions with a bounded height interval.



(c) **Watershed**: A watershed map is the decomposition of the plain into regions that all drain into the ocean at the same point, or all drain into the same river.



(d) **Nautical chart**: A nautical chart, is similar to a contour map but depicts under sea terrain. Since it is used for safe navigation other considerations have to be made while simplifying.

Figure 1.8: Examples of planar decompositions

thus obtained is called a TIN (Triangulated irregular network). The most obvious way to get a contour map from such a tin (and indeed the way we use) is to compute the intersection of all the triangles with the contour planes (as seen in Figure 1.9). That is all the horizontal planes that go through the wanted contour heights. The curves thus generated are very accurate, and almost completely unusable; all the details hide what you really want to see. Clearly simplification is required, what kind of simplification and with what parameters depends on the particular usage and scale of the map.

A watershed map is the decomposition of the terrain into sections where rain goes into the same river. Traditionally such a map was drawn by tracing ridges in the terrain, everything on one side of a ridge flows into one river while everything on the other side flows into another. Such maps can again be used for many different proposes from predicting flooding based on precipitation to examining the impact of a pollution event. Again with newer technology more detailed and accurate maps can be computed automatically. First a TIN is constructed as described above. Next

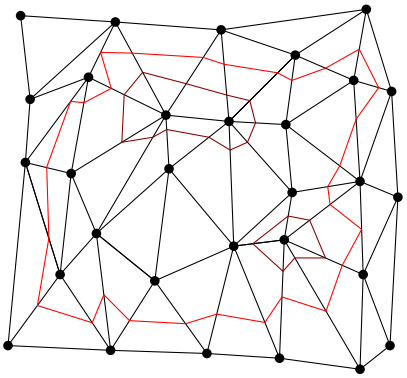


Figure 1.9: Contour segments are generated from a tin by cutting the individual triangles with the contour planes.

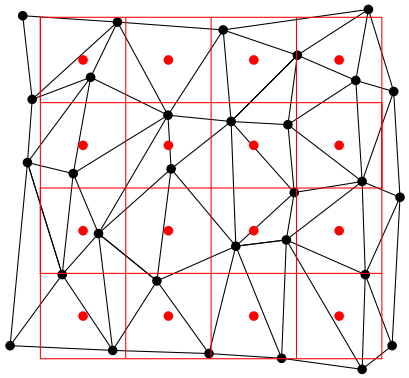


Figure 1.10: A raster is constructed from a tin by interpolating the heights at the center points of the raster cells.

a raster DEM (Digital elevation model) is constructed, often by simply imposing a raster on top of the TIN and assigning the height under the center of the raster cell in the TIN, to that cell (See Figure 1.10). For the new Denmark dataset currently being created, a raster with a cell size of 50cm by 50cm will be constructed. The DEM is then processed using topological simplification to remove smaller depressions in the terrain [31, 4]. A flow directions raster can be computed on this simplified DEM; for every cell one computes which direction the water would flow from that cell by comparing the heights of the neighbouring cells and the current cell, and assigning the flow to the lowest neighbour. Once this is done, time forward processing can be used to compute which cells flow into which rivers. The output of this is a raster, where every cell contains the id of the river it flows into. This raster can then be processed to obtain a vector decomposition, by outputting a segment along the boundary between every adjacent pair of cells, that do not share the same id. Again the generated watershed, while very accurate, is way too detailed for practical use.

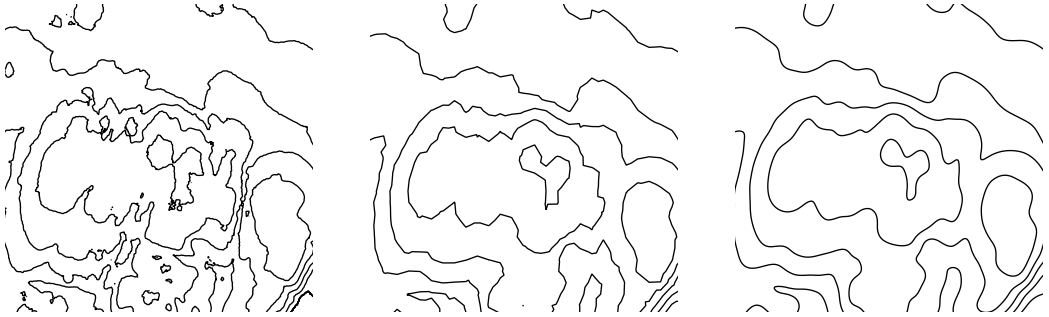
1.5.1 Terrain Simplification

When simplifying planar decompositions there are two fundamentally different approaches that can be used independently (or together). Either you simplify the input model before generating the decomposition or you simplify the decomposition after generating it.

Simplifying the model before decomposition has not been the focus of any of my work during my PhD, however I will briefly mention some of the most common approaches used for simplifying input for contour generation.

The method of terrain simplification most often used for contour generation is running a gaussian kernel over the terrain. Here every pixel in the output terrain is creating by sampling the corresponding neighbouring pixels in the input according to a normal distribution, which yields a nice smooth terrain. See Figure 1.11 for an example of contours generated using this approach. This approach can be refined to provide properties needed for the particular contours. An example of this is a method the Danish cadastral agency (GST) has used for generating nautical charts.

A nautical chart is a document used for navigation which consists of contour lines



(a) A Raw, unsimplified contour map.

(b) Generated from an aggregated raster.

(c) Generated from a raster smoothed with a gaussian kernel.

Figure 1.11: Examples of contours generated from rasters simplified using simple mean aggregation or smoothed using a gaussian kernel.

and plotted measured points (among lots of other things). If you put a -20 meter contour line in the map, it is vitally important that there is at least 20 meters of water underneath the line in real life, since ships would otherwise run aground (and the cadastral agency would be legally liable for the damages). This naturally leads to the following constraint for a simplified input S model used for construction of curves: the elevation of every point in S must be lower than the elevation at the same point in the actual height model H. Let $G(T, \rho)$ be the gaussian smoothing of some terrain T with standard deviation ρ. Then an input terrain for curve generation is generated by the following process. First a standard deviation ρ is picked that gives a nice smooth terrain, and a constant $c < 1$ is picked, say $c = 0.9$.

Now we compute $S_0 = G(H, \rho)$, as our initial estimate of the smoothed terrain, which might be too high at some points. We therefore define $D_i = \max(S_i - H, 0)$ to measure how much too high S_i is. We can now compute a new refinement $S_{i+1} = S_i - G(D_i, \rho c^i)$, where we subtract a smoothed version of how much too high we where; We will continue this process with ever smaller deviation, until a sufficiently high i say n where $\rho c^i \leq 1$ and finally define the output $S = \max(S_n, H)$. While this technique produces mostly nice and smooth curves, it was eventually dropped because there are (many) other requirements for the curves on nautical charts that are almost impossible to fulfill on a method only modifying the input terrain.

A very simple approach that is also used sometimes (depending on how you view the world) is aggregation. Here the input terrain is simplified by aggregation. Here the raster is made smaller by replacing a window of cells by their average, as explained in Section 1.4. While this gives smaller curves in terms of segments, the curves are not very pretty, and do not necessarily represent reality very well. An example of this is shown in Figure 1.11.

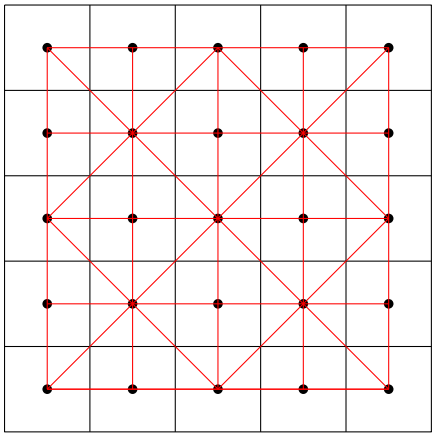


Figure 1.12: The raster (black) is triangulated (red).

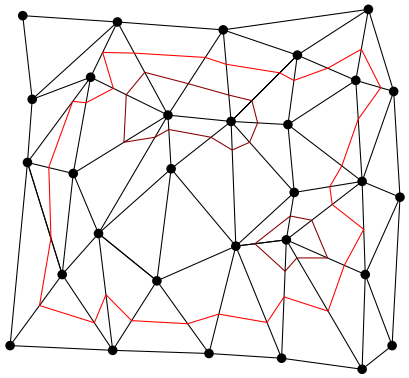


Figure 1.13: Contour segments are generated from a tin by cutting the individual triangles with the contour planes.

1.5.2 Initial results

My first work on decomposition simplification was in [8]. I have not included it in this thesis since it has (in my view) been entirely superseded by [11]. I will though here give a fairly detailed overview of the results and their production.

The purpose of [8] is to produce simplified contour lines of an input terrain given as a TIN or a raster, under a given set of constraints. The focus of the paper is on how to do this I/O efficiently under some realistic assumptions. The focus is not on how to make the curves look pretty. The algorithm works in a number of phases. First the input is converted into triangles which are cut with the contour planes to get a soup of contour segments. The segments are then processed in order to remove ridges such that contour lines form nice rings (every vertex has even degree). The segments are then labelled such that every connected component of segments has the same label. This labelling is done in a way such that for every ring enclosed by another ring, the enclosed will have a higher label than the enclosee. The rings are then relabelled as detailed below. Finally each ring is simplified by an internal algorithm in turn by label order.

The algorithm will simplify a contour map in $\mathcal{O}(\text{sort}(n))$ I/Os, under some realistic assumptions. First we assume that for any horizontal line, all the segments intersected by the line fits in internal memory. Secondly for every face in the decomposition, we assume that all segments adjacent to that face, fits in internal memory.

Before going into details, let me define some notation. A segment is a two dimensional line segment. A chain is a sequence of segments, where each consecutive pair of segments share an endpoint. A ring is a chain where the first and the last segment also share an endpoint.

If the input is a raster, it is converted into a triangulation, as shown in Figure 1.12. If on the other hand the input is a TIN, the triangles are extracted from the TIN (using three sorting steps). In either case the triangles are visited in the order of extraction, and segments are generated by cutting with planes in all the contour heights, as shown in Figure 1.13. Special care is taken such that the intersection points are computed to exactly the same coordinates on adjacent triangles. The details of how to get this right are quite tricky, especially when dealing with

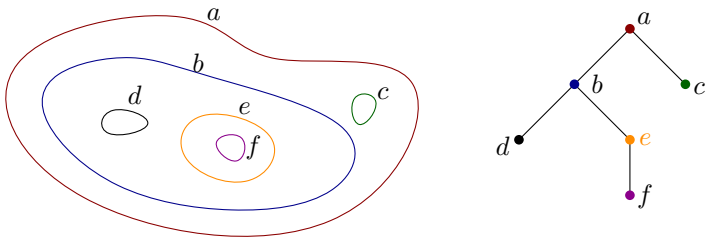


Figure 1.14: Example of ring nesting, on the left the contours have been labelled, and on the right the corresponding ring nesting tree.

floating point coordinates. However it is doable.

The segments are ordered by their minimal y coordinate (and using the other coordinates as tie breakers) by an external sort. The segments are then scanned and if the same segment appears twice both appearances are removed since the segment must then be a ridge segment (assuming the segment generation is done somewhat cleverly).

Next we label the segments such that each ring gets a separate label. The labelling algorithm is inspired by an algorithm from [10], and explained in more detail in Chapter 5. The algorithm is based on a sweepline approach, and in order to be I/O efficient requires that all segments intersecting the sweepline at any particular point in time fits into the internal memory (for realistic datasets this means that $\sqrt{N} = \mathcal{O}(M)$). The algorithm performs two sweeps through the data, first a sweep from the bottom to the top (the up sweep), where connections below the sweepline are maintained, and then a sweep from the top to the bottom (the down sweep), where the connections above the sweepline are combined with the connections below.

The nesting of the rings form a tree, such that a ring a is a descendent of a ring b if b is contained within a (See Figure 1.14). When simplifying a ring a, all segments on the two faces on which the ring is situated need to be considered in order to maintain homotopy. These segments are exactly the segments on the parent, siblings and child rings of a in the nesting tree (In the example of Figure 1.14, the rings a,c,d and e need to be considered when simplifying b).

The labelling algorithm generates labels such that a ring with a lower minimal y coordinate gets a smaller label than one with a higher y coordinate. Yet in order to facilitate simplification we need a different order of labels, specifically we want the labelling to be a level ordering of the nesting tree. That is, given two distinct rings a and b, we require that the labels of a and the labels of b be different. If a and b are on different levels in the tree, the ring on the lower level has to have lowest label. If a and b are on the same level, then the ordering on the labels of a and b must be the same as the ordering of the labels of their respective parents. This relabeling can be done I/O efficiently, as described in [8], however it is somewhat tricky to implement.

We will simplify contour rings one at a time, in a level order traversal of the nesting tree. When we simplify a ring a, we will already have simplified its parent and some of its siblings. In order to maintain homotopy of the final output, we will consider these simplified versions of the parent and siblings as we are simplifying a. We will do the actual simplification internally by assuming that for any ring a, all the edges of it, its parent, its siblings and its children fit in memory at the same time.

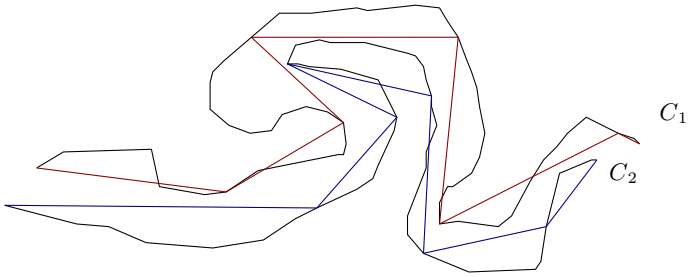


Figure 1.15: When simplifying two chains independently the simplified chains cannot intersect

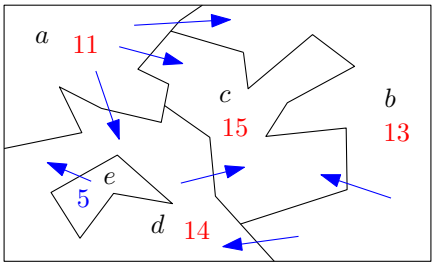


Figure 1.16: Example of segment forwarding, the size of each face (the number of adjacent segments) is written in red. We see that a forwards its segments to both b, c and d, so in total a forwards 33 segments.

The internal simplification we used in [8] is a constraint variant of the Douglas Peucker algorithm [29]. The homotopy constraint is maintained by tracing both the original and the proposed simplified rings in a trapezoidal decomposition, as also described in Chapter 5.

1.5.3 A simpler approach

In Chapter 5 which is based on [11], we present an algorithm for the more general problem of decomposition simplification. That can be seen as an evolution of [8], that solves a harder problem in an easier way. One of the main complications in implementing [8] is that we need to process everything in a specific order, such that we can use the all ready simplified version of one contour level, as a constraint for the next level. If we where somehow able to simplify one ring independently from the other, the algorithm would be much simpler, also it would work on general decompositions where defining a level ordering is not possible. It turns out that this is possible, and in fact it works without changing anything. This is due to the main observation of Chapter 5. Given two planar chains C_1 and C_2, if you simplify C_1 by choosing a subset of its points, such that it does not intersect C_2, and you simplify C_2 by choosing a subset of its points, such that it does not intersect C_1, then the two simplification cannot intersect, see an example in Figure 1.15. This is proved in Lemma 5.3.

For the case of contours this gives rise to a simple algorithm. First generate the contour segments like in [8]. Instead of labelling the rings, we will label the faces and augment every segment with the labels of the two faces it is attached to. It turns out that it is quite simple to adapt the labelling algorithm of [8] to do this. Note that when simplifying a contour ring, the segments needed to constrain

the ring, are the segments of the two faces the contour is situated between. To do the simplification in an I/O efficient manner, we let every face forward all its segments to every neighbour it has, which is larger than itself in terms of segments (see Figure 1.16 for an example). This way the larger neighbour can then simplify the shared ring in memory. The in memory simplification can be done in the same way as [8].

To prove that the algorithm is I/O efficient, we need to bound the number of segments forwarded from smaller to larger neighbours. In Theorem 5.6 we prove that this is at most 3 times the number of segments in the graph.

The algorithm will simplify a planar decomposition in $\mathcal{O}(\text{sort}(n))$ I/Os, under some realistic assumptions. First we assume that for any horizontal line, all the segments intersected by the line fit in internal memory. Secondly for every two adjacent faces in the decomposition, we assume that all segments touching these two faces, fit in internal memory at the same time.

2

Range Mode

This chapter contains the paper "Cell Probe Lower Bounds and Approximations for Range Mode"[39], which is joint work with Mark Greve, Allan Grønlund Jørgensen and Kasper Dalgaard Larsen. This chapter differs from the paper only in notation and other small textual changes.

2.1 Introduction

In this chapter we consider the range mode problem, the range k-frequency problem, and the c-approximate range mode problem. The frequency of a label ℓ in a multiset H of labels, is the number of occurrences of ℓ in H. The mode of H is the most frequent label in H. In case of ties, any of the most frequent labels in H can be designated the mode.

For all the problems we consider the input is an array A of length n containing labels. For simplicity we assume that each label is an integer between one and n. In the range mode problem, we must preprocess A into a data structure that given indices i and j, $1 \leq i \leq j \leq n$, returns the mode, $M_{i,j}$, in the subarray $A[i,j] = A[i], A[i+1], \ldots, A[j]$. We let $F_{i,j}$ denote the frequency of $M_{i,j}$ in $A[i,j]$. In the c-approximate range mode problem, a query is given indices i and j, $1 \leq i \leq j \leq n$, and returns a label that has a frequency of at least $F_{i,j}/c$. In the range k-frequency problem, a query is given indices i and j, $1 \leq i \leq j \leq n$, and returns whether there is a label occurring exactly k times in $A[i,j]$.

For the upper bounds we consider the unit cost RAM with word size $w = \Theta(\log n)$. For lower bounds we consider the cell probe model of Yao [70]. In this model of computation a random access memory is divided into cells of w bits. The complexity of an algorithm is the number of memory cells the algorithm accesses. All other computations are free.

Previous Results. The first data structure supporting range mode queries in constant time was developed in by Krizanc *et al.* [48], this data structure uses $\mathcal{O}(n^2 \log\log n/\log n)$ space. This was subsequently improved to $\mathcal{O}(n^2/\log n)$ space in [61] and finally to $\mathcal{O}(n^2 \log\log n/\log^2 n)$ in [62]. For non-constant query time, the first data structure developed uses $\mathcal{O}(n^{2-2\varepsilon})$ space and supports queries in $\mathcal{O}(n^\varepsilon \log n)$ time, where $0 < \varepsilon \leq \frac{1}{2}$ is a query-space tradeoff constant [48]. The query time was later improved to $\mathcal{O}(n^\varepsilon)$ without changing the space bound [61].

Given the rather large bounds for the range mode problem, the approximate variant of the problem was considered in [18]. With constant query time, they solve 2-approximate range mode with $\mathcal{O}(n \log n)$ space, 3-approximate range mode

with $\mathcal{O}(n \log\log n)$ space, and 4-approximate range mode with linear space. For $(1+\varepsilon)$-approximate range mode, they describe a data structure that uses $\mathcal{O}(\frac{n}{\varepsilon})$ space and supports queries in $\mathcal{O}(\log\log_{1+\varepsilon} n) = \mathcal{O}(\log\log n + \log\frac{1}{\varepsilon})$ time. This data structure gives a linear space solution with $\mathcal{O}(\log\log n)$ query time for c-approximate range mode when c is constant. There are no known non-trivial lower bounds for any of the problems we consider.

Our Results. In this chapter we show the first none trivial lower bounds for range mode data structures and range k-frequency data structures and provide new upper bounds for the c-approximate range mode problem and the range k-frequency problem.

In Section 2.2 we prove our lower bound for range mode data structures. Specifically, we prove that any data structure that uses S cells and supports range mode queries must have a query time of $\Omega(\frac{\log n}{\log(Sw/n)})$. This means that any data structure that uses $\mathcal{O}(n \log^{\mathcal{O}(1)} n)$ space needs $\Omega(\log n / \log\log n)$ time to answer a range mode query. Similarly, any data structure that supports range mode queries in constant time needs $n^{1+\Omega(1)}$ space.

We suspect that the actual lower bound for near-linear space data structures for the range mode problem is significantly larger. However a fundamental obstacle in the cell probe model is to prove lower bounds for static data structures that are higher than the number of bits needed to describe the query. The highest known lower bounds are achieved by the techniques in [60, 58] that uses reductions from problems in communication complexity. We use this technique to obtain our lower bound and our bound matches the highest lower bound achieved with this technique.

Actually our construction proves the same lower bound for queries on the form, is there an element with frequency at least (or precisely) k in $A[i,j]$, where k is given at query time. In the scenario where k is fixed for all queries it is quite easy to give a linear space data structure with constant query time for determining whether there is an element with frequency at least k. We will store an array X of length n, where $X[i]$ stores the lowest index j such that the frequency in the of the mode of $A[i,j]$ is at least k. The answer to a query (i,j) is just $X[i] \leq j$. In Section 2.3 we consider the case of determining whether there is an element with frequency exactly k, which we denote the range k-frequency problem. To the best of our knowledge, we are the first to consider this problem. We show that 2D rectangle stabbing reduces to range k-frequency for any constant $k > 1$. This reduction proves that any data structure that uses S space, needs $\Omega(\log n / \log(Sw/n))$ time for a query [58, 57], for any constant $k > 1$. Secondly, we reduce range k-frequency to 2D rectangle stabbing. This reduction works for any k. This immediately gives a data structure for range k-frequency that uses linear space, and supports queries in optimal $\mathcal{O}(\log n / \log\log n)$ time [45] (we note that 2D rectangle stabbing reduces to 2D range counting). In the restricted case where $k = 1$, this problem corresponds to determining whether there is a unique label in a subarray. The reduction from 2D rectangle stabbing only applies for $k > 1$. We show, somewhat surprisingly, that determining whether there is a label occurring exactly twice (or $k > 1$ times) in a subarray, is exponentially harder than determining if there is a label occurring exactly once. Specifically, we reduce range 1-frequency to four-sided 3D orthogonal range emptiness, which can be solved with $\mathcal{O}(\log^2 \log n)$ query time and $\mathcal{O}(n \log n)$ space by a slight modification of the data structure presented in [3].

In Section 2.4 we present a simple data structure for the 3-approximate range mode problem. The data structure uses linear space and answers queries in constant time. This improves the best previous 3-approximate range mode data structures by a factor $\mathcal{O}(\log\log n)$ either in space or query time. With linear space and constant query time, the best previous approximation factor was 4. In Section 2.5 we use our 3-approximate range mode data structure, to develop a data structure for $(1+\varepsilon)$-approximate range mode. This data structure uses $\mathcal{O}(\frac{n}{\varepsilon})$ space and answers queries in $\mathcal{O}(\log\frac{1}{\varepsilon})$ time. This removes the dependency on n in the query time compared to the previously best data structure, while matching the space bound. Thus, we have a linear space data structure with constant query time for the c-approximate range mode problem for any constant $c > 1$. We note that we get the same bound if we build on the 4-approximate range mode data structure from [18].

Later Results. After the publication of [39], on which this chapter is based, Chan *et al.* [26], provided linear space data structure supporting range mode queries in time $\mathcal{O}(\sqrt{n/\log n})$. They also provide an argument for the hardness of range mode. They show that boolean matrix multiplication of two $\sqrt{n}\times\sqrt{n}$ matrices reduces to n range mode quires in an array of length $\mathcal{O}(n)$. Thus constructing a data structure for range-mode with a query time below $\mathcal{O}(n^{1/2-\varepsilon})$ for some $\varepsilon > 0$ would yield an improvement for non algebraic boolean matrix multiplication.

2.2 Cell probe lower bound for range mode

In this section we show a query lower bound of $\Omega(\log n/\log(Sw/n))$ for any range mode data structure that uses S space for an input array of size n. The lower bound is proved for the slightly different problem of determining the frequency of the mode. The lower bound for range mode follows since the frequency of an element in any range can be determined in $\mathcal{O}(\log\log n)$ time by a linear space data structure . This data structure stores a linear space static rank data structure [67] for each label ℓ in the input, containing the positions in A storing ℓ. The frequency of a label in $A[i,j]$ is the rank difference between $i-1$ and j.

Communication Complexity and Lower Bounds. In communication complexity we have two players Alice and Bob. Alice receives as input a bit string x and Bob a bit string y. Given some predefined function, f, the goal for Alice and Bob is to compute $f(x,y)$ while communicating as few bits as possible.

Lower bounds on the communication complexity of various functions have been turned into lower bounds for static data structure problems in the cell probe model. The idea is as follows [51]: Assume we are given a static data structure problem and consider the function $f(q,D)$ that is defined as the answer to a query q on an input set D for this problem. If we have a data structure for the problem that uses S memory cells and supports queries in time t we get a communication protocol for f where Alice sends $t\log S$ bits and Bob sends tw bits. In this protocol Alice receives q and Bob receives D. Bob constructs the data structure on D and Alice simulates the query algorithm. In each step Alice sends $\log S$ bits specifying the memory cell of the data structure she needs and Bob replies with the w bits of this cell. Finally, Alice outputs $f(q,D)$. Thus, a communication lower bound for f gives a lower bound tradeoff between S and t.

This construction can however only be used to distinguish between polynomial and superpolynomial space data structures. Since range mode queries are trivially solvable in constant time with $\mathcal{O}(n^2)$ space, we need a different technique to obtain lower bounds for near-linear space data structures. Pătraşcu and Thorup [60, 58] have developed a technique for distinguishing between near linear and polynomial space by considering reductions from communication complexity problems to k parallel data structure queries. The main insight is that Alice can simulate all k queries in parallel and only send $\log \binom{S}{k} = \mathcal{O}(k \log \frac{S}{k})$ bits to define the k cells she needs. For the right values of k this is significantly less than $k \log S$ bits which Alice needs if she performs the queries sequentially.

Lopsided Set Disjointness (LSD). In LSD Alice and Bob receive subsets S and T of a universe U. The goal for Alice and Bob is to compute whether $S \cap T \neq \emptyset$. LSD is parameterized with the size $|S| = N$ of Alice's set and the fraction between the size of the universe and N, which is denoted B, e.g. $|U| = NB$. Notice that the size of Bob's set is arbitrary and could be as large as NB. We use $[X]$ to denote the set $\{1, 2, \ldots, X\}$. There are other versions of LSD where the input to Alice has more structure. For our purpose we need Blocked-LSD. For this problem the universe is considered as the cartesian product of $[N]$ and $[B]$, e.g. $U = [N] \times [B]$ and Alice receives a set S such that $\forall j \in [N]$ there exists a unique $b_j \in [B]$ such that $(j, b_j) \in S$, e.g. S is of the form $\{(1, b_1), (2, b_2), \ldots, (N, b_N)\}$ where $b_i \in [B]$. The following lower bound applies for this problem [58].

Theorem 2.1. *Fix $\delta > 0$. In a bounded-error protocol for Blocked-LSD, either Alice sends $\Omega(N \log B)$ bits or Bob sends $\Omega(NB^{1-\delta})$ bits.*

Blocked-LSD Reduces to N/k Parallel Range Mode Queries. Given n, we describe a reduction from Blocked-LSD with a universe of size n ($n = NB$) to N/k parallel range mode queries on an input array A of size $\Theta(n)$. The size of A may not be exactly n but this will not affect our result. The parameters k and B are fixed later in the construction. From a high level perspective we construct an array of permutations of $[kB]$. A query consists of a suffix of one permutation, a number of complete permutations, and a prefix of another permutation. They are chosen such that the suffix determines a subset of Bob's set and the prefix a subset of Alice's set. These two subsets intersect if and only if the frequency of the mode is equal to two plus the number of complete permutations spanned by the query.

Bob stores a range mode data structure and Alice simulates the query algorithm. First we describe the array A that Bob constructs when he receives his input. Let $T \subseteq [N] \times [B]$ be this set. The array Bob constructs consists of two parts which are described separately. We let $\cdot$ denote concatenation of lists. We also use this operator on sets and in this case we treat the set as a list by placing the elements in lexicographic order. Bob partitions $[N]$ into N/k consecutive chunks of k elements, e.g. the i'th chunk is $\{(i-1)k+1, \ldots, ik\}$ for $i = 1, \ldots, N/k$. With the i'th chunk Bob associates the subset L_i of T with first coordinate in that chunk, e.g. $L_i = T \cap (\{(i-1)k+t \mid t = 1, \ldots, k\} \times [B])$. Each L_i is mapped to a permutation of $[kB]$.

We define the mapping $f : (x, y) \to (x - 1 \bmod k)B + y$ and let the permutation be $([kB] \setminus f(L_i)) \cdot f(L_i)$, e.g. we map the elements in L_i into $[kB]$ and prepend the elements of $[kB]$ not mapped to by any element in L_i such that we get a full

permutation of $[kB]$. The first part of A is the concatenation of the permutations defined for each chunk L_i ordered by i, e.g. $([kB] \setminus f(L_1)) \cdot f(L_1) \cdots ([kB] \setminus f(L_{N/k})) \cdot f(L_{N/k})$. The second part of A consists of B^k permutations of $[kB]$. There is one permutation for each way of picking a set of the form $\{(1, b_1), \ldots, (k, b_k)\}$ where $b_i \in [B]$. Let $R_1, \ldots, R_{B^k}$ denote the B^k sets on this form ordered lexicographically. The second part of the array becomes $f(R_1) \cdot ([kB] \setminus f(R_1)) \cdots f(R_{B^k}) \cdot ([kB] \setminus f(R_{B^k}))$.

We now show how Alice and Bob can determine whether $S \cap T \neq \emptyset$ from this array. Bob constructs a range mode data structure for A and sends $|L_i|$ for $i = 1, \ldots, N/k$ to Alice. Alice then simulates the query algorithm on the range mode data structure for N/k queries in parallel. The i'th query determines whether the k elements $Q_i = \{((i-1)k + 1, b_{(i-1)k+1}), \ldots, (ik, b_{ik})\}$ from S have an empty intersection with T (actually L_i) as follows.

Alice determines the end index of $f(Q_i)$ in the second part of A. We note that $f(Q_i)$ always exists in the second part of A by construction and Alice can determine the position without any communication with Bob. Alice also determines the start index of $f(L_i)$ in the first part of A from the sizes she initially received from Bob. The i'th query computes the frequency R_i of the mode between these two indices. Let p be the number of permutations of $[kB]$ stored between the end of $f(L_i)$ and the beginning of $f(Q_i)$ in A, then $F_i - p = 2$ if and only if $Q_i \cap T \neq \emptyset$, and $F_i - p = 1$ otherwise. Since each permutation of $[kB]$ contributes one to F_i, $F_i - p$ is equal to two if and only if at least one of the elements from Q_i is in L_i meaning that $S \cap T \neq \emptyset$. We conclude that Blocked-LSD reduces to N/k range mode queries in an array of size $NB + B^k kB$.

To obtain a lower bound for range mode data structures we consider the parameters k and B and follow the approach from [58]. Let S be the size of Bob's range mode data structure and let t be the query time. In our protocol for Blocked-LSD Alice sends $t \log \binom{S}{N/k} = \mathcal{O}(t\frac{N}{k} \log \frac{Sk}{N})$ bits and Bob sends $twN/k + N/k \log(kB)$ bits. By Theorem 2.1, either Alice sends $\Omega(N \log B)$ bits or Bob sends $\Omega(NB^{1-\delta})$. Fix $\delta = \frac{1}{2}$. Since $N/k \log(kB) = o(N\sqrt{B})$ we obtain that either $t\frac{N}{k} \log(\frac{Sk}{N}) = \Omega(N \log B)$ or $twN/k = \Omega(N\sqrt{B})$. We constrain B such that $B \geq w^2$ and $\log B \geq \frac{1}{2} \log(\frac{Sk}{N}) \Rightarrow B \geq \frac{Sk}{n}$ and obtain $t = \Omega(k)$. Since $|A| = NB + B^k kB$ and we require $|A| = \Theta(n)$, we set $k = \Theta(\log_B n)$. To maximize k we choose $B = \max\{w^2, \frac{Sk}{n}\}$. We obtain that $t = \Omega(k) = \Omega(\log N / \log \frac{Swk}{n}) = \Omega(\log n / \log \frac{Sw}{n})$ since $w > k$.

Summarizing, we get the following theorem.

Theorem 2.2. *Any data structure that uses S space needs $\Omega(\frac{\log n}{\log \frac{Sw}{n}})$ time for a range mode query in an array of size n.*

It follows from the construction that we get the same lower bound for data structures that support queries that are given i, j and k, and returns whether there exists an element with frequency exactly k in $A[i, j]$ or support queries that are given i, j and k and returns whether there is an element with frequency at least k in $A[i, j]$.

2.3 Range k-frequency

In this section, we consider the *range k-frequency* problem and its connection to classic geometric data structure problems. We show that the range k-frequency

problem is equivalent to 2D rectangle stabbing for any fixed constant $k > 1$, and that for $k = 1$ the problem reduces to four-sided 3D orthogonal range emptiness.

In the 2D rectangle stabbing problem the input is n axis-parallel rectangles. A query is given a point, (x, y), and must return whether this point is contained[1] in at least one of the n rectangles in the input. A query lower bound of $\Omega(\log n/\log(Sw/n))$ for data structures using S space is proved in [58], and a linear space static data structure with optimal $\mathcal{O}(\log n/\log\log n)$ query time can be found in [45].

In four-sided 3D orthogonal range emptiness, we are given a set P of n points in 3D, and must preprocess P into a data structure, such that given an open-ended four-sided rectangle $R = (-\infty, x] \times [y_1, y_2] \times [z, \infty)$, the data structure returns whether R contains a point $p \in P$. Currently, the best solution for this problem uses $\mathcal{O}(n \log n)$ space and supports queries in $\mathcal{O}(\log^2 \log n)$ time [3].

For simplicity, we assume that each coordinate is a unique integer between one and $2n$ (rank space).

Theorem 2.3. *Let k be a constant greater than one. The 2D rectangle stabbing problem reduces to the range k-frequency problem.*

Proof. We show the reduction for $k = 2$ and then generalize this construction to any constant value $k > 2$.

Let $R_1, \ldots, R_n$ be the input to the rectangle stabbing problem. We construct a range 2-frequency instance with n distinct labels each of which is duplicated exactly 6 times. Let R_ℓ be the rectangle $[x_{\ell_0}, x_{\ell_1}] \times [y_{\ell_0}, y_{\ell_1}]$. For each rectangle, R_ℓ, we add the pairs (x_{ℓ_0}, ℓ), (x_{ℓ_1}, ℓ) and (x_{ℓ_1}, ℓ) to a list X. Similarly, we add the pairs (y_{ℓ_0}, ℓ), (y_{ℓ_1}, ℓ), and (y_{ℓ_1}, ℓ) to a list Y. We sort X in descending order and Y in ascending order by their first coordinates. Since we assumed all coordinates are unique, the only ties are amongst pairs originating from the same rectangle, here we break the ties arbitrarily. The concatenation of X and Y is the range 2-frequency instance and we denote it A, i.e. the second component of each pair are the actual entries in A, and the first component of each pair is ignored.

We translate a 2D rectangle stabbing query, (x, y), into a query for the range 2-frequency instance as follows. Let p_x be the smallest index where the first coordinate of $X[p_x]$ is x, and let q_y be the largest index where the first coordinate of $Y[p_y]$ is y. If $A[p_x] = A[p_x + 1]$, two consecutive entries in A are defined by the right endpoint of the same rectangle, we set $i_x = p_x + 2$ (we move i_x to the right of the two entries), otherwise we set $i_x = p_x$. Similarly for the y coordinates, if $A[|X| + q_y] = A[|X| + q_y - 1]$ we set $j_y = q_y - 2$ (move j_y left of the two entries), otherwise we set $j_y = q_y$. Finally we translate (x, y) to the range 2-frequency query $[i_x, |X| + j_y]$ on A, see Figure 2.1. Notice that in the range 2-frequency queries that can be considered in the reduction, the frequency of a label is either one, two, three, four or six. The frequency of label ℓ in $A[i_x, |X|]$ is one if $x_{\ell_0} \leq x \leq x_{\ell_1}$, three if $x > x_{\ell_1}$ and zero otherwise. Similar, the frequency of ℓ in $A[|X|+1, |X|+j_y]$ is one if $y_{\ell_0} \leq y \leq y_{\ell_1}$, three if $y > y_{\ell_1}$ and zero otherwise. We conclude that the point (x, y) stabs rectangle R_ℓ if and only if the label ℓ has frequency two in $A[i_x, |X| + j_y]$.

Since $x, y \in \{1, \ldots, 2n\}$, we can store a table with the translations from x to i_x and y to j_y. Thus, we can translate 2D rectangle stabbing queries to range 2-frequency queries in constant time.

[1] points on the border of a rectangle are contained in the rectangle

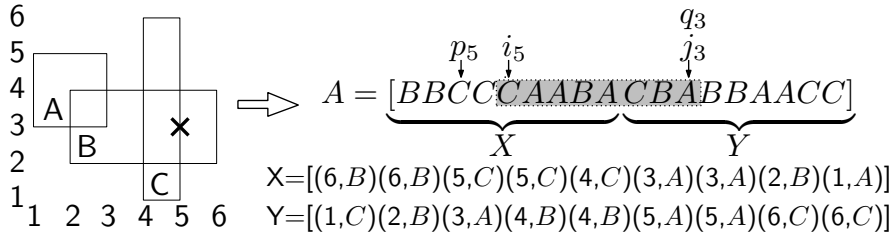


Figure 2.1: Reduction from 2D rectangle stabbing to range 2-frequency. The $\times$ marks a stabbing query, $(5, 3)$. This query is mapped to the range 2-frequency query $[i_5, |X|+j_3]$ in A, which is highlighted. Notice that $i_5 = p_5+2$ since $A[p_5] = A[p_5+1]$.

For $k > 2$ we place $k-2$ copies of each label between X and Y and translate the queries accordingly. □

The following theorem provides a matching upper bound.

Theorem 2.4. *The range k-frequency problem reduces to 2D rectangle stabbing.*

Proof. Let A be the input to the range k-frequency problem. We translate the ranges of A where there is a label with frequency k into $\mathcal{O}(n)$ rectangles as follows. Fix a label $x \in A$, and let $s_x \geq k$ denote the number of occurrences of x in A. If $s_x < k$ then x is irrelevant and we discard it. Otherwise, let $i_1 < i_2 < \ldots < i_s$ be the position of x in A, and let $i_0 = 0$ and $i_{s+1} = n + 1$. Consider the ranges of A where x has frequency k. These are the subarrays, $A[a, b]$, where there exists an integer ℓ such that $i_\ell < a \leq i_{\ell+1}$ and $i_{\ell+k} \leq b < i_{\ell+k+1}$ for $0 \leq \ell \leq s_x - k$. This defines $s_x - k + 1$ two dimensional rectangles, $[i_\ell + 1, i_{\ell+1}] \times [i_{\ell+k}, i_{\ell+k+1} - 1]$ for $\ell = 0, \ldots, s_x - k$, such that x has frequency k in $A[i, j]$ if and only if the point (i, j) stabs one of the $s_x - k + 1$ rectangles defined by x. By translating the ranges of A where a label has frequency k into the corresponding rectangles for all distinct labels in A, we get a 2D rectangle stabbing instance with $\mathcal{O}(n)$ rectangles. □

This means that we get a data structure for the range k-frequency problem that uses $\mathcal{O}(n)$ space and supports queries in $\mathcal{O}(\log n / \log \log n)$ time.

Theorem 2.5. *For $k = 1$, the range k-frequency problem reduces to four-sided orthogonal range emptiness queries in 3D.*

Proof. For each distinct label $x \in A$, we map the ranges of A where x has frequency one (it is unique in the range) to a 3D point. Let $i_1 < i_2 < \cdots < i_s$ be the positions of x in A, and let $i_0 = 0$ and $i_{s+1} = n + 1$. The label x has frequency one in $A[a, b]$ if there exist an integer ℓ such that $i_{\ell-1} < a \leq i_\ell \leq b < i_{\ell+1}$. We define s points, $P_x = \{(i_{\ell-1} + 1, i_\ell, i_{\ell+1} - 1) \mid 1 \leq \ell \leq s\}$. The label x has frequency one in the range $A[a, b]$ if and only if the four-sided orthogonal range query $[-\infty, a] \times [a, b] \times [b, \infty]$ contains a point from P_x (we say that x is inside range $[x_1, x_2]$ if $x_1 \leq x \leq x_2$). Therefore, we let $P = \bigcup_{x \in A} P_x$ and get a four-sided 3D orthogonal range emptiness instance with $\mathcal{O}(n)$ points. □

Thus from [3], we get a data structure for the range 1-frequency problem that uses $\mathcal{O}(n \log n)$ space and supports queries in $\mathcal{O}(\log^2 \log n)$ time and we conclude that for data structures using $\mathcal{O}(n \log^{\mathcal{O}(1)} n)$ space, the range k-frequency problem is exponentially harder for $k > 1$ than for $k = 1$.

2.4 3-Approximate range mode

In this section, we construct a data structure that given a range $[i, j]$ computes a 3-approximation of $F_{i,j}$.

We use the following observation from in [18]. If we can cover $A[i, j]$ with three disjoint subintervals $A[i, x], A[x + 1, y]$ and $A[y + 1, j]$ then we have $\frac{1}{3}F_{i,j} \leq \max\{F_{i,x}, F_{x+1,y}, F_{y+1,j}\} \leq F_{i,j}$.

First, we describe a data structure that uses $\mathcal{O}(n \log\log n)$ space, and then we show how to reduce the space to $\mathcal{O}(n)$. The data structure consists of a tree T of polynomial fanout where the i'th leaf stores $A[i]$, for $i = 1, \ldots, n$. For a node v let T_v denote the subtree rooted at v and let $|T_v|$ denote the number of leaves in T_v. The fanout of node v is $f_v = \lceil\sqrt{|T_v|}\rceil$. The height of T is $\Theta(\log\log n)$. Along with T, we store a lowest common ancestor (LCA) data structure, which given indices i and j, finds the LCA of the leaves corresponding to i and j in T in constant time [40].

For every node $v \in T$, let $R_v = A[a, b]$ denote the consecutive range of entries stored in the leaves of T_v. The children $c_1, \ldots, c_{f_v}$ of v partition R_v into f_v disjoint subranges $R_{c_1} = A[a_{c_1}, b_{c_1}]$, $\ldots$, $R_{c_{f_v}} = A[a_{c_{f_v}}, b_{c_{f_v}}]$ each of size $\mathcal{O}(\sqrt{|T_v|})$. For every pair of children c_r and c_s where $r < s - 1$, we store $F_{a_{c_{r+1}}, b_{c_{s-1}}}$. Furthermore, for every child range R_{c_i} we store $F_{a_{c_i},k}$ and $F_{k,b_{c_i}}$ for every prefix and suffix range of R_{c_i} respectively. To compute a 3-approximation of $F_{i,j}$, we find the LCA of i and j. This is the node v in T for which i and j lie in different child subtrees, say T_{c_x} and T_{c_y} with ranges $R_{c_x} = [a_{c_x}, b_{c_x}]$ and $R_{c_y} = [a_{c_y}, b_{c_y}]$. We then lookup the frequency $F_{a_{c_{x+1}}, b_{c_{y-1}}}$ stored for the pair of children c_x and c_y, as well as the suffix frequency $F_{i,b_{c_x}}$ stored for the range $A[i, b_{c_x}]$ and the prefix frequency $F_{a_{c_y},j}$ stored for $A[a_{c_y}, j]$, and return the max of these.

Each node $v \in T$ uses $\mathcal{O}(|T_v|)$ space for the frequencies stored for each of the $\mathcal{O}(|T_v|)$ pairs of children, and for all the prefix and suffix range frequencies. Since each node v uses $\mathcal{O}(|T_v|)$ space and the LCA data structure uses $\mathcal{O}(n)$ space, our data structure uses $\mathcal{O}(n \log\log n)$ space. A query makes one LCA query and computes the max of three numbers which takes constant time.

We just need one observation to bring the space down to $\mathcal{O}(n)$. Consider a node $v \in T$. The largest possible frequency that can be stored for any pair of children of v, or for any prefix or suffix range of a child of v is $|T_v|$, and each such frequency can be represented by $b = 1 + \lfloor\log|T_v|\rfloor$ bits. We divide the frequencies stored in v into chunks of size $\lfloor\frac{\log n}{b}\rfloor$ and pack each of them in one word. This reduces the total space usage of the nodes at depth i to $\mathcal{O}(n/2^i)$. We conclude that the data structure uses $\mathcal{O}(n)$ space and supports queries in constant time.

Theorem 2.6. *There exists a data structure for the* 3*-approximate range mode problem that uses $\mathcal{O}(n)$ space and supports queries in constant time.*

2.5 $(1 + \varepsilon)$-Approximate range mode

In this section, we describe a data structure using $\mathcal{O}(\frac{n}{\varepsilon})$ space that given a range $[i, j]$, computes a $(1 + \varepsilon)$-approximation of $F_{i,j}$ in $\mathcal{O}(\log\frac{1}{\varepsilon})$ time. Our data structure consists of two parts. The first part solves all queries $[i, j]$ where $F_{i,j} \leq \lceil\frac{1}{\varepsilon}\rceil$ (small frequencies), and the latter solves the remaining. The first data structure also decides whether $F_{i,j} \leq \lceil\frac{1}{\varepsilon}\rceil$. We use that $\frac{1}{\log(1+\varepsilon)} = \mathcal{O}(\frac{1}{\varepsilon})$ for any $0 < \varepsilon \leq 1$.

Small Frequencies. For $i = 1, \ldots, n$ we store a table, Q_i, of length $\lceil \frac{1}{\varepsilon} \rceil$, where the value in $Q_i[k]$ is the largest integer $j \geq i$ such that $F_{i,j} = k$. To answer a query $[i, j]$ we do a successor search for j in Q_i. If j does not have a successor in Q_i then $F_{i,j} > \lceil \frac{1}{\varepsilon} \rceil$, and we query the second data structure. Otherwise, let s be the index of the successor of j in Q_i, then $F_{i,j} = s$. The data structure uses $\mathcal{O}(\frac{n}{\varepsilon})$ space and supports queries in $\mathcal{O}(\log \frac{1}{\varepsilon})$ time.

Large Frequencies. For every index $1 \leq i \leq n$, define a list T_i of indeces with length $t = \lceil \log_{1+\varepsilon}(\varepsilon n) \rceil$, with the following invariant: For all j, if $T_i[k-1] < j \leq T_i[k]$ then $\lceil \frac{1}{\varepsilon}(1+\varepsilon)^k \rceil$ is a $(1+\varepsilon)$-approximation of $F_{i,j}$. The following assignment of values to the lists T_i satisfies this invariant:

Let $m(i, k)$ be the largest integer $j \geq i$ such that $F_{i,j} \leq \lceil \frac{1}{\varepsilon}(1+\varepsilon)^{k+1} \rceil - 1$. For T_1 we set $T_1[k] = m(1, k)$ for all $k = 1, \ldots, t$. For the remaining T_i we set

$$T_i[k] = \begin{cases} T_{i-1}[k] & \text{if } F_{i,T_{i-1}[k]} \geq \lceil \frac{1}{\varepsilon}(1+\varepsilon)^k \rceil + 1 \\ m(i,k) & \text{otherwise} \end{cases} .$$

The n lists are sorted by construction. For T_1, it is true since $m(i, k)$ is increasing in k. For T_i, it follows that $F_{i,T_i[k]} \leq \lceil \frac{1}{\varepsilon}(1+\varepsilon)^{k+1} \rceil - 1 < F_{i,T_i[k+1]}$, and thus $T_i[k] < T_i[k+1]$ for any k.

Let s be the index of the successor of j in T_i. We know that $F_{i,T_i[s]} \leq \lceil \frac{1}{\varepsilon}(1+\varepsilon)^{s+1} \rceil - 1$, $F_{i,T_i[s-1]} \geq \lceil \frac{1}{\varepsilon}(1+\varepsilon)^{s-1} \rceil + 1$ and $T_i[s-1] < j \leq T_i[s]$. It follows that

$$\lceil \tfrac{1}{\varepsilon}(1+\varepsilon)^{s-1} \rceil + 1 \leq F_{i,j} \leq \lceil \tfrac{1}{\varepsilon}(1+\varepsilon)^{s+1} \rceil - 1 , \tag{2.1}$$

and that $\lceil \frac{1}{\varepsilon}(1+\varepsilon)^s \rceil$ is a $(1+\varepsilon)$-approximation of $F_{i,j}$.

The second important property of the n lists, is that they only store $\mathcal{O}(\frac{n}{\varepsilon})$ different indices, which allows for a space-efficient representation. If $T_{i-1}[k] \neq T_i[k]$ then the following $\lceil \frac{1}{\varepsilon}(1+\varepsilon)^{k+1} \rceil - 1 - \lceil \frac{1}{\varepsilon}(1+\varepsilon)^k \rceil - 1 \geq \lfloor (1+\varepsilon)^k \rfloor - 3$ entries, $T_{i+a}[k]$ for $a = 1, \ldots, \lfloor (1+\varepsilon)^k \rfloor - 3$, are not changed, hence we store the same index at least $\max\{1, \lfloor (1+\varepsilon)^k \rfloor - 2\}$ times. Therefore, the number of changes to the n lists, starting with T_1, is bounded by $\sum_{k=1}^{t} \frac{n}{\max\{1, \lfloor (1+\varepsilon)^k \rfloor - 2\}} = \mathcal{O}(\frac{n}{\varepsilon})$. This was observed in [18], where similar lists are maintained in a partially persistent search tree [30].

We maintain these lists without persistence such that we can access any entry in any list T_i in constant time. Let $I = \{1, 1+t, \ldots, 1+\lfloor (n-1)/t \rfloor t\}$. For every $\ell \in I$ we store T_ℓ explicitly as an array S_ℓ. Secondly, for $\ell \in I$ and $k = 1, \ldots, \lceil \log_{1+\varepsilon} t \rceil$ we define a bit vector $B_{\ell,k}$ of length t and a change list $C_{\ell,k}$, where

$$B_{\ell,k}[a] = \begin{cases} 0 & \text{if } T_{\ell+a-1}[k] = T_{\ell+a}[k] \\ 1 & \text{otherwise} \end{cases} .$$

Given a bit vector L, define $\mathrm{sel}(L, b)$ as the index of the b'th one in L. We set

$$C_{\ell,k}[a] = T_{\ell + \mathrm{sel}(B_{\ell,k}, a)}[k] .$$

Finally, for every $\ell \in I$ and for $k = 1 + \lceil \log_{1+\varepsilon} t \rceil, \ldots, t$ we store $D_\ell[k]$ which is the smallest integer $z > \ell$ such that $T_z[k] \neq T_\ell[k]$. We also store $E_\ell[k] = T_{D_\ell[k]}[k]$. We store each bit vector in a rank and select data structure [43] that uses $\mathcal{O}(\frac{n}{w})$ space

for a bit vector of length n, and supports $rank(i)$ in constant time. A rank(i) query returns the number of ones in the first i bits of the input.

Each change list, $C_{\ell,k}$ and every D_ℓ and E_ℓ list is stored as an array. The bit vectors indicate at which indices the contents of the first $\lceil \log_{1+\varepsilon} t \rceil$ entries of $T_\ell, \ldots, T_{\ell+t-1}$ change, and the change lists store what the entries change to. The D_ℓ and E_ℓ arrays do the same thing for the last $t - \lceil \log_{1+\varepsilon} t \rceil$ entries, exploiting that these entries change at most once in an interval of length t.

Observe that the arrays, $C_{\ell,k}, D_\ell[k]$ and $E_\ell[k]$, and the bit vectors, $B_{\ell,k}$ allow us to retrieve the contents of any entry, $T_i[k]$ for any i, k, in constant time as follows. Let $\ell = \lfloor i/t \rfloor t$. If $k > \lceil \log_{1+\varepsilon} t \rceil$ we check if $D_\ell[k] \leq i$, and if so we return $E_\ell[k]$, otherwise we return $S_\ell[k]$. If $k \leq \lceil \log_{1+\varepsilon} t \rceil$, we determine $r = \text{rank}(i - \ell)$ in $B_{\ell,k}$ using the rank and select data structure. We then return $C_{\ell,k}[r]$ unless $r = 0$ in which case we return $S_\ell[k]$.

We argue that this correctly returns $T_i[k]$. In the case where $k > \lceil \log_{1+\varepsilon} t \rceil$, comparing $D_\ell[k]$ to i indicates whether $T_i[k]$ is different from $T_\ell[k]$. Since $T_z[k]$ for $z = \ell, \ldots, i$ can only change once, $T_i[k] = E_\ell[k]$ in this case. Otherwise, $S_\ell[k] = T_\ell[k] = T_i[k]$. If $k \leq \lceil \log_{1+\varepsilon} t \rceil$, the rank r of $i - \ell$ in $B_{\ell,k}$, is the number of changes that has occurred in the k'th entry from list T_ℓ to T_i. Since $C_{\ell,k}[r]$ stores the value of the k'th entry after the r'th change, $C_{\ell,k}[r] = T_i[k]$, unless $r = 0$ in which case $T_i[k] = S_\ell[k]$.

The space used by the data structure is $\mathcal{O}(\frac{n}{\varepsilon})$. We store $3\lceil \frac{n}{t} \rceil$ arrays, S_ℓ, D_ℓ and E_ℓ for $\ell \in I$, each using t space, in total $\mathcal{O}(n)$. The total size of the change lists, $C_{\ell,k}$, is bounded by the number of changes across the T_i lists, which is $\mathcal{O}(\frac{n}{\varepsilon})$ by the arguments above. Finally, the rank and select data structures, $B_{\ell,k}$, each occupy $\mathcal{O}(\frac{t}{w}) = \mathcal{O}(\frac{t}{\log n})$ words, and we store a total of $\lceil \frac{n}{t} \rceil \lceil \log_{1+\varepsilon} t \rceil$ such structures, thus the total space used by these is bounded by

$$\mathcal{O}(\frac{t}{\log n}\frac{n}{t}\log_{1+\varepsilon} t) = \mathcal{O}(n\frac{\log_{1+\varepsilon} t}{\log n}) = \mathcal{O}(\frac{n}{\varepsilon}\frac{\log t}{\log n}) = \mathcal{O}(\frac{n}{\varepsilon}\frac{\log(n\log(\varepsilon n))}{\log n}) = \mathcal{O}(\frac{n}{\varepsilon}).$$

We use that if $\lceil \frac{1}{\varepsilon} \rceil \geq n$ then we only store the small frequency data structure. We conclude that our data structures uses $\mathcal{O}(\frac{n}{\varepsilon})$ space.

To answer a query $[i, j]$, we first compute a 3-approximation of $F_{i,j}$ in constant time using the data structure from Section 2.4. Thus, we find $f_{i,j}$ satisfying $f_{i,j} \leq F_{i,j} \leq 3f_{i,j}$. Choose k such that $\lceil \frac{1}{\varepsilon}(1+\varepsilon)^k \rceil + 1 \leq f_{i,j} \leq \lceil \frac{1}{\varepsilon}(1+\varepsilon)^{k+1} \rceil - 1$ then the successor of j in T_i must be in one of the entries, $T_i[k], \ldots, T_i[k + \mathcal{O}(\log_{1+\varepsilon} 3)]$. As stated earlier, the values of T_i are sorted in increasing order, and we find the successor of j using a binary search on an interval of length $\mathcal{O}(\log_{1+\varepsilon} 3)$. Since each access to T_i takes constant time, we use $\mathcal{O}(\log \log_{1+\varepsilon} 3) = \mathcal{O}(\log \frac{1}{\varepsilon})$ time.

Theorem 2.7. *There exists a data structure for $(1+\varepsilon)$-approximate range mode that uses $\mathcal{O}(\frac{n}{\varepsilon})$ space and supports queries in $\mathcal{O}(\log \frac{1}{\varepsilon})$ time.*

The careful reader may have noticed that our data structure returns a frequency, and not a label that occurs approximately $F_{i,j}$ times. We can augment our data structure to return a label instead as follows.

We set $\varepsilon' = \sqrt{(1+\varepsilon)} - 1$, and construct our data structure from above. The *small frequency* data structure is augmented such that it stores the label $M_{i,Q_i[k]}$ along with $Q_i[k]$, and returns this in a query. The *large frequency* data structure is augmented such that for every update of $T_i[k]$ we store the label that caused the

update. Formally, let $a > 0$ be the first index such that $T_{i+a}[k] \neq T_i[k]$. Next to $T_i[k]$ we store the label $L_i[k] = A[i + a - 1]$. In a query, $[i, j]$, let s be the index of the successor of j in T_i computed as above. If $s > 1$ we return the label $L_i[s - 1]$, and if $s = 1$ we return $M_{i,Q_i[\lceil 1/\varepsilon' \rceil]}$, which is stored in the small frequency data structure.

In the case where $s = 1$ we know that $\lceil \frac{1}{\varepsilon'} \rceil \leq F_{i,j} \leq \lceil \frac{1}{\varepsilon'}(1 + \varepsilon')^2 \rceil - 1 = \lceil \frac{1}{\varepsilon'}(1 + \varepsilon) \rceil - 1$ and we know that the frequency of $M_{i,Q_i[\lceil 1/\varepsilon' \rceil]}$ in $A[i, j]$ is at least $\lceil \frac{1}{\varepsilon'} \rceil$. We conclude that the frequency of $M_{i,Q_i[\lceil 1/\varepsilon' \rceil]}$ in $A[i, j]$ is a $(1 + \varepsilon)$-approximation of $F_{i,j}$.

If $s > 1$ we know that $\lceil \frac{1}{\varepsilon'}(1 + \varepsilon')^{s-1} \rceil + 1 \leq F_{i,j} \leq \lceil \frac{1}{\varepsilon'}(1 + \varepsilon')^{s+1} \rceil - 1$ by equation (2.1), and that the frequency, f_L, of the label $L_i[s - 1]$ in $A[i, j]$ is at least $\lceil \frac{1}{\varepsilon'}(1 + \varepsilon')^{s-1} \rceil + 1$. This means that $F_{i,j} \leq \frac{1}{\varepsilon'}(1 + \varepsilon')^{s+1} \leq (1 + \varepsilon')^2 f_L = (1 + \varepsilon) f_L$, and we conclude that f_L is a $(1 + \varepsilon)$-approximation of $F_{i,j}$.

The space needed for this data structure is $\mathcal{O}(\frac{n}{\varepsilon'}) = \mathcal{O}(\frac{n(\sqrt{1+\varepsilon}+1)}{\varepsilon}) = \mathcal{O}(\frac{n}{\varepsilon})$, and a query takes $\mathcal{O}(\log \frac{1}{\varepsilon'}) = \mathcal{O}(\log \frac{1}{\varepsilon} + \log(\sqrt{1 + \varepsilon} + 1)) = \mathcal{O}(\log \frac{1}{\varepsilon})$ time.

2.6 Concluding remarks

We have shown that using only linear space we can get any constant factor approximation of the mode in constant time. Secondly, we considered the range k-frequency problem and showed how this problem is strongly related to geometric data structure problems. We found matching upper and lower bounds for any constant $k > 1$, and showed that for $k = 1$ it is exponentially easier to solve for near linear space data structures. Unfortunately, we were not able to exploit this in our efforts to prove anything new regarding the range mode problem. The range mode problem seems to be much harder than the range k-frequency problem, but we were not able to put any structure on the range mode problem that could be used to prove a lower bound as we did for the range k-frequency problem.

3

Implicit Dictonaries

In this chapter we consider the problem of creating *implicit dictionaries* [54] with the *working set* and *finger search* properties. The chapter is a merger of [19] which is joint work with Gerth Stølting Brodal and Casper Kejlberg-Rasmussen, and [20] which is joint work with Gerth Stølting Brodal and Jesper Sindahl Nielsen. Several section have been extended and the terminology has been unified.

3.1 Introduction

A *dictionary* is a data structure storing a set of elements with distinct comparable keys such that an element can be located efficiently given its key. It may also support predecessor and successor queries where given a query k it must return the element with the greatest key less than k or the element with smallest key greater than k. A dynamic dictionary also supports insertion the and deletion of elements.

A dictionary has the *finger search property* if the time for searching is dependent on the rank distance t between a specific element f, called the finger, and the query key k. In the static case $\mathcal{O}(\log t)$ search can be achieved by exponential search on a sorted array of elements starting at the finger. Dynamic finger search data structures have been widely studied, some of the famous dynamic structures that support finger searches are splay trees, randomized skip lists and level linked (2-4)-trees. These all support finger search in $\mathcal{O}(\log t)$ time, respectively in the amortised, expected and worst case sense. For an overview of data structures that support finger search see [21].

We consider two variants of finger search structures. The first variant is the *finger search dictionary* where the search operation also changes the finger to the returned element. The second variant is the *change finger dictionary* where the change-finger operation is separate from the search operation.

A dictionary has *the working set property* if the time for searching for an element e depend on the number of distinct elements ℓ searched for since e was last searched for. The splay tree [65], a skip list variant [16], and the working set structure [42], all achieve a query time of $\mathcal{O}(\log \ell)$ in the amortised, expected or worst-case sense.

The *unified access bound*, which is a generalization of the working set bound and the finger search property, is achieved in [12]. The unified access bound states that, if $\ell(g)$ is the number of distinct elements accessed since g was last accessed, and $d(g, e)$ denotes the rank distance between g and e, then the search time for e must be $\mathcal{O}(\min_g \log(\ell(g) + d(g, e) + 2))$.

We consider the problems in the implicit model. Here a data structure of n elements is an array with exactly n entries, each entry containing exactly one element

Reference	Insert/Delete	Search	Predecessor	Additional space (words)
[33]	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	None
[42]	$\mathcal{O}(\log n)$	$\mathcal{O}(\log \ell)$	$\mathcal{O}(\log \ell)$	$\mathcal{O}(n)$
[17, Sec. 2]	$\mathcal{O}(\log n)$	$\mathcal{O}(\log \ell)$ exp.	$\mathcal{O}(\log n)$	$\mathcal{O}(\log \log n)$
[17, Sec. 3]	$\mathcal{O}(\log n)$	$\mathcal{O}(\log \ell)$ exp.	$\mathcal{O}(\log \ell)$ exp.	$\mathcal{O}(\sqrt{n})$
This chapter	$\mathcal{O}(\log n)$	$\mathcal{O}(\log \ell)$	$\mathcal{O}(\log n)$	None
[22]	$\mathcal{O}(\log n)$	$\mathcal{O}(\log \ell)$	$\mathcal{O}(\log \ell)$	None

Table 3.1: The operation time, and space overhead of important structures for the working set dictionary problem.

which is indivisible. Computation is done on a machine with a constant number of registers with a word size of $\Theta(\log n)$ bits. All operations on registers are unit cost, similar to the RAM model. There is some debate about how much memory can be used between operations. We therefore partition this into two different models. In the weak implicit model $\mathcal{O}(1)$ extra words of information may be stored between operations. Examples are [33, 34, 54]. In the strong implicit model *no* additional space is allowed, only the number of elements n is assumed to be implicitly maintained. Examples assuming the strong implicit model are [15, 35]. In either model the only allowed operations on elements are comparisons and swaps.

In both models almost all structure has to be encoded in the order of the elements. As there is no agreement on the exact definition of the implicit model, it is interesting to study the limits of the strict model. We show that for a static change-finger dictionary in the strict model, if we want a search time of $\mathcal{O}(\log t)$, then change-finger must take time $\Omega(n^\epsilon)$, while in the weak model a sorted array achieves $\mathcal{O}(\log n)$ change-finger time.

Extensive research has been done in the implicit/in-place models, from as early as binary heaps [68], to an in-place 3-d convex hull algorithm [27]. Implicit dictionaries have been the topic of several papers. Among the first [53] gave a dictionary supporting insert, delete and search in $\mathcal{O}(\log^2 n)$ time. In [34] an implicit B-tree is presented, supporting insert,delete, and Predecessor in $\mathcal{O}(\log^2 n / \log \log n)$ time , and finally in [33] a worst case optimal and cache oblivious dictionary is presented, supporting the operations in $\mathcal{O}(\log n)$ time and $\mathcal{O}(\log_B n)$ I/Os. For a more extensive overview see [52].

In [17] two dictionaries with low space overhead are presented, achieving the working set property in the expected sense, see Table 3.1.

As a continuation of the publication of [19], on which this chapter is partly based Gerth Stølting Brodal and Casper Kejlberg-Rasmussen published [22], where they show how to construct strict implicit dictonary with the working set property where also the Predecessor and Successor operations run in time $\mathcal{O}(\log \ell)$, where ℓ is with respect to the found element.

Preliminaries. A common implicit data structure technique is the pair encoding of bits. When we have two distinct consecutive elements x and y, then they encode a 1 if $x \leq y$ and 0 otherwise.

We will use set notation on a data structure when appropriate, e.g. $|X|$ will

denote the number of elements in the structure X and $e \in X$ will denote that the element e is in the structure X. Given two data structures or sets X and Y, we say that $X \prec Y \Leftrightarrow \forall(x,y) \in X \times Y : x < y$. We use $d(e_1, e_2)$ to denote the rank distance between two elements, that is the difference of the index of e_1 and e_2 in the sorted key order of all elements in the structure.

At any time f will denote the current finger element and t the rank distance between this and the current search key. The running time of the search operation is hereafter denoted by $q(t,n)$. Throughout the chapter we require that $q(t,n)$ is nondecreasing in both t and n, $q(t,n) \geq \log t$ and that $q(0,n) < \log \frac{n}{2}$. We define $Z_q(n) = \min\{t \in \mathbb{N} \mid q(t,n) \geq \log \frac{n}{2}\}$, i.e. $Z_q(n)$ is the smallest rank distance t, such that $q(t,n) > \log \frac{n}{2}$. Note that $Z_q(n) \leq \frac{n}{2}$ (since by assumption $q(t,n) \geq \log t$), and if q is a function of only t, then Z_q is essentially equivalent to $q^{-1}(\log \frac{n}{2})$. As an example $q(t,n) = \frac{1}{\varepsilon}\log t$, gives $Z_q(n) = \lceil (\frac{n}{2})^{\varepsilon} \rceil$, for $0 < \varepsilon \leq 1$. We require that for a given q, $Z_q(n)$ can be evaluated in constant time, and that $Z_q(n+1) - Z_q(n)$ is bounded by a fixed constant for all n.

Our results. To facilitate our working set and finger search results, we need a *movable* dictionary, i.e. a dictionary stored in a consecutive sub-array that can be moved to the left or the right, one position at a time. We construct a movable dictionary from a constant number of the implicit and cache-oblivious dictionaries from [33], achieving a dictionary inheriting the same properties, but which is also movable. The movable dictionary is in itself an interesting result because it is a general transformation, that can be applied to any data structure that can be laid out in an array and grows/shrinks in one end and supports insertions and deletions. Hence we can plug in say a binary heap, and get a movable binary heap. The movable dictonary is presented in Section 3.2.

We present an implicit dictionary with the working set property that supports insertions, deletions, and predecessor queries in $\mathcal{O}(\log n)$ time and search queries in $\mathcal{O}(\log \ell)$ time. Our result improves the construction of [17, Section 2] by requiring no additional space. Furthermore our structure is cache-oblivious and supports insert, delete and predecessor operations in $\mathcal{O}(\log_B n)$ cache-misses and search in $\mathcal{O}(\log_B \ell)$ cache misses. In the literature the working set property is often stated in terms of the number of operations. We note that if we perform a search for an element whenever it is inserted, we will also satisfy these kinds of bounds. The construct of the working set dictonary is given in Section 3.3 and a detailed description of how to manage the memory is given in Section 3.4. In Section 3.5 we prove correctness and running time bounds.

In Section 3.6 we present a static *change-finger implicit dictionary* supporting predecessor in time $\mathcal{O}(q(t,n))$, and change-finger in time $\mathcal{O}(Z_q(n) + \log n)$, for any function $q(t,n)$. Note that by choosing $q(t,n) = \frac{1}{\varepsilon}\log t$, we get a search time of $\mathcal{O}(\log t)$ and a change finger time of $\mathcal{O}(n^{\varepsilon})$ for any $0 < \varepsilon \leq 1$.

In Section 3.7 we outline a construction for creating a dynamic *change-finger implicit dictionary*, supporting insert and delete in time $\mathcal{O}(\log n)$, predecessor and successor in time $\mathcal{O}(q(t,n))$ and change-finger in time $\mathcal{O}(Z_q(n)\log n)$. Note that by setting $q(t,n) = \frac{2}{\varepsilon}\log t$, we get a search time of $\mathcal{O}(\log t)$ and a change-finger time of $\mathcal{O}(n^{\varepsilon/2}\log n) = \mathcal{O}(n^{\varepsilon})$ for any $0 < \varepsilon \leq 1$, which is asymptotically optimal in the strict model. It remains an open problem if one can get better bounds in the dynamic case by using $\mathcal{O}(1)$ additional words.

Finally in Section 3.8, we present a simple technique for proving lower bounds in the strict implicit model. First we prove (Lemma 3.1) that for any algorithm A on a strict implicit data structure of size n that runs in time at most τ, whose arguments are keys or elements from the structure, there exists a set $\mathcal{X}_{\mathtt{A},n}$ of at most $\mathcal{O}(2^\tau)$ array entries, such that A touches only array entries from $\mathcal{X}_{\mathtt{A},n}$, no matter the arguments to A or the content of the data structure. We use this to show that for any *change-finger implicit dictionary* with a search time of $q(t,n)$, change-finger will take time $\Omega(Z_q(n) + \log n)$ for some t (Theorem 3.2).

We prove that for any *change-finger implicit dictionary* search will take time at least $\log t$ (Theorem 3.3). A similar argument applies for predecessor and successor. This means that our previous requirement $q(t,n) \geq \log t$ is necessary. We show that for any *finger-search implicit dictionary* search must take at least $\log n$ time as a function of both t and n, i.e. it is impossible to create any meaningful finger-search dictionary in the strict implicit model (Theorem 3.4).

By Theorem 3.2 and 3.3 the static data structure presented in Section 3.6 is optimal w.r.t. search and change-finger time trade off, for any function $q(t,n)$ as defined above. In the special case where the restriction $q(0,n) < \log \frac{n}{2}$ does not hold, [33] provides the optimal trade off. For completeness we also show that search time for the working-set dictonary in the strict implicit model, presented in Section 3.3, is asymptoticaly optimal with respect to ℓ and n, since it must be at least $\log \ell$ by Theorem 3.5.

3.2 A movable dictionary

In this section we describe an implicit movable dictionary which can be laid out in an array in the range $[i;j]$, where $n = j - i + 1$ is the number of elements in the dictionary. When deleting an element from the dictionary we are allowed to shrink the dictionary from the left or the right end, such that the structure now lies in the range $[i+1;j]$ or $[i;j-1]$, respectively. Likewise we can insert and expand the dictionary at the left or right end such that the structure now lies in the range $[i-1;j]$ or $[i;j+1]$, respectively. The structure also supports search and predecessor operations. All operations run in $\mathcal{O}(\log n)$ time. The movable dictionary is implicit except for $\mathcal{O}(\log n)$ extra bits that need to be stored/encoded externally (in the D_i structures in Section 3.3).

The dictionary supports the following operations:

- Insert-left(e) and insert-right(e): inserts an element e into the dictionary which grows in the left and right side, respectively.
- Delete-left(x) and delete-right(x): deletes the element with key x from the dictionary which shrinks in the left and right side, respectively.
- Search(x): returns the element e with key x in the dictionary if such an element exits, otherwise **none** is returned.
- Predecessor(x): is given a key x and returns the element e in the dictionary with the largest key less than x.

An amortised solution can be obtained using two of the dictionaries by Franceschini and Grossi [33] (in the following denoted FG dictionaries). Let r be an index

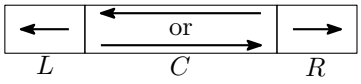


Figure 3.1: We have three FG dictionaries L, C and R, where L always grows/shrinks in the left direction, and R grows/shrinks in the right, and C will change direction during the execution of the jobs to shrink or grow L or R.

in the range $i \leq r \leq j$. One FG dictionary denoted R is located in the range $[r; j]$ and grows to the right as normal, and one FG dictionary denoted L is located in the range $[i; r-1]$ and grows to the left, i.e. for L we have inverted all the indexes of the original FG dictionary. The insert-left and insert-right operations insert elements into L and R, respectively. The delete-left operation searches for the element e to be deleted in L and R. If e is in L it is deleted from L and we are done. Otherwise e is deleted from R and an arbitrary element is deleted from L and inserted into R – provided L is non-empty. If L is empty we first rebuild the data structure such that L and R differ in size by at most one, by repeatedly reinserting into new L and R structures starting from the new index $r = \lceil \frac{i+j}{2} \rceil$. The delete-right operation is handled symmetrically. To search for an element with a given key, we search in L and then in R; to find the predecessor element of a given key we find the predecessor in L and R and return the largest of the two. Since [33] supports all operations in $\mathcal{O}(\log n)$ time, all operations run in $\mathcal{O}(\log n)$ amortised time, which e.g. can be seen using the potential function $\Phi = | \, |L| - |R| \, |$.

In the following we describe how to deamortise the above construction using incremental rebalancing of L and R. An additional FG dictionary C is placed between L and R (see Figure 3.1).

In the following we w.l.o.g. assume that $n \geq 24$, such that all intervals stated below are guaranteed to include an integer. If L or R get outside the range $[\frac{3}{24}n, \frac{7}{24}n]$, say L is getting too big/small, we initialize an incremental *job* to make L smaller/ bigger by transferring elements to/from C. Each time an insert and delete operation is executed we perform a constant number of steps of the current job. While resizing L there might be a pending job waiting for resizing R, and vice versa. During the execution of a job we have a temporary FG dictionary, which can be one of either L', C' or R', depending on how far we are in the execution of the job (see Figure 3.2).

3.2.1 Methods and jobs

The insert-left and delete-left operations, and the grow-left and shrink-left jobs described here have analogous right-versions.

Search(x) We always have the structures L, C and R, and possibly one of the structures L', C' or R'. We search each of the at most four structures. If we find an element e with key x we return e, otherwise we return **none**.

Predecessor(x) As in search we search for the predecessor in each of the structures L, C, R and possibly one of L', C' or R', and return the largest of the four candidates found.

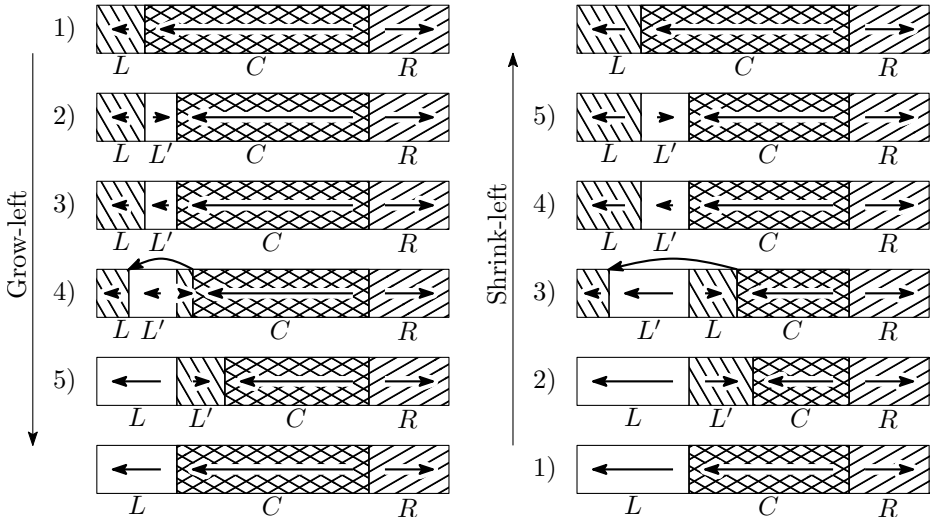


Figure 3.2: The steps of the two operations grow-left and shrink-left, notice that they are almost each other's inverse. (Left) The five steps of the grow-left operation, notice that in step 4) the arrow at the top means that we have split L up into two by use of address-mapping. (Right) The five steps of the shrink-left operation, in step 3) we have again used address-mapping to split L in two.

Insert-left(e) We insert e into L. If $|L| > \frac{7}{24}n$ we initialize a shrink-left job unless a left job is already running/pending.

Delete-left(x) We delete the element with key x from L. We can do this even though the element we want to delete resides in L', C, C', R or R' by swapping the element we want to delete with one from L. We can swap elements by performing two deletions and two insertions. If $|L| < \frac{3}{24}n$ we initialize a grow-left job unless a left job is already running/pending.

Grow-left The job consists of the following steps to be performed incrementally (see Figure 3.2 (left)). Notice that during the incremental work, deletions and insertions are performed on L and R by the update operations. We let n_{init} denote the size of the dictionary when the job is initialized, and assume that n_{init} is remembered when the job is initialized.

1) If C is not growing to the left then turn C around so it grows toward L. We turn C around by creating a new C' in the growing end of C which grows towards C, into which we insert all the elements of C, one element at a time.

2) Construct L' of size $\lceil \frac{2}{24}n_{\text{init}} \rceil$ at the beginning of L, growing to the right, by deleting elements from C and inserting them into L'.

3) Turn L' around so it faces L, like we turned C in step 1).

4) Continue deleting an element from C and inserting it into L', so L' expands into L. The element overridden in L is moved into the empty place in C where we took the element to place in L'. We do this by splitting L into two pieces by address-mapping, see steps 3) and 4) in Figure 3.2 (left). When we have moved L completely to the right of L', we swap the names of L and L'.

5) Merge L' back into C, by deleting an element from L' and inserting it into C until L' is empty.

Shrink-left The job consists of the following steps (see Figure 3.2 (right)). Notice the similarity to grow-left.

1) If C is not growing to the left then turn C around so it grows toward L.

2) Create L' by deleting $\lceil \frac{5}{24} n_{\text{init}} \rceil$ elements from C, one element at a time and inserting them into L', which we create to the left of C.

3) Swap the names of L and L'. Delete an element from L' and insert it into C so it expands into L, then move the element overridden in L to the empty space to the left of L', do this one element at a time until L is moved completely to the left of L'.

4) Turn L' around so it faces C.

5) Merge L' back into C.

3.2.2 Correctness

The correctness of the search and predecessor operations follows directly from the fact that the dictionary consists of at most four FG dictionaries. Similarly the insert-left and insert-right operations insert a single new element into an FG dictionary and otherwise only moves elements between the FG dictionaries. The only operations remaining to be considered are the delete-left and delete-right operations. In the following we only consider the delete-left operation (delete-right is symmetric). The only technical detail we need to argue about is that there always is a non-empty FG dictionary L oriented to the left that has its leftmost element stored in the leftmost entry in the subarray.

In the following when considering a job, we let n_{init}, n_0, n_{finish} denote the size of the movable dictionary: when the job was initialized, when the execution of the job started, and just after it is finished, respectively.

By performing the incremental work sufficiently fast, we will be able to perform the job during at most βn_0 movable dictionary updates, for any constant $\beta > 0$. An upper bound on the number of primitive steps (that is movement of one element from one FG dictionary to another one, and possibly move in memory) per update is: During the execution of the job at most βn_0 insertions can take place, i.e. the dictionary always has size at most $(1 + \beta)n_0$. Therefor each of the five steps of a job require at most $(1 + \beta)n_0$ primitive steps. In total there are at most $5(1 + \beta)n_0$ primitive steps. By performing at least $5(1 + \beta)/\beta$ primitive steps per update, the job finishes within βn_0 updates.

To relate n_{init} and n_0 we make the observation that any job under execution will finish during the next βn updates, where n is the current number of elements in the dictionary. To see this, observe that a job that has run for d updates needs to be executed for at most $\beta n_0 - d \leq \beta(n_0 - d) \leq \beta n$ further updates, provided $\beta \leq 1$. From this it follows that when a job is initialized, it at most takes βn_{init} updates before the current job finishes and the new job starts being executed, i.e. $(1 - \beta)n_{\text{init}} \leq n_0 \leq (1 + \beta)n_{\text{init}}$.

Let t_{finish} denote the number of updates between the initialization of a job until it is finished. We have $t_{\text{finish}} \leq \beta n_{\text{init}} + \beta n_0 \leq \beta n_{\text{init}} + \beta(1+\beta)n_{\text{init}} = (\beta^2 + 2\beta)n_{\text{init}}$. We get $n_{\text{finish}} \leq n_{\text{init}} + t_{\text{finish}} \leq (1 + \beta^2 + 2\beta)n_{\text{init}}$ and $n_{\text{finish}} \geq n_{\text{init}} - t_{\text{finish}} \geq (1 - \beta^2 - 2\beta)n_{\text{init}}$.

During the lifetime of a job, i.e. between its initialization and its the time it finish, there are always at least $\frac{3}{24}n_{\text{init}} - t_{\text{finish}} \geq (\frac{3}{24} - \beta^2 - 2\beta)n_{\text{init}}$ elements that still can be deleted from the leftmost FG dictionaries which shrink the subarray from the left. By selecting β sufficiently small such that $\beta^2 + 2\beta < \frac{3}{24}$, this number is always non-zero.

What remains to be argued is that *i*) $\frac{3}{24}n_{\text{finish}} \leq |L| \leq \frac{7}{24}n_{\text{finish}}$ when a left job is finished, *ii*) $|C| \geq \lceil \frac{2}{24}n_{\text{init}} \rceil$ when a grow job starts its execution, and *iii*) $|L| \geq \lceil \frac{5}{24}n_{\text{init}} \rceil$ immediately before step 3) in a shrink job. We need *i*) to ensure that $\frac{3}{24}n \leq |L| \leq \frac{7}{24}n$ holds just before a job is initialized, and *ii*) and *iii*) to ensure that grow-left and shrink-left are well defined, respectively.

The above can be shown by the following observations:

i) After a shrink or grow job $|L| \leq \frac{5}{24}n_{\text{init}} + 1 + t_{\text{finish}} \leq \frac{6}{24}n_{\text{init}} + t_{\text{finish}}$ which is less than $\frac{7}{24}n_{\text{finish}}$ for $\beta^2 + 2\beta \leq \frac{1}{31}$. Similarly after a shrink or grow job $|L| \geq \frac{5}{24}n_{\text{init}} - t_{\text{finish}}$ which is greater than $\frac{3}{24}n_{\text{finish}}$ for $\beta^2 + 2\beta \leq \frac{2}{27}$.

ii) Before grow-left $|C| \geq n_0 - |L| - |R| \geq (n_{\text{init}} - \beta n_{\text{init}}) - (\frac{7}{24}n_{\text{init}} + \beta n_{\text{init}}) - \frac{7}{24}n_{\text{init}}(1+\beta)$ which is greater than $\frac{3}{24}n_{\text{init}} \geq \lceil \frac{2}{24}n_{\text{init}} \rceil$ for $\beta \leq \frac{7}{55}$.

iii) In shrink-left $|L| \geq \frac{7}{24}n_{\text{init}} - t_{\text{finish}}$ which is greater than $\frac{6}{24}n_{\text{init}} \geq \lceil \frac{5}{24}n_{\text{init}} \rceil$ for $\beta^2 + 2\beta \leq \frac{1}{24}$. We note that setting $\beta = \frac{1}{63}$ will satisfy all the stated constraints.

The $\mathcal{O}(\log n)$ time bounds for the operations follow from the $\mathcal{O}(\log n)$ time bounds of the FG dictionaries. In the cache-oblivious model we notice that because the FG dictionary is cache-oblivious and we only use a constant number of FG dictionaries, where we split at most one of them into two parts by address-mapping then we only multiply the bound on the cache-misses from the FG dictionary by a constant factor. Hence all operations cause $\mathcal{O}(\log_B n)$ cache-misses.

We notice that we can make the movable dictionary implicit such that we do not need to store $\mathcal{O}(\log n)$ bits between operations. We do this by introducing a block D of $\mathcal{O}(\log n)$ elements to the left of L which *pair-encodes* the $\mathcal{O}(\log n)$ bits. With pair-encoding we mean that each consecutive pair of elements encodes a bit. As we need to read this block to get the $\mathcal{O}(\log n)$ bits, we can maintain (and possibly move) D when we perform insert-left, insert-right, delete-left and delete-right operations. From a cache-oblivious viewpoint this does also not change the asymptotic bound on the number of cache-misses.

3.3 Construction of the working set dictionary

In the following we describe our working set dictionary archiving insertions, deletions and predecessor searches in $\mathcal{O}(\log n)$ time and searches in $\mathcal{O}(\log \ell)$. We first describe the overall structure leaving the details of the memory layout to be handled in Section 3.4. The structure is composed of $\mathcal{O}(\log \log n)$ blocks, where the i'th block B_i stores $\mathcal{O}(2^{2^i})$ elements. The main design goal is to have elements that have been

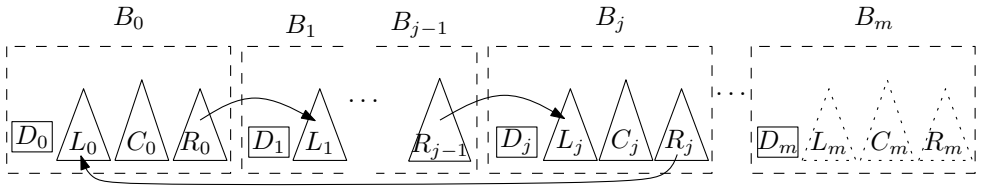


Figure 3.3: Layout of the data structure. The arrows indicate the movement of elements after an element in R_j has been searched for. The dotted lines in block B_m indicate that the structures do not necessarily exist.

searched for within the last ℓ distinct searches located in one of the first $\mathcal{O}(\log\log\ell)$ blocks.

Block B_i consists of a list D_i of size w_i, where $w_i = \lceil \alpha 2^i \rceil$ for some appropriate constant α, and three implicit movable dictionaries, L_i, C_i and R_i. We use D_i to pair-encode $\mathcal{O}(2^i)$ bits, used for memory management in the working set dictionary and for storing the data needed between operations in the movable dictionaries L_i, C_i and R_i. Block B_i contains exactly $2 \cdot 2^{2^i} + w_i$ elements, except for the last block B_m that might contain less than $2 \cdot 2^{2^i} + w_i$ elements, as this is the block that grows or shrinks when we insert or delete, respectively.

When an element e is searched for it is moved from its current block B_j to the block B_0. To make room for this in B_0, we move an element from each block B_i to B_{i+1} until we reach the block B_j, where e was originally located. We move elements from R_i to L_{i+1}, for $i = 0, \ldots, j-1$ (see Figure 3.3). Once R_i is empty we move C_i to R_i, and L_i to C_i. Doing this we can guarantee that at all elements in $L_j \cup C_j \cup \bigcup_{i=0}^{j-1} B_i$ have been accessed more recently than the elements in R_j, i.e. least 2^{2^i} distinct elements have been searched for since any element in R_i was last searched for. We can give this guarantee because an element will be located in C_i at least until searches for 2^{2^i} other elements have been performed.

3.3.1 Invariants

Our data structure satisfies the invariants below. Here I.1 to I.4 are about the sizes of data structures and are important for memory management. On the other hand I.5 to I.8 are about the location of elements according to when they were last searched for and are important for achieving the working set property.

I.1 $|C_i| \leq 2^{2^i}$ and $|R_i| \neq 0 \Rightarrow |C_i| = 2^{2^i}$, for all i.

I.2 $|D_i| \leq w_i$ and $|L_i| + |C_i| + |R_i| \neq 0 \Rightarrow |D_i| = w_i$, for all i.

I.3 $|L_i| + |R_i| = 2^{2^i}$, for all $i < m$, and $|L_m| + |R_m| \leq 2^{2^m}$.

I.4 $|L_i| < 2^{2^i}$, for all i.

I.5 All elements searched for since L_i was last empty are contained in L_i, D_i or B_j for some $j < i$.

I.6 For any e in some C_i either at least $|L_i|$ distinct elements have been searched for after e was last searched for or e has never been searched for.

I.7 For any e in some R_i either at least 2^{2^i} distinct elements have been searched for after e was last searched for or e has never been searched for.

I.8 For any e in D_i, L_i or C_i, for $i > 0$, either at least $2^{2^{i-1}}$ distinct elements have been searched for after e was last searched for, or e has never been searched for.

From the invariants we make the following observations:

O.1 $|D_i| = w_i$ for all $i < m$ (from I.2 and I.3).

O.2 $|R_i| > 0$ for all $i < m$ (from I.3 and I.4).

O.3 $|C_i| = 2^{2^i}$ for all $i < m$ (from I.1 and O.2).

O.4 $|B_i| = w_i + 2 \cdot 2^{2^i}$ for all $i < m$ (from O.1, O.3, and I.3).

O.5 For $i > 0$ and any e in B_i, either at least $2^{2^{i-1}}$ distinct elements have been searched for after e was last searched for or e has never been searched for (from I.7 and I.8).

3.3.2 Operations

Our data structure uses the operations shift and find internally, and supports the operations insert, delete, predecessor and search. Below is a detailed description of all operations, pseudo-code for all the operations can be found in Section 3.9.

Shift(j) handles the case when $|R_j| = 0$ and $|L_j| = 2^{2^j}$, i.e. I.4 is violated for block B_j. This is done by discarding R_j, renaming C_j to R_j, renaming L_j to C_j, and creating a new empty L_j. After shift(j) finishes I.4 also holds for B_j.

Find(x) finds the data structure S_i containing the element with key x or returns **none** if no such element exists. Here S_i will be either D_i, L_i, C_i or R_i for some i. This is done by searching for x in the blocks starting with B_0 and going in an incremental linear fashion towards B_m. Within each block, x is searched for in D_i using a linear scan, and the implicit movable dictionaries L_i, C_i and R_i are searched for x using their built-in search operation. As soon as x is found, a reference to the data structure S_i containing the element is returned, and no further blocks are considered. In the case when x is not found in any of the blocks **none** is returned.

Predecessor(x) returns the element e in the data structure with the largest key less than x. This is done for $B_0, \ldots, B_m$ by a linear scan of D_i and invoking the built-in predecessor operation on L_i, C_i and R_i and returning the element among the results with the highest key.

Insert(e) inserts the element e into the data structure. This is done by inserting e into one of the data structures in B_m. It is inserted into D_m if $|D_m| < w_m$. Otherwise, if $|C_m| < 2^{2^m}$ it is inserted into C_m, else it is inserted into R_m. If this makes $|L_m| + |R_m| = 2^{2^m}$, then a new block B_{m+1} is initialized by incrementing m by one.

Delete(x) deletes the element with key x from the data structure. We first check if x is in the dictionary by performing a find(x) operation. If x is not found we return. Here S_j will be one of D_j, L_j, C_j or R_j. If B_m is empty, m is decremented by one. An arbitrary element e is deleted from the first of the structures R_m, C_m, L_m and D_m that is non-empty. If e has key x we return, else the element with key x is deleted from S_j and e is inserted into S_j.

Search(x) returns the element e with key x or **none** if such an element does not exist. This is done by performing a find(x) operation, finding the data structure S_j containing x. If x is not in the data structure then **none** is returned.

If x is found in a data structure S_j then the element e with key x is found by running the built-in search method on S_j. If S_j is either D_0 or L_0 we return e immediately. If $S_j = C_j$ and $|R_j| > 0$, an arbitrary element g is removed from R_j, e is removed from C_j and g is inserted in C_j. In the other case where $S_j \neq C_j$ or $|R_j| = 0$, the element e with key x is deleted from S_j.

In all cases we then proceed by deleting an arbitrary element h from R_{i-1} and inserting it into L_i, for $i = j, \ldots, 1$. In the special case where $i = j$ and $S_j = D_j$ we insert h into D_j instead of L_j. Next we insert e into L_0. Now for $i = 0, \ldots, j$ we check whether $|L_i| = 2^{2^i}$, and if this is the case we perform a shift(i) operation. Finally we return e.

3.4 Memory management

From O.4 we know that any block except the last will contain a fixed number of elements, namely $2 \cdot 2^{2^i} + w_i$. This implies that we can lay out the blocks sequentially in the array, and then we only have to worry about memory management inside each block. The last block B_m can vary in size, and is located at the end of the array where growing and shrinking must occur.

By I.2 we know that D_i will be completely constructed before the other structures are needed, therefore we lay it out sequentially in the beginning of the block. The remaining structures will be laid out sequentially in the order: L_i, C_i, R_i. That is we lay out structures as show in Figure 3.3 from consecutively left to right.

Right before we insert an element into L_i, we move C_i and R_i one position to the right to make room. We can move L_i, C_i or R_i to the right by performing a delete-left operation on an arbitrary element e followed by an insert-right(e). This moving will take time $\mathcal{O}(2^i)$. We do the same when inserting into C_i, but here we only move R_i one position to the right. We never need to move structures to the left.

To perform queries on the substructures in a block B_i we need to store various information in D_i. We need n_{L_i}, n_{C_i} and n_{R_i}: the size of L_i, C_i and R_i, respectively. We store n_{C_i} and n_{R_i} in D_i explicitly using 2^i bits each, whereas n_{L_i} can be computed as $n_{L_i} = |B_i| - w_i - n_{C_i} - n_{R_i}$. Furthermore we store in D_i the $\Theta(2^i)$ bits we allow the movable dictionaries L_i, C_i and R_i to maintain between operations, denoted $data_{L_i}$, $data_{C_i}$ and $data_{R_i}$.

We maintain all these bits in D_i, using pair-encoding. The fields are stored in the following order: $data_{L_i}$, n_{C_i}, $data_{C_i}$, n_{R_i}, $data_{R_i}$. Whenever we add an element to or remove an element from D_i we maintain the ordering of the pair by performing a swap if needed.

To perform an operation on block B_i we need to know the index b_i of the first element, which can be computed as $b_0 = 0$, and $b_i = b_{i-1} + 2 \cdot 2^{2^{i-1}} + w_{i-1}$. We may also need $|D_i|$ which can be computed as $|D_i| = \min(w_i, n - b_i)$, and $|B_i|$ which can be computed as $|B_i| = w_i + 2 \cdot 2^{2^i}$ if $i < m$ and $|B_m| = n - b_m$ otherwise.

Whenever we want to perform an operation on C_i, we first extract n_{L_i}, n_{C_i}, n_{R_i} and $data_{C_i}$ from the pair-encoding in D_i and put them into registers. From the sizes and the value of b_i we can compute the index of the first element in C_i. Using that information we can run the operation on the implicit movable dictionary. Once that is done we write $data_C$ back to the pair-encoding in D_i. Totally this requires $\mathcal{O}(2^i)$ time. We do similarly if we perform an operation on L_i or R_i.

When performing a shift operation we override n_{R_i} and $data_{R_i}$ with n_{C_i} and $data_{C_i}$ and we override n_{C_i} and $data_{C_i}$ with n_{L_i} and $data_{L_i}$. This renames the data structures, initiating a new empty L_i before the old full one, and "deletes" the old empty R_i.

During an insert operation, when D_i increases to w_i, we initialize n_{L_i}, $data_{L_i}$, n_{C_i}, $data_{C_i}$, n_{R_i} and $data_{R_i}$. Finally we calculate m when it is needed as the minimal value where $\sum_{j=0}^{m} 2 \cdot 2^{2^j} + w_j > n$.

3.5 Analysis of the working set structure

We now analyse the working set dictionary, we first argue that all operations maintain the invariants, which proves the correctness of the structure and the operations. Then we analyse the running time and cache-misses that each operation incurs.

3.5.1 Maintaining Invariants

First note that the find and predecessor operations do not alter the structure, so they trivially maintain the invariants. Also note that all the invariants hold for the empty data structure.

In the following C_i will refer to the data structure before the operation and C_i' the same data structure after the operation, similarly for the other data structures and m.

The insert operation First note that all invariants are trivially maintained for $i < m$ since we do not change blocks $B_0, \ldots, B_{m-1}$. Since I.1 held before the operation we know that either $|C_m| = 2^{2^m}$ or $|R_m| = 0$. In the first case we will not insert anything into C_m, but into R_i. In the latter case we will only insert something into R_m if $|C_m| = 2^{2^m}$, so in both cases I.1 is maintained for $i = m$. We note that if $m' = m + 1$, I.1 holds for $i = m'$ since $|R_{m'}| = 0$.

Since I.2 held before the operation we know that either $|L_m| = |C_m| = |R_m| = 0$ or $|D_m| = w_m$. In the first case we only insert something into C_m or R_m if $|D_m| = w_m$. In the second case we do not alter $|D_m|$, so in both cases, I.2 is maintained for $i = m$. If $m' = m+1$, then I.2 holds for $i = m'$ because $|L_m| = |C_m| = |R_m| = 0$. If $m' = m + 1$ then we also get that $|L_m| + |R_m| = 2^{2^m}$ so I.3 holds for $i = m' - 1$.

Since we never insert anything into L_m and all elements we insert into D_m, C_m or R_m by definition have never been searched for, then I.4 to I.8 are maintained for $i = m$.

The delete operation I.1 to I.4 only deal with sizes of data structures. Since we only changed the sizes in B_m, they are maintained for $i < m$. I.1 is maintained for $i = m$ since we only delete from C_m if R_m is empty. I.2 is maintained for $i = m$ since we only delete from D_m if L_m, C_m and R_m are empty. I.3 does not apply for m', not even if $m' = m - 1$. I.4 is maintained since we do not add any elements to L_m. Since we only ever move elements to the left, I.5 is maintained. Since we only touch B_m and B_j, I.6 to I.8 are maintained for all other blocks.

If element e is deleted from L_m, then C_m is empty so I.6 holds for block B_m. Now if we insert the element e into $S_j = C_j$, then it will come from B_m. We have two cases. If $j < m$ then O.5 tells us that at least $2^{2^{m-1}} \geq 2^{2^j}$ distinct elements have been searched for since e was last searched for. By I.4 we know that $2^{2^j} > |L_j|$, so I.6 is maintained for B_j. If on the other hand $j = m$ then if e comes from C_m, L_m or D_m then we just insert e into C_m and I.6 is maintained, so lets assume that e comes from R_m. Here I.4 and I.7 imply that at least $2^{2^m} > |L_m|$ elements have been searched for after e was last searched for (or e has never been searched for), so I.6 is maintained for block B_j.

If element e is inserted into $S_j = R_j$ and $m > j$ then O.5 imply that at least $2^{2^{m-1}} \geq 2^{2^j}$ elements have been searched for after e and I.7 is maintained. Now if element e is inserted into $S_j = R_j$ and $m = j$, then we know that $x \neq e$ and we have deleted and inserted e into $S_j = R_j$ and deleted x, which maintains I.7.

If an element e is inserted into D_j, L_j or C_j, then by O.5 we know that at least $2^{2^{m-1}} \geq 2^{2^{j-1}}$ elements have been searched for after e and I.8 is maintained.

The shift operation We assume that B_j satisfies all invariants except I.4 before $\mathsf{shift}(j)$. Since the shift operation requires that $|L_j| = 2^{2^j}$ and $C'_j = L_j$ I.1 holds for j after the shift operation. Because $|L_j| = 2^{2^j}$ and I.2 holds before the operation we know that $|D_j| = w_j$. Since the shift operation did not change D_j, I.2 also holds for j after the operation. When verifying I.3 we have two cases: if $j = m$, then from I.1 we know that $|C_m| \leq 2^{2^m}$ so $|L'_m| + |R'_m| = 0 + |C_m| \leq 2^{2^m}$ so I.3 holds. Else if $j < m$ then by O.3 we know that $|C_j| = 2^{2^j}$. Now since L'_j is empty and $R'_j = C_j$, then I.3 holds for $j < m$ after the operation. Since L'_j is empty, no elements have been accessed after it was last empty, thus I.5 trivially holds for j. Likewise because $\left|L'_j\right| = 0$, then I.6 is maintained for j. Because $\mathsf{shift}(j)$ assumed $|L_j| = 2^{2^j}$, and because $R'_j = C_j$, then I.6 immediately implies that I.7 holds for j after $\mathsf{shift}(j)$. Lastly, since all elements in L'_j and C'_j come from L_j, and D'_j contains the same elements as D_j then I.8 held for j since it holds before the $\mathsf{shift}(j)$ operation.

The search operation We will show that all invariants except I.4 hold for the search operation, just before we perform the shift operations. This is sufficient since the search operation will maintain all invariants except I.4, and since we perform a shift on every block where I.4 does not hold, then after all the shift operations have been performed, I.4 will hold, along with all the other invariants.

The search operation will change the blocks $B_0, \ldots, B_j$. We observe that an element is moved from B_i to B_{i+1} for $i < j$ and that an element is moved from B_j to B_0. Therefore the number of elements contained in each block is not altered by the search operation.

The only time we delete something from C_i is if the element e, with key x, happens to be located in C_i. In this case we replace the element e from C_i with

some element g we extracted from R_i unless R_i happened to be empty. So either $|C_i|$ remains the same as before or it decreases and then $|R_i| = 0$, so therefore we I.1 is maintained.

We never delete anything from D_0, and when we delete an element from D_i it is replaced by an element from R_{i-1}, so the size of D_i never changes. Also since the size of B_i is maintained for all i, then the sum of the sizes of the remaining data structures in B_i must also be maintained, and therefore I.2 holds.

It is clear that for $0 < i < j$, we always remove an element from R_i and insert an element into L_i, so I.3 is maintained for $0 < i < j$. If the element e with key x was in D_j or L_j, then it is replaced by an element h from R_{j-1}, so the size of all data structures in B_j is also maintained. If the element e with key x was in R_j, then $|L'_j| = |L_j| + 1$ either by inserting an element h from R_{j-1} or in the case where $j = 0$ by inserting e itself, and $|R'_j| = |R_j| - 1$. Otherwise e must come from C_j, and again we have that the size of $|L'_j| = |L_j| + 1$, and $|R'_j| = |R_j| - 1$ unless $j = m$, so I.3 is maintained for $i = j$. We now verify that I.3 holds for $i = 0$, here we have two cases if $j = 0$, then we do not move any elements so I.3 is trivially maintained. Else we have $j > 0$, so we insert e into L_0 so $|L'_0| = |L_0| + 1$, and we move an element from R_0 to B_1 so $|R'_0| = |R_0| - 1$ and I.3 is maintained for $i = 0$. Lastly I.3 is trivially maintained for $i > j$, so I.3 holds for all blocks i.

We now check if I.5 is maintained. First we note that the element e with key x located in S_j being searched for will always be put in L_0 or D_0 so this element cannot violate any L_i. Likewise the moving of element g between C_j and R_j if $S_j = C_j$ also does not affect I.5. Finally when we move an element from R_{i-1} to L_i or D_i, this does not violate I.5 for $i \leq j$ and for $i > j$ we do no change B_i at all, so I.5 is maintained.

Now we want to show that I.6 is maintained. Let g be some element in C'_i, if g was not in C_i then g must have come from R_i and by I.7 at least 2^{2^i} elements have been searched for after g, and we know by I.4 that $|L'_i| \leq |L_i| + 1 \leq 2^{2^i}$, so I.6 holds for g. If on the other hand g was also in C_i then we know by I.6 that at least $|L_i|$ distinct elements have been searched for after g, and if $|L'_i| = |L_i|$ then I.6 is maintained. In the other case $|L'_i| = |L_i| + 1$, we first notice that $i \leq j, S_j \neq D_i$ and $S_j \neq L_i$ since in those cases $|L_i| = |L'_i|$. Now by I.5 this means that e has not been searched for since L_i was last emptied because $e \notin L_i, D_i, B_k$ for $k < i$, so we have yet another distinct, element namely e, that has been searched for since L_i was last emptied. So I.6 holds for g, and I.6 is maintained.

Since we never add new elements to R_i for any i, I.7 is trivially maintained. Also I.8 is trivially maintained for any element g that was also in its corresponding structure before. If $i > 0$ then a new element g added to L_i or D_i comes from R_{i-1} where I.7 implies that at least $2^{2^{i-1}}$ elements have been searched for after g, so I.8 is maintained, and I.8 is trivially maintained for $i = 0$.

3.5.2 Running time and correctness

The find operation If an element e with key x is in the data structure, $\mathsf{find}(x)$ will find it as promised. The element will be located in exactly one of the data structures in one of the blocks, and find looks in all of them.

Let ℓ be the number of distinct elements searched for since we last searched for e, and assume that e is in some block B_j. By O.5 we know that at least $2^{2^{j-1}}$ elements have been searched for after e was last searched for so $\ell \geq 2^{2^{j-1}}$, i.e. $2^j = \mathcal{O}(\log \ell)$.

For each block we use constant time to calculate b_i, $|D_i|$, and whether $i = m$. This can be done since we have already computed b_{i-1} once b_i is needed. The time used for the find operation in block B_i is $\mathcal{O}(2^i)$ doing the linear scan in D_i and $\mathcal{O}(\log 2^{2^i}) = \mathcal{O}(2^i)$ for doing searches in L_i, C_i and R_i from the bounds on the movable dictionary. The total time for doing searches in all the blocks is then

$$\mathcal{O}\left(\sum_{i=0}^{j} 2^i\right) = \mathcal{O}(2^j) = \mathcal{O}(\log \ell).$$

From the cache-oblivious viewpoint we incur $\mathcal{O}(2^i/B)$ cache-misses when searching D_i and $\mathcal{O}(\log_B 2^{2^i}) = \mathcal{O}(2^i/\log B)$ when searching in L_i, C_i and R_i, so in total we incur $\mathcal{O}\left(\sum_{i=0}^{j} 2^i/\log B\right) = \mathcal{O}\left(2^j \log B\right) = \mathcal{O}(\log_B \ell)$ cache-misses for the find operation.

The insert operation First note that we maintain m in such a way that there is always room for at least one element in B_m, so insert will find a place to insert the element. We will use $\mathcal{O}(\log\log \ell)$ time calculating m, $|D_m|$ and b_m. We then insert an element into either D_m - which takes constant time - or into one of the other structure of size at most n - which will take $\mathcal{O}(\log n)$ time from the bounds on the movable dictionary. It might also cause D_m to get size w_m, in which case we will use an additional $\mathcal{O}(\log n)$ time setting up L_m, C_m and R_m. In total it runs in $\mathcal{O}(\log n)$ time.

From the cache-oblivious viewpoint we incur $\mathcal{O}(\log(n)/B)$ cache-misses when calculating/maintaining m, $|D_m|$ and b_m and finding the place to insert the element. Then we either insert into D_m taking $\mathcal{O}(\log(n)/B)$ cache-misses, or we insert into one of the structures of size at most n which takes $\mathcal{O}(\log_B n)$ cache-misses from the bounds on the movable dictionary. Finally we might also incur $\mathcal{O}(\log(n)/B)$ cache-misses when setting up L_m, C_m and R_m. In total we incur at most $\mathcal{O}(\log_B n)$ cache-misses.

The delete operation The delete operation always deletes the element that it promises to delete. Other elements may also be removed but they are always reinserted before the operation finishes. We will use $\mathcal{O}(\log\log n)$ time calculating m, $|D_i|$ and b_i for all i. The delete operation performs a find operation running in time $\mathcal{O}(\log \ell)$ and at most two deletes and one insert on movable dictionaries of size n so it uses $\mathcal{O}(\log n)$ time.

From the cache-oblivious viewpoint we incur $\mathcal{O}(\log(n)/B)$ cache-misses when maintaining m, $|D_i|$ and b_i for all i. We also perform a find operation incurring $\mathcal{O}(\log_B \ell)$ cache-misses and at most two deletes and one insert on movable dictionaries which incur $\mathcal{O}(\log_B n)$ cache-misses. In total we incur $\mathcal{O}(\log_B n)$ cache-misses.

The predecessor operation We will use $\mathcal{O}(\log\log n)$ time to calculate m, $|D_i|$ and b_i for all i, then we perform a predecessor search on each data structure on all the levels $0, \ldots, \log\log n$ using $\mathcal{O}(\sum_{i=0}^{\log\log n} 3\log(2^{2^i}) + 2^i) = \mathcal{O}(\log n)$ time. Since it finds the predecessor of x in every structure it will find the real predecessor in one of them, this must be the maximal value found, and if we keep track of the maximal from all the blocks we have visited then this is already counted in the $\mathcal{O}(\log n)$ time bound.

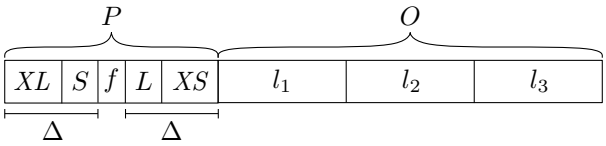


Figure 3.4: Memory layout of the static finger search dictionary.

From the cache-oblivious viewpoint we incur $\mathcal{O}(\log(n)/B)$ cache-misses when calculating m, $|D_i|$ and b_i for all i. We also perform a predecessor search on all levels $0, \ldots, \log\log n$ which incurs $\mathcal{O}(\sum_{i=0}^{\log\log n} 3\log_B(2^{2^i}) + 2^i/B) = \mathcal{O}(\log_B n)$ cache-misses. In total we then incur $\mathcal{O}(\log_B n)$ cache-misses.

The search operation The search operation uses find, so it *will* find the location of the element with key x if it is in the structure, and it will return the correct answer. Furthermore it never deletes an element without inserting it again so all elements remain somewhere in the data structure. Search performs a find operation taking time $\mathcal{O}(\log \ell)$. It performs at most a constant number of shifts, insertions and deletions on each of the $\mathcal{O}(\log\log \ell)$ blocks looked at, taking $\mathcal{O}(\sum_{i=0}^{\log\log \ell} 2^i) = \mathcal{O}(\log \ell)$ time. We will use $\mathcal{O}(\log\log n)$ time calculating m, $|D_i|$ and b_i for all $i \leq \log\log \ell$. So in total it uses $\mathcal{O}(\log \ell)$ time.

From the cache-oblivious viewpoint we incur $\mathcal{O}(\log_B \ell)$ cache-misses on the find operation, we then perform a constant number of shift, insert and delete operations on each of the $\mathcal{O}(\log\log \ell)$ blocks incurring $\mathcal{O}(\sum_{i=0}^{\log\log \ell} \log_B 2^{2^i}) = \mathcal{O}(\log_B \ell)$ cache-misses. In total we incur $\mathcal{O}(\log_B \ell)$ cache-misses.

3.6 Static finger search

In this section we present a simple *change-finger implicit dictionary*, achieving an optimal trade off between the time for search and changer-finger.

Given some function $q(t, n)$, as defined in Section 3.1, we are aiming for a search time of $\mathcal{O}(q(t, n))$. Let $\Delta = Z_q(n)$, recall that $Z_q(n)$ is the smallest rank distance Δ, such that $q(\Delta, n) > \log \frac{n}{2}$. Note that we are allowed to use $\mathcal{O}(\log n)$ time searching for elements with rank-distance $t \geq \Delta$ from the finger, since $q(t, n) = \Omega(\log n)$ for $t \geq \Delta$.

Intuitively, we start with a sorted list of elements. We cut the $2\Delta + 1$ elements closest to f (f being in the center), from this list, and swap them with the first $2\Delta + 1$ elements, such that the finger element is at position $\Delta+1$. The elements that were cut out form the *proximity structure* P, the rest of the elements are in the *overflow structure* O (see Figure 3.4).

A search for x is performed by first doing an exponential search for x in the proximity structure, and if x is not found there, by doing binary searches for it in the remaining sorted sequences.

The proximity structure consists of sorted lists $XS \prec S \prec \{f\} \prec L \prec XL$. The list S contains the up to Δ elements smaller then f that are closest to f w.r.t. rank distance. The list L contains the up to Δ closest to f, but larger than f. Both are sorted in ascending order. The list XL contains a possibly empty sorted sequence of elements larger than elements from L, and XS contains a possibly empty sorted sequence of elements smaller than elements from S. Here $|XL|+|S| = \Delta = |L|+|XS|$,

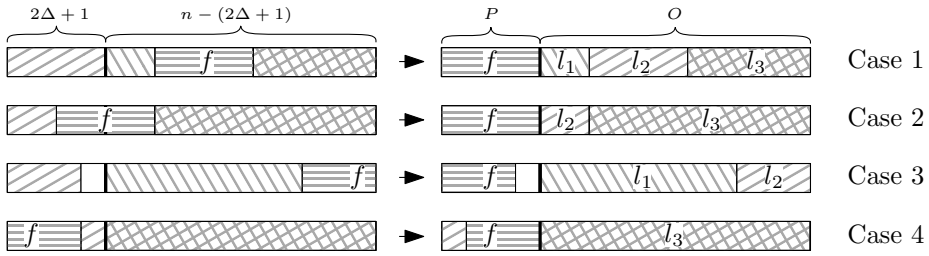


Figure 3.5: Cases for the change-finger operation. The left side is the sorted array. In all cases the horizontally marked segment contains the new finger element and must be moved to the beginning. In the final two cases, there are not enough elements around f so P is padded with what was already there. The emphasized bar in the array is the $2\Delta + 1$ break point between the proximity structure and the overflow structure.

$|S| = \min\{\Delta, \text{rank}(f) - 1\}$ and $|L| = \min\{\Delta, n - \text{rank}(f)\}$. The overflow structure consists of three sorted sequences $l_2 \prec l_1 \prec \{f\} \prec l_3$, each possibly empty. The full ordering of all sequences is $l_2 \prec XS \prec l_1 \prec S \prec \{f\} \prec L \prec XL \prec l_3$.

To perform a change-finger operation, we first revert the array back to one sorted list and the index of f is found by doing a binary search. Once f is found there are 4 cases to consider, as illustrated in Figure 3.5. Note that in each case, at most $2|P|$ elements have to be moved. Furthermore the elements can be moved such that at most $\mathcal{O}(|P|)$ swaps are needed. In particular cases 2 and 4 can be solved by a constant number of list reversals.

For reverting to a sorted array and for doing search, we need to compute the lengths of all sorted sequences. These lengths uniquely determine the case used for construction, and the construction can thus be undone. To find $|S|$ a binary search for the split point between XL and S, is done within the first Δ elements of P. This is possible since $S \prec \{f\} \prec XL$. Similarly $|L|$ and $|XS|$ can be found. The separation between l_2 and l_3, can be found by doing a binary search for f in O, since $l_1 \cup l_2 \prec \{f\} \prec l_3$. Finally if $|l_3| < |O|$, the separation between l_1 and l_2 can be found by a binary search, comparing candidates against the largest element from l_2, since $l_2 \prec l_1$.

When performing the search operation for some key k, we first determine if $k < f$. If this is the case, an exponential search for k in S is performed. We can detect if we have crossed the boundary to XL, since $S \prec \{f\} \prec XL$. If the element is found it can be returned. If $k > f$ we do an identical search in L. Otherwise the element is neither located in S nor L, and therefore $d(k, f) > \Delta$. All lengths are reconstructed as above, and the element is searched for using binary search in XL and l_3 if $k > f$ and, otherwise in XS, l_1 and l_2.

Analysis The change-finger operation first computes the lengths of all lists in $\mathcal{O}(\log n)$ time. The case used for constructing the current layout is then identified and reversed in $\mathcal{O}(\Delta)$ time. We locate the new finger f' by binary search in $\mathcal{O}(\log n)$ time and afterwards the $\mathcal{O}(\Delta)$ elements closest to f' are moved to P. We get $\mathcal{O}(\Delta + \log n)$ time for change-finger.

For searches there are two cases to consider. If $t \leq \Delta$, it will be located by the exponential search in P in $\mathcal{O}(\log t) = \mathcal{O}(q(t, n))$ time, since by assumption $q(t, n) \geq$

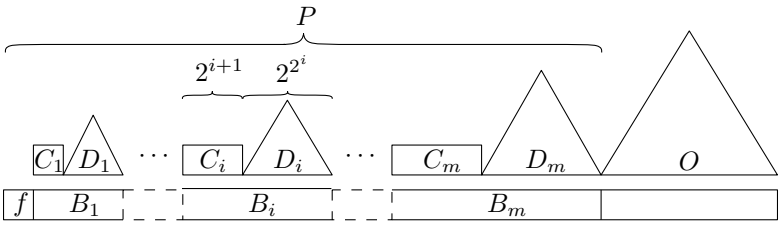


Figure 3.6: Memory layout of dynamic finger search dictionary.

$\log t$. Otherwise the lengths of the sorted sequences will be recovered in $\mathcal{O}(\log n)$ time, and a constant number of binary searches will be performed in $\mathcal{O}(\log n)$ time total. Since $t \geq \Delta \Rightarrow q(t,n) \geq \log \frac{n}{2}$, we again get a search time of $\mathcal{O}(q(t,n))$.

3.7 A dynamic structure

For any function $q(t,n)$, as defined in the introduction, we present a dynamic *change-finger implicit dictionary* that supports change-finger, search, insert and delete in $\mathcal{O}(\Delta \log n), \mathcal{O}(q(t,n)), \mathcal{O}(\log n)$ and $\mathcal{O}(\log n)$ time respectively, where $\Delta = Z_q(n)$ and n is the number of elements when the operation was started.

The data structure consists of two parts: a *proximity structure* P which contains the elements near f and an *overflow structure* O which contains elements further from f w.r.t. rank distance. We partition P into several smaller structures $B_1, \ldots, B_m$. Elements in B_i are closer to f than elements in B_{i+1}. The overflow structure O is an *implicit movable dictionary* as described in the Section 3.2. See Figure 3.6 for the layout of the data structure. During a change-finger operation the proximity structure is rebuilt such that $B_1, \ldots, B_m$ correspond to the new finger, and the remaining elements are put in O.

The total size of P is $2\Delta + 1$. The i'th block B_i consists of a counter C_i and an implicit movable dictionary D_i. The counter C_i contains a pair encoded number c_i, where c_i is the number of elements in D_i smaller than f. The sizes within B_i are $|C_i| = 2^{i+1}$ and $|D_i| = 2^{2^i}$, except in the final block B_m where they might be smaller (B_m might be empty). In particular we define:

$$m = \min \left\{ m' \in \mathbb{N} \,\middle|\, \sum_{i=0}^{m'} \left(2^{i+1} + 2^{2^i} \right) > 2\Delta \right\} .$$

We will maintain the following invariants for the structure:

I.1 $\forall i < j, e_1 \in B_i, e_2 \in B_j : d(f, e_1) < d(f, e_2)$

I.2 $\forall e_1 \in B_1 \cup \cdots \cup B_m,\ e_2 \in O : d(f, e_1) \leq d(f, e_2)$

I.3 $|P| = 2\Delta + 1$

I.4 $|C_i| \leq 2^{i+1}$

I.5 $|D_i| > 0 \Rightarrow |C_i| = 2^{i+1}$

I.6 $|D_m| < 2^{2^m}$ and $\forall i < m : |D_i| = 2^{2^i}$

I.7 $|D_i| > 0 \Rightarrow c_i = |\{e \in D_i \mid e < f\}|$

We observe that the above invariants imply:

O.1 $\forall i < m : |B_i| = 2^{i+1} + 2^{2^i}$ (From I.5 and I.6)

O.2 $|B_m| < 2^{m+1} + 2^{2^m}$ (From I.4 and I.6)

O.3 $d(e, f) \leq 2^{2^k-1} \leq \Delta \Rightarrow e \in B_j$ for some $j \leq k$ (From I.1 – I.6)

3.7.1 Block operations

The following operations operate on a single block and are internal helper functions for the operations described in the next subsection.

block_delete(k, B_i): Removes the element e with key k from the block B_i. This element *must* be located in B_i. First we scan C_i to find e. If it is not found it must be in D_i, so we delete it from D_i. If $e < f$ we decrement c_i. In the case where $e \in C_i$ and D_i is nonempty, an arbitrary element g is deleted from D_i and if $g < f$ we decrement c_i. We then overwrite e with g, and fix C_i to encode the new number c_i. In the final case where $e \in C_i$ and D_i is empty, we overwrite e with the last element from C_i.

block_insert(e, B_i): Inserts e into block B_i. If $|C_i| < 2^{i+1}$, e is inserted into C_i and we return. Otherwise we insert e into D_i. If D_i was empty we set $c_i = 0$. In either case if $e < f$ we increment c_i.

block_search(k, B_i): Searches for an element e with key k in the block B_i. We scan C_i for e, if it is found we return it. Otherwise if D_i is nonempty we perform a search on it, to find e and we return it. If the element is not found `nil` is returned.

block_predecessor(k, B_i): Finds the predecessor element for the key k in B_i. Do a linear scan through C_i and find the element l_1 with largest key less than k. Afterwards do a predecessor search for key k on D_i, call the result l_2. Return $\max(l_1, l_2)$, or that no element in B_i has key less than k.

3.7.2 Operations

In order to maintain correct sizes of P and O as the entire structure expands or contracts a rebalance operation is called in the end of every insert and delete operation. This is an internal operation that does not require I.3 to be valid before invocation.

rebalance(): Balance B_m such that the number of elements in P less than f is as close to the number of elements greater than f as possible. We start by evaluating $\Delta = Z_q(n)$, the new desired proximity size. Let s be the number of elements in B_m less than f which can be computed as $c_m + |\{e \in C_m \mid e < f\}|$. While $2\Delta + 1 > |P|$ we move elements from O to P. We move the predecessor of f in O from O to B_m if $O \prec \{f\} \vee (s < \frac{|B_m|}{2} \wedge \neg(\{f\} \prec O))$ and otherwise we move the successor of f to O. While $2\Delta + 1 < |P|$ we move elements from B_m to O. We move the largest element from B_m to O if $s < \frac{B_m}{2}$. Otherwise we move the smallest element.

change-finger(k): To change the finger of the structure to k, we first insert every element of $B_m \ldots B_1$ into O. We then remove the element e with key k from O, and place it at index 1 as the new f, and finish by performing rebalance.

insert(e): Assume $e > f$. The case $e < f$ can be handled similarly. Find the first block B_i where e is smaller than the largest element l_i from B_i (which can be found using a predecessor search) or $l_i < f$. Now if $l_i > f$ for all blocks $j \geq i$, block_delete

the largest element and block_insert it into B_{j+1}. In the other case where $l_i < f$ for all blocks $j \geq i$, block_delete the smallest element and block_insert it into B_{j+1}. The final element that does not have a block to go into, will be put into O, then we put e into B_i. In the special case where e did not fit in any block, we insert e into O. In all cases we perform rebalance.

delete(k): We perform a block_search on all blocks and a search in O to find out which structure the element e with key k is located in. If it is in O we just delete it from O. Otherwise assume $k < f$ (the case $k > f$ can be handled similarly), and assume that e is in B_i, then block_delete e from B_i. For each $j > i$ we block_delete the predecessor of f in B_j, and insert it into B_{j-1} (in the case where there is no predecessor, we block_delete the successor of f instead). We also delete the predecessor of f from O and insert it in B_m. The special case where $k = f$, is handled similarly to $k < f$, we note that after this the predecessor of f will be the new finger element. In all cases we perform a rebalance.

search(k), predecessor(k) and successor(k), all follow the same general pattern. For each block B_i starting from B_1, we compute the largest and the smallest element in the block. If k is between these two elements we return the result of block_search, block_predecessor or block_successor respectively on B_i, otherwise we continue with the next block. In case k is not within the bounds of any block, we return the result of search(k), predecessor(k) or successor(k) respectively on O.

3.7.3 Analysis

By the invariants, we see that every C_i and D_i except the last, have fixed size. Since O is a movable dictionary it can be moved right or left as this final C_i or D_i expands or contracts. Thus the structure can be maintained in a contiguous memory layout.

The correctness of the operations follows from the fact that I.1 and I.2, implies that elements in B_j or O are further away from f than elements from B_i where $i < j$. We now argue that search runs in time $\mathcal{O}(q(t,n))$. Let e be the element we are searching for. If e is located in some B_i then at least half the elements in B_{i-1} will be between f and e by I.1. We know from O.1 that $t = d(f,e) \geq \frac{|B_{i-1}|}{2} \geq 2^{2^{i-1}-1}$. The time spent searching is $\mathcal{O}(\sum_{j=1}^{i} \log |B_j|) = \mathcal{O}(2^i) = \mathcal{O}(\log t) = \mathcal{O}(q(t,n))$. If on the other hand e is in O, then by I.3 there are $2\Delta + 1$ elements in P, of these at least half are between f and e by I.2, so $t \geq \Delta$, and the time used for searching is $\mathcal{O}(\log n + \sum_{j=1}^{k} \log |B_j|) = \mathcal{O}(\log n) = \mathcal{O}(q(t,n))$. The last equality follows by the definition of Z_q. The same arguments work for predecessor and successor.

Before the change-finger operation the number of elements in the proximity structure by I.3 is $2\Delta+1$. During the operation all these elements are inserted into O, and the same number of elements are extracted again by rebalance. Each of these operations are just insert or delete on a movable dictionary or a block taking time $\mathcal{O}(\log n)$. In total we use time $\mathcal{O}(\Delta \log n)$.

Finally to see that both Insert and Delete run in $\mathcal{O}(\log n)$ time, notice that in the proximity structure doing a constant number of queries in every block is asymptotically bounded by the time to do the queries in the last block. This is because their sizes increase double-exponentially. Since the size of the last block is bounded by n we can guarantee $\mathcal{O}(\log n)$ time for doing a constant number of queries on every block (this includes predecessor/successor queries). In the worst case, we need to insert an element in the first block of the proximity structure, and

"bubble" elements all the way through the proximity structure and finally insert an element in the overflow structure. This will take $\mathcal{O}(\log n)$ time. At this point we might have to rebalance the structure, but this merely requires deleting and inserting a constant number of elements from one structure to the other, since we assumed $Z_q(n)$ and $Z_q(n+1)$ differ by at most a constant. Deletion works in a similar manner.

3.8 Lower bounds

To prove our lower bounds we use an abstracted version of the strict implicit model. The strict model requires that *nothing* but the elements and the number of elements are stored between operations, and that during computation elements can only be used for comparison. With these assumptions a decision tree can be formed for a given n, where nodes correspond to element comparisons and element loads, and leaves contain the answers. Note that in the weak model a node could probe a cell containing an integer, giving it a degree of n, which prevents any of our lower bound arguments.

Lemma 3.1. *Let* $\mathtt{A}$ *be an operation on an implicit data structure of length* n*, running in time* τ *worst case, that takes any number of keys as arguments. Then there exists a set* $\mathcal{X}_{\mathtt{A},n}$ *of size* 2^τ*, such that executing* $\mathtt{A}$ *with any arguments will touch only cells from* $\mathcal{X}_{\mathtt{A},n}$ *no matter the content of the data structure.*

Proof. Before loading any elements from the data structure, A can reach only a single state which gives rise to a root in a decision tree. When A is in some node s, the next execution step may load some cell in the data structure, and transition into another fixed node, or A may compare two previously loaded elements or arguments, and given the result of this comparison transition into one of two distinct nodes. It follows that the total number of nodes A can enter within its τ steps is $\sum_{i=0}^{\tau-1} 2^i < 2^\tau$. Now each node can access at most one cell, so it follows that at most 2^τ different cells can be probed by any execution of A within τ steps. □

Observe that no matter how many times an operation that takes at most τ time is performed, it will only be able to reach the same set of cells, since the decision tree is the same for all invocations.

Theorem 3.2. *For any* change-finger implicit dictionary *with a search time of* $q(t,n)$ *as defined in Section 3.1,* change-finger *requires* $\Omega(Z_q(n) + \log n)$ *time.*

Proof. Let $e_1 \dots e_n$ be a set of elements in sorted order with respect to the keys $k_1 \dots k_n$. Let $t = Z_q(n) - 1$. By definition $q(t+1,n) \geq \log \frac{n}{2} > q(t,n)$. Consider the following sequence of operations:

for $i = 0 \dots \frac{n}{t}$:
  change-finger(k_{it})
  for $j = 0 \dots t-1$: search(k_{it+j})

Since the rank distance of any query element is at most t from the current finger and q is non-decreasing each search operation takes time at most $q(t,n)$. By Lemma 3.1 there exists a set $\mathcal{X}$ of size $2^{q(t,n)}$ such that all queries only touch cells in $\mathcal{X}$. We note that $|\mathcal{X}| \leq 2^{q(t,n)} \leq 2^{\log(n/2)} = \frac{n}{2}$.

Since all n elements were returned by the query set, the change-finger operations must have copied at least $n - |\mathcal{X}| \geq \frac{n}{2}$ elements into $\mathcal{X}$. We performed $\frac{n}{t}$ change-finger operations, thus on average the change-finger operations must have moved at least $\frac{t}{2} = \Omega(Z_q(n))$ elements into $\mathcal{X}$.

For the $\log n$ term in the lower bound, we consider the sequence of operations change-finger(k_i) followed by search(k_i) for i between 1 and n. Since the rank distance of any search is 0 and $q(0, n) < \log \frac{n}{2}$ (by assumption), we know from Lemma 3.1 that there exists a set $\mathcal{X}_s$ of size at most $2^{\log(n/2)}$, such that search only touches cells from $\mathcal{X}_s$. Assume that change-finger runs in time $c(n)$, then from Lemma 3.1 we get a set $\mathcal{X}_c$ of size at most $2^{c(n)}$ such that change-finger only touches cells from $\mathcal{X}_c$. Since every element is returned, the cell initially containing the element must be touched by either change-finger or search at some point, thus $|\mathcal{X}_c| + |\mathcal{X}_s| \geq n$. We see that $2^{c(n)} \geq |\mathcal{X}_c| \geq n - |\mathcal{X}_s| \geq n - 2^{\log(n/2)} = 2^{\log(n/2)}$, i.e. $c(n) \geq \log \frac{n}{2}$. □

Theorem 3.3. *For a* change-finger implicit dictionary *with* search *time $q'(t, n)$, where q' is nondecreasing in both t and n, it holds that $q'(t, n) \geq \log t$.*

Proof. Let $e_1 \dots e_n$ be a set of elements with keys $k_1 \dots k_n$ in sorted order. Let $t \leq n$ be given. First perform change-finger(k_1), then for i between 1 and t perform search(k_i). From Lemma 3.1 we know there exists a set $\mathcal{X}$ of size at most $2^{q'(t,n)}$, such that any of the search operations touch only cells from $\mathcal{X}$ (since any element searched for has rank distance at most t from the finger). The search operations return t distinct elements so $t \leq |\mathcal{X}| \leq 2^{q'(t,n)}$, and $q'(t, n) \geq \log t$. □

Theorem 3.4. *For* finger-search implicit dictionary*, the* finger-search *operation requires at least $g(t, n) \geq \log n$ time for any rank distance $t > 0$ where $g(t, n)$ is nondecreasing in both t and n.*

Proof. Let $e_1 \dots e_n$ be a set of elements with keys $k_1 \dots k_n$ in sorted order. First perform finger-search(k_1), then perform finger-search(k_i) for i between 1 and n. Now for all queries except the first, the rank distance $t \leq 1$ and by Lemma 3.1 there exists a set of memory cells $\mathcal{X}$ of size $2^{g(1,n)}$ such that all these queries only touch cells in $\mathcal{X}$. Since all elements are returned by the queries we have $|\mathcal{X}| = n$, so $g(1, n) \geq \log n$, since this holds for $t = 1$ it holds for all t. □

We can conclude that it is not possible to achieve any form of meaningful finger-search in the strict implicit model. The static *change-finger implicit dictionary* from Section 3.6 is by Theorem 3.2 optimal within a constant factor, with respect to the search to change-finger time trade off, assuming the running time of change-finger depends only on the size of the structure.

Theorem 3.5. *For a working set dictonary with a* search *time of $q'(\ell, n)$, where q' is nondecreasing in both t and n, it holds that $q'(\ell, n) \geq \log \ell$.*

Proof. Let $e_1 \dots e_n$ be a set of elements with keys $k_1 \dots k_n$ in sorted order. First perform search(k_i) for i between 1 and ℓ, then perform search(k_i) for i between 1 and ℓ again in the same order.

We will look at the last ℓ searches, and note that the working set number of each element search of is less then ℓ (it is in fact exactly ℓ).

From Lemma 3.1 we know there exists a set $\mathcal{X}$ of size at most $2^{q'(\ell,n)}$, such that any of the search operations touch only cells from $\mathcal{X}$. The search operations return ℓ distinct elements so $\ell \leq |\mathcal{X}| \leq 2^{q'(\ell,n)}$, and $q'(t, n) \geq \log \ell$. □

3.9 Pseudocode

The following section contains psudocode for the various algorithms described in this chapter.

Algorithms for the dynamic workingset dictonary

Algorithm 3.1: shift(j)

Require: $|R_j| = 0$
$R_j \leftarrow C_j$
$C_j \leftarrow L_j$
$L_j \leftarrow \emptyset$

Algorithm 3.2: find(x)

for $i = 0 \dots m$:
  if D_i.contains(x): **return** D_i
  else if L_i.contains(x): **return** L_i
  else if C_i.contains(x): **return** C_i
  else if R_i.contains(x): **return** R_i
return none

Algorithm 3.3: insert(e)

if $|D_m| < W_m$: D_m.insert(e)
else if $|C_m| < 2^{2^m}$: C_m.insert(e)
else: R_m.insert(e)
if $|L_m| + |R_m| = 2^{2^m}$: $m \leftarrow m + 1$

Algorithm 3.4: delete(x)

$S_j \leftarrow$ find(x)
if $|B_m| = 0$: $m \leftarrow m - 1$
if $|R_m| > 0$: $e \leftarrow R_m$.deleteOne()
else if $|C_m| > 0$: $e \leftarrow C_m$.deleteOne()
else if $|L_m| > 0$: $e \leftarrow L_m$.deleteOne()
else: $e \leftarrow D_m$.deleteOne()
if key(e) $\neq x$: S_j.replace(x, e)

Algorithm 3.5: predecessor(x)

```
e ← none
for i = 0 ... m:
   e ← max(e, D_i.predecessor(x))
   e ← max(e, L_i.predecessor(x))
   e ← max(e, C_i.predecessor(x))
   e ← max(e, R_i.predecessor(x))
return e
```

Algorithm 3.6: search(x)

```
S_j ← find(x)
if S_j = none: return none
e ← S_j.find(x)
if S_j = D_0 ∨ S_j = L_0: return e
if S_j = C_j ∧ |R_j| > 0:
   g ← R_j.deleteOne()
   C_j.replace(x, g)
else
   S_j.delete(x)
for i = j − 1 ... 0:
   h ← R_i.deleteOne()
   if i + 1 = j ∧ S_j = D_j: D_j.insert(h)
   else: L_{i+1}.insert(h)
L_0.insert(e)
for i = 0 ... j:
   if |L_i| = 2^{2^i}: shift(i)
return e
```

4

Computing Multiresolution Rasters

This chapter contains [7] which is joint work with Lars Arge, Gerth Stølting Brodal and Constantinos Tsirogiannis. The paper is included with major changes in notation, also several proofs where extended.

4.1 Introduction

Rasters are one of the most common formats for modelling spatial data. A raster is a 2-dimensional grid of square cells where each cell is assigned a real value. Among other applications, rasters are used to represent real-world terrains; in this case each cell corresponds to a region of a terrain, and the value of the cell indicates the average height of the terrain in this region. Today, it is possible to acquire massive rasters that represent terrains with very fine resolution; the size of each cell in such a raster can be less than one square meter. Yet, studying a terrain in such a small scale might lead to wrong conclusions. This happens for example when we want to identify landforms on terrains; when we study a terrain at a scale of a few meters, we might identify many small peaks concentrated within a small area. Yet, when looking on a larger scale, these peaks may be a part of another landform; for instance a rough ridge, or a valley.

To tackle this problem, we need to have a method that can analyse the same raster in many different scales. Fisher *et al.* [32] use such a method in their landform classification algorithm; their algorithm constructs multiple rasters R_s, where a cell c of R_s covers the same region as $s \times s$ cells of the original fine-resolution raster R. The value assigned to c is equal to the average of the values of the original $s \times s$ cells. Given the constructed rasters R_s, it is then possible to search for landforms at different scales.

Reconstructing a raster in different resolutions is an important tool for many other scientific applications; in remote sensing, Woodcock and Strahler [69] introduced an algorithm to extract the average size of tree canopies in grayscale images of forests. Here, an image is represented by a raster of square pixels, where each pixel is assigned a grayscale value. Their algorithm reconstructs many instances of a given image raster, in exactly the same way as the algorithm of Fisher *et al.* constructs different instances of a terrain raster. For their application, it is critical to construct one instance of the image for every pixel size which is an integer multiple of the pixel size in the original image, until a single pixel covers almost the entire image. This approach has been also used in other image processing algorithms [14].

Therefore, all of the different applications that we described above lead to the same algorithmic problem; let R be a raster that consists of $\sqrt{N} \times \sqrt{N}$ cells. For every integer $s \in \{2, 3, \ldots, \sqrt{N}\}$ we want to compute a raster R_s of $\lceil \sqrt{N}/s \rceil \times$

$\lceil \sqrt{N}/s \rceil$ cells where each cell of R_s stores the average of the values of the $s \times s$ cells of R that cover the same region.

External Memory Algorithms As already mentioned, today many available raster datasets are massive, and may consist of terabytes of data. A raster of this size cannot fit entirely in the main memory of a normal computer; thus, it can only be stored entirely in the hard disk. When we want to process the dataset, we have to transfer blocks of data from the disk to the main memory. We call such a block transfer an I/O-operation, or an I/O for short. Unfortunately, an I/O can take the same time as a million CPU operations. Thus, when designing an algorithm that may process such a large dataset, we want to minimise the number of block transfers that are required to process the full dataset.

For this reason, Aggarwal and Vitter [5] introduced a computational model that takes into account the number of block transfers between the disk and the main memory. This model considers two important parameters: the size of the internal memory M, and the maximum size B of a block of data that we can transfer from/to the disk. The efficiency of an algorithm in this model is equal to the number of I/Os that the algorithm requires during its execution. We call this concept of efficiency the *I/O-efficiency* of the algorithm. The I/O-efficiency of an algorithm is expressed as a function of the input size N, but also of the block size B and memory size M. To scan a set of N records stored in the disk we need $\mathcal{O}(\text{scan}(N))$ I/Os, where $\text{scan}(N) = N/B$. To sort a set of N records we need $\mathcal{O}(\text{sort}(N))$ I/Os, where $\text{sort}(N) = N/B \log_{M/B} N/B$.

Today computers contain several layers of memory; these include layers of cache used between the main memory of the computer and the processor. In this context, the values of parameters M and B differ for every pair of consecutive layers of cache that we consider. Then, to minimise the number of block transfers between all layers, the algorithm must be designed so that it achieves an optimal I/O-performance without knowing the parameters M and B. The algorithms that have this property are known as *cache-oblivious* algorithms [37].

When designing algorithms in the I/O Model, we take grate care in moving the right data in and out of memory in block sizes of B. However such *paging strategies* cannot be used in the cache-oblivious model, where instead we assume that the optimal paging strategy is used. When analysing our algorithm for a specific M and B, we will give an example of a paging strategy that would work for this M and B, and then conclude that the optimal strategy must work at least as well.

Previous Results For the problem of computing multiple resolution instances of a given raster, we study the case where the raster does not fit in the main memory of the computer. We want to design an external memory algorithm for this problem that has optimal performance both in terms of I/Os and in terms of CPU operations. In a previous paper, Arge *et al.* [9] proposed two external memory algorithms for this problem; the first algorithm requires $\mathcal{O}(\text{sort}(N))$ I/Os and $\mathcal{O}(N \log N)$ CPU time, and is easy to implement. Their second algorithm requires $\mathcal{O}(\text{scan}(N))$ I/Os and $\mathcal{O}(N)$ CPU time, which is obviously optimal. Yet, this algorithm assumes that M is at least $\Theta(B^{1+\varepsilon})$ for some selected $\varepsilon > 0$. This algorithm is *cache-aware*, which means that M and B should be known to the algorithm to achieve this performance. Moreover, this algorithm has a strong limitation when it comes to

its implementation; it requires that $\Theta(B)$ files are open simultaneously during its execution. Nowadays, B can be as large as a few million units, while most operating systems can maintain only a relatively small number of files open at the same time (usually around a thousand).

Our Results In this chapter we present a new, cache-oblivious algorithm that achieves the optimal performance of $\mathcal{O}(\text{scan}(N))$ I/Os and $\mathcal{O}(N)$ CPU time, without making any assumptions on the size of the main memory; that is it performs $\mathcal{O}(\text{scan}(N))$ I/Os even when $M = \mathcal{O}(B)$. The new algorithm is easy to implement; we have developed a purely cache-oblivious implementation of the algorithm, and we have tested its performance against an implementation of the algorithm of Arge *et al.* that requires $\mathcal{O}(\text{sort}(N))$ I/Os. Recall that the $\mathcal{O}(\text{scan}(N))$ algorithm of Arge *et al.* is not practically implementable due to limitations of today's operating systems. The new algorithm performs extremely well and, as expected, clearly outperforms the older approach. We consider this to be a solid proof that non-trivial cache-oblivious algorithms can be implemented to perform efficiently in practice, and be used in real-world applications in the place of standard cache-aware implementations.

4.2 Description of the algorithm

Preliminaries For a raster R we denote by $R[i, j]$ the cell that appears in the i-th row and j-th column of R. We use $|R|$ to indicate the number of cells of this raster. We assume that R is a square; it consists of $\sqrt{N}$ rows and $\sqrt{N}$ columns of cells. Yet, it is easy to show that our analysis holds also for rasters that do not have an equal number of rows and columns. Given a cell $R[i, j]$ of R, consider the set of cells $R[k, \ell]$ for which it holds that $1 \leq k \leq i$ and $1 \leq \ell \leq j$. We denote the sum of the values of these cells by $\text{psum}(i, j)$, that is:

$$\text{psum}(i, j) = \sum_{\substack{1 \leq k \leq i \\ 1 \leq \ell \leq j}} R[k, \ell].$$

The value $\text{psum}(i, j)$ is the so-called *prefix sum* of cell $R[i, j]$.

Let R be a raster of dimensions $\sqrt{N} \times \sqrt{N}$, and let s be an integer such that $1 < s \leq \sqrt{N}$. We define R_s as the raster of dimensions $\lceil \sqrt{N}/s \rceil \times \lceil \sqrt{N}/s \rceil$ such that the value of any cell $R_s[i, j]$ is equal to the average value of all cells $R[k, \ell]$ for which we have that $(i-1)s+1 \leq k \leq is$ and $(j-1)s+1 \leq \ell \leq js$. We say that R_s is the *scale instance* of R at s, and we call s the *scale* of this instance. Considering the size of a scale instance R_s, we observe that as we increase s the number of cells of R_s decreases quadratically. In fact, Arge *et al.* [9] showed that the total size of all scale instances R_s is $\Theta(N)$. We retrieve the following lemma from their paper.

Lemma 4.1. *Given a raster R of $\sqrt{N} \times \sqrt{N}$ cells, the total number of cells for all rasters R_s with $2 \leq s \leq \sqrt{N}$ is less than $0.65 \cdot N$.*

Proof. The total number of cells for all rasters R_s is:

$$\sum_{s=2}^{\sqrt{N}} \frac{N}{s^2} < N \left(\sum_{s=1}^{\infty} \frac{1}{s^2} - 1 \right) = N \left(\frac{\pi^2}{6} - 1 \right).$$

□

4.2.1 A Solution Based on Prefix Sums

In the rest of this section we describe our new cache-oblivious approach for computing all scale instances of a raster R. To describe this new approach, we first present some concepts used by Arge *et al.* [9]. For any scale instance R_s of a raster R, Arge *et al.* observed that we can express the value of a cell $R_s[i,j]$ using the prefix sums of the cells of R as $R_s[i,j] = \frac{\text{sum}(i,j,s)}{s^2}$, where:

$$\begin{aligned}\text{sum}(i,j,s) &= \text{psum}(is,js) - \text{psum}(is,(j-1)s) \\ &-\text{psum}((i-1)s,js) + \text{psum}((i-1)s,(j-1)s). \end{aligned} \tag{4.1}$$

Hence, to compute R_s we only need to extract the prefix sums from all cells $R[i',j']$ of R such that *both* i' and j' are integer multiples of s. It is easy to compute all rasters R_s if R fits in the main memory; first we compute a matrix that has $\sqrt{N}\times\sqrt{N}$ entries, and which stores the prefix sums for all cells in R. Then we can compute the value of each cell of R_s in constant time using the equation above, with only four random accesses to the entries of this matrix. Since the total number of cells of all rasters R_s is $\Theta(N)$, this approach leads to an internal memory algorithm that runs in $\Theta(N)$ CPU operations. However, it is not straightforward how to compute the rasters R_s efficiently if R does not fit in the main memory. To solve this problem we provide the following definitions.

Let P_1 denote the 2-dimensional matrix of $\sqrt{N} \times \sqrt{N}$ entries, such that for every entry $P_1[i,j]$ of this matrix we have that $P_1[i,j] = \text{psum}(i,j)$. For any $s \in \{2,3,\ldots,\sqrt{N}\}$, let P_s be the matrix that has $\lceil\sqrt{N}/s\rceil \times \lceil\sqrt{N}/s\rceil$ entries, where $P_s[i,j] = P_1[is,js]$. Thus, P_s stores all the prefix sums that are needed for constructing R_s; the value of each cell $R_s[i,j] = \frac{\text{sum}_s(i,j)}{s^2}$, where:

$$\begin{aligned}\text{sum}_s(i,j) &= P_s[i,j] - P_s[i,j-1] \\ &-P_s[i-1,j] + P_s[i-1,j-1].\end{aligned}$$

Therefore, assume that we already had an efficient algorithm for computing all matrices P_s. Then, we can extract from these matrices all scale instances R_s I/O-efficiently, in only $\mathcal{O}(\text{scan}(N))$ I/Os and $\Theta(N)$ CPU operations by simply scanning each matrix P_s, and maintaining four pointers to access the prefix sums needed for computing each value $R_s[i,j]$.

Hence, we now focus on designing an efficient algorithm for computing matrices P_s for every $s \in \{2,3,\ldots,\sqrt{N}\}$. It is easy to compute P_1; we can do this by scanning R, starting from $R[1,1]$ and visiting all cells in increasing order of their row and column indices and using equation (4.1) above. To compute a matrix P_s with $s > 1$, we could scan P_1 and extract each entry $P_1[i,j]$ such that both i and j are multiples of s. However, in this manner we spend $\mathcal{O}(\text{scan}(N))$ I/Os to extract each matrix P_s, leading to $\mathcal{O}(\sqrt{N}\cdot\text{scan}(N))$ I/Os for extracting all of these matrices.

To speed up the computation of the matrices P_s, we can exploit the following property; consider two distinct integers t and u such that $t,u \in \{2,3,\ldots,\sqrt{N}\}$, and $t = xu$, for some $x \in \mathbb{N}$, $x > 1$. Then it holds that $P_t[i,j] = P_u[ix,jx]$ for every entry $P_t[i,j]$ of matrix P_t. In other words, the entries of matrix P_t are a subset of the entries of P_u if t is divisible by u. Thus, we can construct P_t by processing a matrix P_u that can be much smaller than P_1. To construct P_t faster, we want to use the smallest matrix P_u for which t is a multiple of u; we must find the largest $u < t$ which is a divisor of t. We call this number the *largest distinct divisor* of t, and

we denote it by $\mathrm{ldd}(t)$. Note that if t is not a prime, then $t/\mathrm{ldd}(t)$ is the smallest prime factor of t. Consider two matrices P_t and P_u such that $t, u \in \{1, 2, \ldots, \sqrt{N}\}$, and $u = \mathrm{ldd}(t)$. We say that matrix P_t *derives* from matrix P_u, and that P_t is a *derived matrix* of P_u. In a similar manner, we say that scale instance R_t derives from instance R_u. For a matrix P_s we denote the set of matrices that derive from P_s by D_s, that is:

$$D_s = \{P_t \mid t \in \{2, 3, \ldots, \sqrt{N}\} \text{ and } s = \mathrm{ldd}(t)\} .$$

To compute matrices P_s, we first scan R to construct matrix P_1 that stores all prefix sums. Then, we extract all matrices D_1 that derives from P_1; these are the matrices P_s such that s is a prime $\leq \sqrt{N}$. To do this, we use a function `ExtractDerived`(P_s); the input of this function is a prefix sum matrix P_s, and the output is the set of the matrices that derive from P_s. We describe later in more detail how this function works. After constructing matrices $P_s \in D_1$, we apply again function `ExtractDerived` on these matrices to extract all sets of matrices D_s. We continue this process recursively, until we have computed all matrices P_s for the values $s \in \{2, 3, \ldots, \sqrt{N}\}$. We call the algorithm that we just described for computing all the scale instances of R as `MultirasterSpeedUp`.

It is easy to prove that `MultirasterSpeedUp` computes the scale instances of R correctly, assuming that function `ExtractDerived`(P_s) computes correctly the derived matrices of any given P_s. By Lemma 4.1, excluding the performance figures of `ExtractDerived`, the rest of the algorithm requires only $\mathcal{O}(\mathrm{scan}(N))$ I/Os and $\Theta(N)$ CPU operations. Next we show how we design function `ExtractDerived`.

4.2.2 Extracting the Derived Matrices

To compute the matrices D_s that derive from a given matrix P_s, we first have to compute all scale values t such that P_{ts} is a matrix that derives from P_s. We call these values the *derived indices* of s. We denote the set of these values by S_s. We observe that:

$$S_s = \{t \mid t \in \{2, 3, \ldots, \lfloor \sqrt{N}/s \rfloor\} \text{ and } s = \mathrm{ldd}(ts)\} .$$

Given s, we can calculate all derived scales S_s using the following observations;

Let s, t be two natural numbers such that $s = \mathrm{ldd}(ts)$, that is ts derives from s. First t is a prime since s is the largest distinct divisor in ts. Secondly $t \leq \mathrm{spd}(s)$, where $\mathrm{spd}(s)$ is the *smallest prime divisor* of s. This is true since otherwise $ts/\mathrm{spd}(s)$ would be a divisor of ts larger then s.

Based on the above, to compute S_s we first compute $\mathrm{spd}(s)$; we go through all integers $k \in \{2, \ldots, \lfloor \sqrt{s} \rfloor\}$ in increasing order, and we stop when we find the first k that divides s. Next we compute all prime numbers in the range $[2, \mathrm{spd}(s)]$ by trivially trying all possible pairs of integers within this range, and checking if the largest of the two is divided by the smallest. For the special case $s = 1$ the smallest prime divisor is undefined, and we consider that S_s consists of all prime numbers smaller than $\sqrt{N}$. Thus, for $s > 1$ we can compute scale values S_s in $\mathcal{O}(s)$ CPU operations. We need at most $\mathcal{O}(\mathrm{scan}(s))$ I/Os to store these values. For $s = 1$ this process requires $\mathcal{O}(N)$ CPU operations and $\mathcal{O}(\mathrm{scan}(N))$ I/Os.

To extract the derived matrices D_s, we use P_s to construct an intermediate file F_s that contains altogether the entries of all matrices in D_s, and then process this file

to extract each derived matrix I/O-efficiently. More specifically, file F_s is organised as follows; for every prime $t \in S_s$, and for every entry $P_{ts}[i,j] \in P_{ts}$, F_s contains a record of the form: $\{it, jt, t, P_s[it, jt]\}$.

The two first fields of the record indicate which is the entry in P_s that has the same value as $P_{ts}[i,j]$. The third field indicates the scale of P_{ts}, and the last field carries the value $P_{ts}[i,j]$. Most importantly, the records in F_s appear in lexicographical order of their three first fields.

Thus, F_s stores a record for each entry of the matrices in D_s, including multiples. The number of records in F_s is $\mathcal{O}(|P_s|)$; the number of entries of P_s is $|R_s|$, and due to Lemma 4.1 the total number of cells of all the scale instances of a raster R_s cannot exceed $|R_s|$. To construct F_s, we create an individual file $F_{s,k}$ for each matrix $P_{ks} \in D_s$. File $F_{s,k}$ contains only records of the form $\{ik, jk, ks, \otimes\}$, where $\otimes$ is a symbolic "no-data" value. Then we merge all those files into F_s in a bottom-up manner; first we generate F_s by merging the two files $F_{s,k}$ and $F_{s,t}$ that correspond to the two smallest matrices P_{ts} and P_{ks} in D_s; that is t, k are the two largest values in S_s. We go on merging F_s each time with the smallest remaining file $F_{s,u}$, until all files are merged into F_s.

Next we fill in the prefix sum values at the last field of each record in F_s with a single simultaneous scan of F_s and P_s. To extract matrices D_s from F_s we scan F_s once per matrix in D_s. The matrices are extracted in order of decreasing size; in the first scan of F_s we extract the largest matrix $P_{ts} \in D_s$, and so on and so forth. To extract P_{ts}, we pick the records in F_s whose third field is equal to t. We then throw away these records from F_s, creating a new smaller instance of F_s. When F_s becomes empty we will have extracted all derived matrices in D_s. The correctness of the algorithm follows from how we handle the prefix sum values in the records of file F_s. Next we prove the efficiency of this algorithm.

Lemma 4.2. *Function* `ExtractDerived` *computes the set of matrices D_s that derive from P_s in $\mathcal{O}(\text{scan}(|P_s| + s))$ I/Os and $\mathcal{O}(|P_s| + s)$ CPU operations.*

Proof. We showed that for $s > 1$ computing the scales S_s takes $\mathcal{O}(\text{scan}(s))$ I/Os and $\mathcal{O}(s)$ CPU time. Recall that for the case $s = 1$, we can compute S_s in $\mathcal{O}(\text{scan}(N))$ I/Os and $\mathcal{O}(N)$ CPU operations. Now we prove that for any $s > 1$ we can construct all matrices D_s in $\mathcal{O}(\text{scan}(|P_s|))$ I/Os and $\mathcal{O}(|P_s|)$ CPU operations. To construct file F_s, we merge several smaller files $F_{s,t}$, one merge at a time. As soon as file $F_{s,t}$ gets merged with F_s the records of $F_{s,t}$ become a part of F_s; from this point and on, these records are scanned once each time we merge F_s with another file $F_{s,k}$. Hence, each record that initially belonged to file $F_{s,t}$ gets scanned as many times as the number of primes that are smaller or equal to t; this is because S_s contains all primes in the range $[2, \text{spd}(s)]$, and because we merge files $F_{s,k}$ in decreasing order of k. In the mathematical literature, the number of primes that are smaller or equal to t is denoted by $\pi(t)$. As each record of $F_{s,t}$ is scanned $\pi(t)$ times, and as $F_{s,t}$ has $|P_{st}|$ records, the total number of records scanned when constructing F_s is:

$$\sum_{t \in S_s} \pi(t) \cdot |P_{st}| = \frac{N}{s^2} \sum_{t \in S_s} \frac{\pi(t)}{t^2} \ . \tag{4.2}$$

The following upper bound is known for $\pi(t)$ [64]: $\pi(t) < 1.26 \frac{t}{\ln t}$. Combining this with (4.2) we get:

$$\frac{N}{s^2}\sum_{t\in S_s}\frac{\pi(t)}{t^2} < 1.26\frac{N}{s^2}\sum_{t\in S_s}\frac{1}{t\ln t} = \frac{1.26}{\log e}\frac{N}{s^2}\sum_{t\in S_s}\frac{1}{t\log t}\ , \tag{4.3}$$

where e is the base of the natural logarithm. We have that:

$$\sum_{t\in S_s}\frac{1}{t\log t} < \sum_{i=0}^{\infty}\sum_{\substack{t \text{ is prime}\\ 2^{2^i}\le t<2^{2^{i+1}}}}\frac{1}{t\log t} \le \sum_{i=0}^{\infty}\sum_{\substack{t \text{ is prime}\\ 2^{2^i}\le t<2^{2^{i+1}}}}\frac{1}{2^i t}\ . \tag{4.4}$$

From the mathematical literature we know that [64] (Equation 2.30):

$$\sum_{\substack{t \text{ is prime}\\ t\le x}}\frac{1}{t} = \lg\lg x + \mathcal{O}(1)\ . \tag{4.5}$$

Applying this on (4.4) we get:

$$\sum_{i=0}^{\infty}\sum_{\substack{t \text{ is prime}\\ 2^{2^i}\le t<2^{2^{i+1}}}}\frac{1}{2^i t} = \mathcal{O}\left(\sum_{i=0}^{\infty}\frac{i+1}{2^i}\right) = \mathcal{O}(1)\ . \tag{4.6}$$

Combining (4.3) and (4.6) we get that the total number of records that we need to scan in order to construct F_s is $\mathcal{O}(|P_s|)$. This requires $\mathcal{O}(\text{scan}(|P_s|))$ I/Os. During the merging we do one comparison for every record that we scan, which implies that we do $\mathcal{O}(|P_s|)$ operations in the CPU in total.

It remains now to show that extracting all matrices of D_s from F_s requires $\mathcal{O}(\text{scan}(|P_s|))$ I/Os and $\mathcal{O}(|P_s|)$ time in the CPU. Recall that we extract the matrices P_t in increasing order of t, hence, the records of P_{st} will get scanned as many as $\pi(t)$ times each. Therefore the records scanned in this part of the algorithm are as many as the records scanned for constructing F_s. We showed that this number is equal to $\mathcal{O}(|P_s|)$, implying $\mathcal{O}(\text{scan}(|P_s|))$ I/Os and $\mathcal{O}(|P_s|)$ CPU operations for extracting the matrices for F_s, and the lemma follows. □

By construction, our algorithm does not require knowledge of M and B, hence it is cache-oblivious. Also, its performance does not depend on a lower bound on the size of M. We obtain the following theorem.

Theorem 4.3. *Given a raster R of $\sqrt{N}\times\sqrt{N}$ cells, we can compute all scale instances of R cache-obliviously in $\mathcal{O}(\text{scan}(N))$ I/Os and $\mathcal{O}(N)$ CPU operations.*

Proof. Function `ExtractDerived` is called only once for each matrix P_s so, according to Lemma 4.2, the total number of I/Os and CPU operations required by the entire algorithm is $\mathcal{O}(\text{scan}(\sum_s(|P_s|+s)))$ and $\mathcal{O}(\sum_s(|P_s|+s))$ respectively. Since P_s has the same size as R_s, then according to Lemma 4.1 and because $\sum_s|P_s| = \Theta(N)$, the theorem follows. □

4.2.3 Ordering the Prefix Sum Matrices

So far, we have described an algorithm that computes efficiently all scale instances of a given raster R. However, this algorithm does not output the scale instances of R in the right order. More specifically, from the description of algorithm `MultirasterSpeedUp` we can see that there can be pairs of scale instances R_s and R_t with $s < t$ such that R_t appears in the output before R_s. Yet, for most practical applications, it makes sense to have those instances sorted in the output in order of increasing scale value. Fortunately, we can solve this problem while achieving the same performance as with the algorithm `MultirasterSpeedUp`.

Theorem 4.4. *Given a raster R of $\sqrt{N}\times\sqrt{N}$ cells, we can compute cache-obliviously all scale instances of R, and output these instances in order of increasing scale using $\mathcal{O}(\text{scan}(N))$ I/Os and $\mathcal{O}(N)$ CPU operations.*

Proof. In hart our algorithm is recursive, first we produce P_p for all primes, and we recurse on each of these to further subdivide them. To get the output in the right order we will assume that the matrices from a recursion are returned in order, and make sure what we merge the result matrices to maintain the invariant. In order for the merging to be efficient we will recurse on the children in order of increasing size. Since we are only merging three streams at the same time, arguing for $\mathcal{O}(N)$ CPU operations implies the cache-oblivious bound of $\mathcal{O}(\text{scan}(N))$.

Lets consider the merging steps in the recursion of P_s. The size of the resulting matrix stream returned by the recursion on a given derived matrix P_{st} will be of size at most $0.65|P_{st}|$ by Lemma 4.1. We first merge this with P_{st} it self to create a matrix stream of size at most $1.65|P_{st}|$. This matrix stream is then merged with the resulting stream of the recursion on larger streams in turn. In total it will be merged with $\pi(t)$ other streams. So the total amount of work is:

$$\sum_{t\in S_s} \pi(t)1.65|P_{st}| \ .$$

By the same argument as in the proof for Lemma 4.2 this will sum to $\mathcal{O}(|P_s|)$. Since we at most use $\mathcal{O}(|P_s|)$ to recurse on every P_s, and since their total size is $\mathcal{O}(N)$ by Lemma 4.1 we use at most $\mathcal{O}(N)$ CPU operations on merging the resulting matrices by order. As stated above this also implies the $\mathcal{O}(\text{scan}(N))$ cache-oblivious I/Os. □

4.2.4 Improving the practical performance of the algorithm

Earlier in this section, we described how we can extract the prefix sum matrices D_s from a matrix P_s by building an intermediate file F_s. This approach requires merging several smaller files, and needs only $\mathcal{O}(\text{scan}(|P_s|))$ I/Os. Yet we can avoid this merging process, and thus improve the I/O-performance of the algorithm by a constant factor; to build F_s, we scan P_s and stream the records that correspond to the entries of matrices in D_s in the form of queries to P_s. After extracting the prefix sum value of a queried record, we append this record in F_s.

To do this, the records are streamed to P_s in lexicographical order of their three first fields. To produce the stream of the ordered records we build a min-heap structure. Each leaf node $v[t]$ of the heap corresponds to a derived matrix $P_{st} \in D_s$ and stores the next record of P_{st} that has to be streamed. The root of the heap stores the next record to be queried to P_s. Figure 4.1 illustrates the structure of

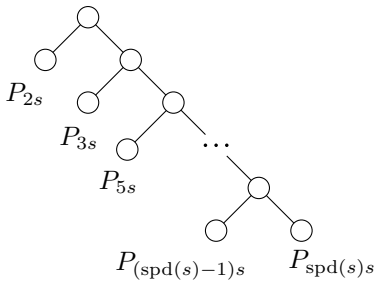


Figure 4.1: The structure of the skewed heap that we use to stream the records.

the heap; we can see that the heap is as skewed as it can get in favour of the larger derived matrices. The heap contains one leaf node for each derived matrix of P_s, so the size of the heap is $\mathcal{O}(\mathrm{spd}(s))$. Although we do not know M, we can build the heap so that at any point the nodes of the $\mathcal{O}(M)$ topmost levels appear in memory. For the rest of the levels, a record will have to pay one I/O for every B levels that it goes up in the heap. Although this method is oblivious of M, we show that we can stream all records to P_s so that the number of I/Os decreases as M increases.

Lemma 4.5. *Let P_s be a prefix sum matrix. We can stream all the records that correspond to the entries of the derived matrices of D_s in lexicographical order in $\mathcal{O}(\mathrm{scan}(|P_s|/\log M))$ I/Os and $\mathcal{O}(|P_s|)$ CPU operations.*

Proof. We lay the merge heap out linearly in memory such that root and the current location in P_{2s} comes before the right child of the root and P_{3s} and so on. That is we will store the tree in the order of a pre-order traversal.

We will analyse the paging strategy that maintains the top aM elements in memory for some $a > 0$. We note that producing all the entries for the top aM matrices incur no cache misses. While the remainder incur $\frac{1}{B}$ for every level they have to traverse up the tree until they reach height aM.

So the total number of cache faults must be a B'th of:

$$
\begin{aligned}
&\sum_{\substack{t\in S_s\\ \pi(t)>aM}} (\pi(t)-aM)|P_{st}| \\
&\leq \sum_{\substack{t\in S_s\\ t>aM}} \pi(t)|P_{st}| = \frac{N}{s^2}\sum_{\substack{t\in S_s\\ t>aM}} \frac{\pi(t)}{t^2} \\
&\leq \frac{1.26}{\log e}\frac{N}{s^2}\sum_{\substack{t\in S_s\\ t>aM}} \frac{1}{t\log t} \\
&\leq \frac{1.26}{\log e}\frac{N}{s^2}\sum_{\substack{t \text{ is prime}\\ t>aM}} \frac{1}{t\log t} \\
&\leq \frac{1.26}{\log e}\frac{N}{s^2}\sum_{i=\lfloor \log\log aM\rfloor}^{\infty}\ \sum_{\substack{t \text{ is prime}\\ 2^{2^i}<t\leq 2^{2^{i+1}}}} \frac{1}{t\log t}
\end{aligned}
$$

$$
\begin{aligned}
&\leq \frac{1.26}{\log e}\frac{N}{s^2}\sum_{i=\lfloor \log\log aM\rfloor}^{\infty}\sum_{\substack{t \text{ is prime}\\ 2^{2^i}<t\leq 2^{2^{i+1}}}}\frac{1}{t2^i}\\
&\leq \frac{1.26}{\log e}\frac{N}{s^2}\sum_{i=\lfloor \log\log aM\rfloor}^{\infty}\frac{1}{2^i}\sum_{\substack{t \text{ is prime}\\ 2^{2^i}<t\leq 2^{2^{i+1}}}}\frac{1}{t}\\
&\leq \frac{1.26}{\log e}\frac{N}{s^2}\sum_{i=\lfloor \log\log aM\rfloor}^{\infty}\frac{1}{2^i}\left(\sum_{\substack{t \text{ is prime}\\ t\leq 2^{2^{i+1}}}}\frac{1}{t}-\sum_{\substack{t \text{ is prime}\\ t\leq 2^{2^i}}}\frac{1}{t}\right)\\
&\leq \frac{1.26}{\log e}\frac{N}{s^2}\sum_{i=\lfloor \log\log aM\rfloor}^{\infty}\frac{\mathcal{O}(1)}{2^i}\\
&= \mathcal{O}\left(\frac{N}{s^2}\sum_{i=\lfloor \log\log aM\rfloor}^{\infty}\frac{1}{2^i}\right)=\mathcal{O}\left(\frac{N}{s^2}\frac{1}{\log M}\right)=\mathcal{O}\left(\frac{|P_s|}{\log M}\right)
\end{aligned}
$$

The bound on the CPU work follows by doing the same analysis with constant M and B. □

4.3 Implementation and benchmarks

We implemented `MultirasterSpeedUp` and evaluated its efficiency on datasets of various sizes. In the experiments that we conducted, we tried several alternatives for implementing the most important routines of the algorithm, and we assessed the efficiency of the implementation for each of these alternatives. We also compared the performance of our implementation with an older implementation of the $\mathcal{O}(\text{sort}(N))$ algorithm of Arge *et al.* [9]. Recall that it is not currently possible to implement the $\mathcal{O}(\text{scan}(N))$ algorithm of Arge *et al.* due to restrictions in standard operating systems; this algorithm requires that B files are open simultaneously, and while B today is in the order of millions of units, standard operating systems allow for about a thousand files open at the same time.

To measure the performance of our algorithm we used massive raster datasets of many sizes. The datasets that we used originate from a massive raster that consists of roughly 26 billion cells, arranged in 146974 rows and 176121 columns. This raster models the terrain surface over the entire region of Denmark. Each cell of the raster represents a square region on the terrain that has dimension of 2 meters. The elevation of each cell is stored as a 4-byte floating point number, and the entire dataset is stored in a geotif file that has 97 gigabytes size. From this dataset, we constructed all scale instances R_s for $s \leq 146974$, and we used the largest of these instances as input for the algorithm; we did this to evaluate the performance of the algorithm for a large range of different input sizes.

As already mentioned, we tried different options for implementing the key routines of the algorithm. These are the routines that involve merging or extracting a sequence of files from/to another larger file. For those routines we evaluated how the performance of the algorithm is affected when trying to merge/extract several files simultaneously. The routines that we tweaked are the following:

- The part of `ExtractDerived` where, given a prefix sum matrix P_s, we merge several files to construct an intermediate file F_s which contains the records that correspond to all the entries of the derived matrices D_s.

- The part of `ExtractDerived` where we extract the derived matrices D_s from the intermediate F_s.

For the above routines we measured how the performance of the algorithm changes if we change the number of files that are merged or extracted together. For the first routine we use f_1 to denote the number of files that we merged simultaneously at each point for constructing F_s. For the second routine we use f_2 to denote the number of derived matrices that we extracted together each time that we performed a scan of F_s. In the previous description of the algorithm, we convey that the value of each of these two parameters is equal to two. We also implemented a version of the routine that constructs the intermediate file F_s based on the mechanism of the skewed heap described in Section 4.2.4. Recall that this method does not merge any files in order to construct F_s. All versions of our implementation work in a purely cache-oblivious manner.

The algorithms were implemented in C++ using the software library `TPIE` (the *Templated Portable I/O Environment*) [1]. This library offers I/O-efficient algorithms for scanning and sorting large files in external memory. Our experiments where run on a machine with a 3.2GHz four-core Xeon CPU (W3565). The main memory of the computer is 12GB. This workstation has 20 disks that have a btrfs (raid 0) file system configuration. The operating system on this computer was Linux version 2.6.38. During our experiments, 8GB of memory was managed by our software, and the rest was left to the operating system for disk cache. For each of the versions of our implementation, the maximum amount of disk space used at any time during the execution was 672 GB.

In our first experiment, we ran our implementation of the algorithm on the 97GB dataset for all possible combinations of values of the two parameters $f_1, f_2 \in \{2, 3, 10, 20, 35, 50\}$. We also ran the implementation of the algorithm using the skewed heap approach for all values of parameter $f_2 \in \{2, 3, 10, 20, 40, 50\}$. From all the possible versions that we tried, the best running time was achieved by the version that uses the skewed heap approach, and parameter value $f_2 = 50$; the running time in this case was 2 hours and 15 minutes. The best running time that we got without using the skewed heap approach was for the version with parameter values $f_1 = f_2 = 50$. In this case, the running time was 2 hours and 28 minutes. The worst running time that we got among all versions was from the version that has parameter values $f_1 = 2$, and $f_2 = 2$; the running time for this version was 3 hours and 35 minutes. In general, the running time of each version that behaved like a decreasing function on the values of parameters f_1 and f_2. We also did experiments with values of f_1 and f_2 larger then 50, however here we did not see an additional improvement in runtime. Running the implementation of the $\mathcal{O}(\text{sort}(N))$ algorithm of Arge *et al.* on the largest dataset yielded a running time of 13 hours and 14 minutes. This running time is a bit less than four times larger than the worst running time that we got for any version of our implementation.

For our next experiment, we ran the two best versions of our implementation on the datasets that we got from extracting the 100 largest scale instances of the 97GB raster, including the initial raster itself. We also ran on these datasets the implemen-

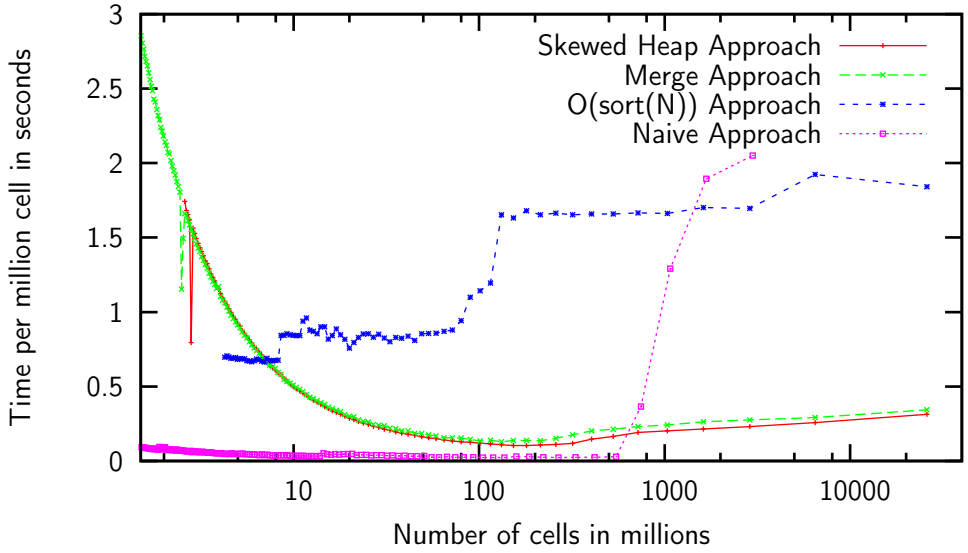


Figure 4.2: The performance of the two best versions of our implementation, together with the implementation of the $\mathcal{O}(\text{sort}(N))$ algorithm of Arge *et al.* , and the naive internal memory algorithm. The x-axis shows the input sizes using a logarithmic scale with base 10. The y-axis shows running times divided by input size.

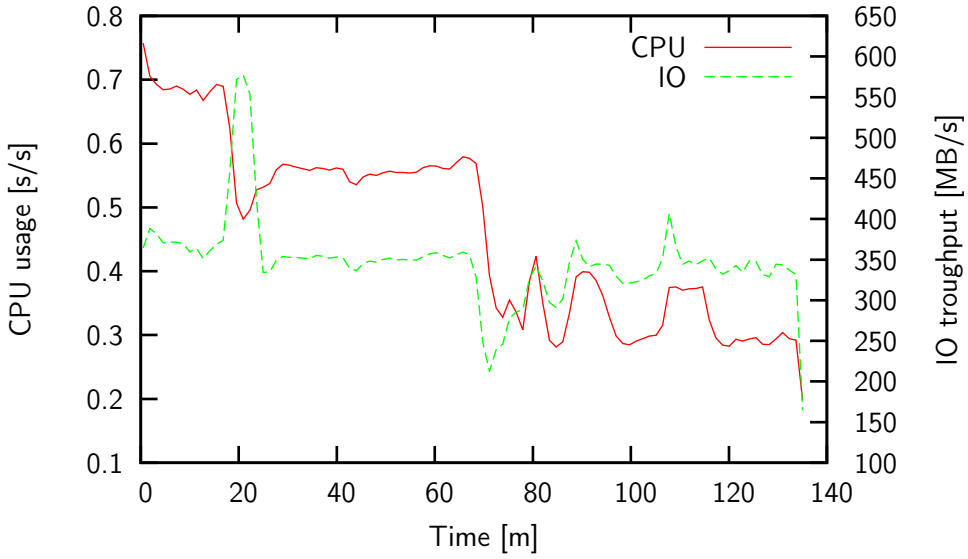


Figure 4.3: The CPU utilisation and I/O-throughput of the best version of our implementation.

tation of the $\mathcal{O}(\text{sort}(N))$ algorithm of Arge *et al.*, and the naive internal-memory algorithm that uses prefix sums. Figure 4.2 illustrates the performance of the four implementations. There, we get a good impression on how the performance of our implementation scales with the size of the input. This is a strong indication that the theoretical bounds that we proved for the performance of the algorithm can be reflected in practice.

The results of both experiments show evidently the practical efficiency of our algorithm, when also compared to the implementation of the algorithm of Arge *et al.*.

Of course, it could be argued here that this result is hardly surprising; in theory, an $\mathcal{O}(\text{sort}(N))$ algorithm has obviously worse asymptotical behaviour than an $\mathcal{O}(\text{scan}(N))$ algorithm. However, in practice, the performance of an $\mathcal{O}(\text{sort}(N))$ algorithm scales linearly in terms of I/Os. Figure 4.2 provides some evidence on this argument for the algorithm of Arge *et al.*, at least for the range of input sizes that we considered. The explanation behind this phenomenon is that the ratio M/B in most computers has a value close to one thousand, and therefore the term $\log_{M/B}(N/B)$ in $\text{sort}(N)$ is not larger than two in all practical cases. Thus, it is not unrealistic to observe $\mathcal{O}(\text{sort}(N))$ algorithms performing better in practice than $\mathcal{O}(\text{scan}(N))$ algorithms. More than that, in our case, we compare a cache-aware implementation with a cache-oblivious one, and we could expect that this is an advantage for the performance of the cache-aware implementation. Yet, as we see from our experiments, this is clearly not the case; the cache-oblivious algorithm performs much better in practice. This result shows that purely cache-oblivious software can be developed to perform efficiently in real-world applications. It is interesting to see if we can get similar results also for other external memory problems.

In our last experiment, we ran the best version of our implementation on the largest of our datasets, and at every minute of the execution we measured the rate of the CPU utilisation and the I/O-throughput of this implementation. Figure 4.3 illustrates the results of this experiment. We see that both the I/O-throughput and CPU utilisation were fairly constant during the run. Also, for the largest part of the execution of the algorithm, the CPU utilisation remained above or close to 40%; hence, the running time of the algorithm was almost equally distributed between the CPU and the I/O-operations.

5

Decomposition Simplification

This chapter contains [11] which is joint work with Lars Arge and Jungwoo Yang. The paper is included here with some proofs extended, and a new section detailing the labelling algorithm.

5.1 Introduction

Spatial data analysis has received considerable attention over the last few decades. Spatial data can be analysed in various ways, but visualizing the data is obviously a first simple method to understand the data. For visualization, spatial data is often represented by geometric primitives such as points, lines, and polygons, and often such representations form a *planar subdivision*, that is, an embedding of a planar graph with straight line edges. For example, (2.5-dimensional) terrain data is often visualized by (2-dimensional) contour maps, just like various terrain analysis results, such as drainage divisions, take the form of planar subdivisions. One reason for the increasing focus on spatial data analysis is that massive amounts of spatial data is increasingly available. For example, advances in mapping technology such as Light Detection and Ranging (LIDAR) technology has made high-resolution terrain data available. However, with the increasing size of the data also comes the need for intelligent data simplification. Such simplification is e.g. motivated by advanced analysis tools only being able to handle a limited amount of data.

In this chapter, we consider the planar subdivision simplification problem, where we are given a planar subdivision $\mathcal{P}$ and a constant $\varepsilon_{xy} > 0$ and want to construct a simpler planar subdivision $\mathcal{P}'$ such that no point on $\mathcal{P}'$ is moved more than a distance of ε_{xy} away from a point on $\mathcal{P}$ (*xy-constraint*) and such that $\mathcal{P}$ and $\mathcal{P}'$ are homotopic (*homotopy-constraint*). Intuitively, the homotopy-constraint means that $\mathcal{P}$ can be transformed to $\mathcal{P}'$ by smoothly moving edges and points, such that faces and neighbor relations between faces are preserved; Refer to Section 5.2 for more precise definitions of the *xy*- and homotopy-constraints.

We are interested in practically efficient algorithms for massive subdivisions that are too large to fit in main memory and must reside in disk. In such cases the movement of data between main memory and disk is often the bottleneck in a computation. We will therefore consider algorithms in the I/O-model of computation [5]. In this model the machine consists of a main memory that can hold M data elements and a disk of unbounded size. A block of B consecutive elements can be transferred between disk and main memory in a single I/O. Computation takes place in main memory, and the complexity of an algorithm is measured in terms of the number of I/Os it performs. Furthermore, since we are interested in practical applications of

algorithms for planar subdivision simplification, we will make two practically realistic assumptions (also made in previous similar work) namely that 1) all edges in $\mathcal{P}$ crossing any horizontal line fit in main memory, and 2) all edges in $\mathcal{P}$ incident to two faces sharing an edge fit in main memory.

Previous results. A large number of I/O-efficient algorithms have been developed in the last two decades. See recent surveys for an overview [6, 66]. Here we mention that scanning and sorting N elements takes $\mathcal{O}(\text{scan}(N)) = \mathcal{O}(N/B)$ and $\mathcal{O}(\text{sort}(N)) = \mathcal{O}(N/B \log_{M/B}(N/B))$ I/Os, respectively. We are not aware of any direct previous work on I/O-efficient algorithms for planar subdivision simplification. However, there has been a lot of work on the related problem of terrain simplification; Refer e.g. to [38, 25] and the references therein. Unfortunately, most often the developed approaches do not provide guarantees on accuracy (fulfill our constraints) or they are not I/O-efficient. Similarly, there is a lot of previous work on simplifying a polygonal line in the plane; Refer e.g. to [41] for a survey. However, trying to simplify a planar subdivision by simplifying individual polygonal lines in the subdivision will likely lead to intersections between the simplified polygonal lines and thus to violation of the homotopy-constraint.

Very recently, however, an I/O- and practically-efficient (but not particularly simple) algorithm for the very related contour map simplification problem was developed by Arge et al. [8]. The problem is defined as follows. Let $\mathcal{T}$ be a terrain represented as a planar triangulation with heights associated with the nodes (also known as a triangulated irregular network or TIN); the height of an arbitrary point p is obtained by linear interpolation between the three nodes of the triangle of $\mathcal{T}$ containing the xy-projection of p. The h-level set of $\mathcal{T}$ is the set of (planar) edges obtained by intersecting $\mathcal{T}$ with a horizontal plane at height h. A contour is a connected component of a level set, and a contour map $\mathcal{H}$ is the union of multiple level sets. $\mathcal{H}$ is a planar subdivision and if we for simplicity assume that none of the level sets are defined by heights of the nodes of $\mathcal{T}$ then all contours in $\mathcal{H}$ are cycles. The contour map simplification problem is simply the planar subdivision problem on $\mathcal{H}$ with the added *z-constraint*, that for any point p on a contour in the h-level set of the simplified subdivision (map) $\mathcal{H}'$ the difference between h and the height of p in $\mathcal{T}$ is less than ε_z. Note that the problem reduces to normal planar subdivision simplification if the $(h - \varepsilon_z)$-level and $(h + \varepsilon_z)$-level set are added to $\mathcal{H}$ before the simplification. In fact, this is exactly how the problem is solved in the recent $\mathcal{O}(\text{sort}(N))$ I/O algorithm by Arge *et al.* [8]. The algorithm first I/O-efficiently constructs a so-called topology tree that encodes how the contours in $\mathcal{H}$ are nested (that is, their inside/outside relationships). Then this tree is traversed and the contour c in each node v is loaded into main memory in turn, along with the contours in the parent, siblings and children of v. Note that these *constraint* contours are exactly the contours that are adjacent to one of the two faces adjacent to c; Refer to Figure 5.1. Then c is simplified under the xy- and homotopy-constraints relative to the constraint contours with an internal memory algorithm, which is a variant of the Douglas-Peucker polygonal line segment simplification algorithm [63, 29]. Finally, the simplified contour c' is reinserted in the topology tree (to ensure that no later simplified contour intersect c'). Arge *et al.* [8] performed experiments on massive contour maps and showed that the algorithm performs well in practice. Note, however, that the algorithm cannot easily be adapted to work on planar subdivisions,

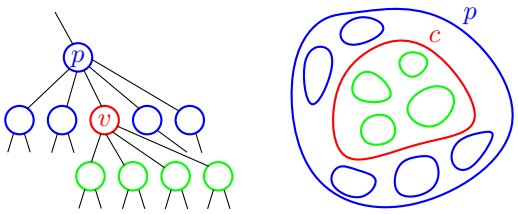


Figure 5.1: Illustration of the Arge *et al.* [8] algorithm. Contour c in a node v of the topology-tree is simplified relative to the contours in children (green) and parent and siblings (blue).

since the topology tree (nesting) is only well-defined for contour maps (and not for general subdivisions).

Our results. In this chapter, we present the first I/O- and practically-efficient algorithm for planar subdivision simplification. Our algorithm is not only more general than the contour map simplification algorithm in [8] but also simpler. To illustrate this, consider how our algorithm works in the case where the subdivision is a contour map $\mathcal{H}$. In this case the algorithm first considers each contour c in turn and assigns it to the largest of the two faces adjacent to c (the face with most adjacent contour edges), along with all other contours adjacent to the other (smallest) face as *constraint contours*. Note that the constraint contours assigned to a face with c are either the contours in the children of the node v in the topology tree for $\mathcal{H}$ containing c or the contours in v's parent and siblings; Refer to Figure 5.1. Note also that a given contour can be assigned to many faces. Next the algorithm considers each face f in $\mathcal{H}$ in turn and simplifies each contour c assigned to f in internal memory under the xy- and homotopy-constraints relative to the constraint contours assigned with c and all contours (except for c itself) adjacent to f in $\mathcal{H}$. Note that this means that just as in [8], c is simplified relative to contours in the parent, siblings and children nodes of the node v in $\mathcal{H}$.

We present our algorithm in two sections below: The external part of the algorithm is described in Section 5.3 and the internal memory part in Section 5.5. Our algorithm is simpler (and more general) than the previous algorithm [8] since it completely avoids the topology tree. This is accomplished by collecting constraint contours (by assigning contours to faces) before the actual simplification, which makes the construction and traversal, as well as update, of the topology tree unnecessary. However, since only original contours in $\mathcal{H}$ are considered as constraints during the simplification, the correctness (fulfillment of the homotopy-constraint) is not straightforward since intersections between adjacent contours could potentially be introduced when they are simplified separately. Also the efficiency of the algorithm is not straightforward, since (as mentioned) a contour can be assigned as a constraint to many faces. In Section 5.4 we provide a correctness proof and show that our algorithm is as efficient as the previous contour map simplification algorithm [8].

We have implemented our algorithm and in Section 5.6 we present the results of experimenting with it on real-life data, more precisely on data derived from a detailed terrain model of Denmark containing over 12 billion data elements. We both compare its performance to the previous algorithm for contour map simplification [8], and investigate its performance on general planar subdivisions by using it to simplify

a so-called catchment decomposition of a terrain. For the contour map simplification problem our algorithm is significantly faster than the previous algorithm, while obtaining approximately the same simplification factor w.r.t. to the number of edges . For the catchment decomposition simplification problem our algorithm also performs well. Our experiments reveal that the simplification factor depends significantly on the length of the boundary between adjacent faces, with the largest simplification being obtained when adjacent faces share many edges (as is the case in contour map simplification). Overall, our experiments confirm that our algorithm is not only simple and general but also practically efficient.

5.2 Preliminaries

A *path* P is a set of edges defined by pairs of consecutive points in a sequence $p_1, \ldots, p_n$ of $n > 1$ points in $\mathbb{R}^2$. Abusing notation slightly, we use P to denote both the sequence of points and the path itself. A *sub-path* of P is defined by a consecutive subsequence $p_i, p_{i+1}, \ldots, p_j, 1 \leq i < j \leq n$ of P, and a *simplification* P' of P is simply a subsequence of P. A *polygon* is a path P where $p_1 = p_n$. A point p is within a polygon P if any path from p to infinity crosses at least one edge of P. We will abuse notation slightly and also use a polygon P to define the closed region of space defined by points within or on the boundary of P.

Let $\mathcal{P}$ be a planar subdivision with N edges. An edge e is said to be a *face edge* of a face f in $\mathcal{P}$ if e is adjacent to f, and we use E_f to denote the set of all face edges of f. The *size* $|f|$ of a face f is defined to be $|E_f|$. A face is a *neighbor* of another face if they share a face edge. A *maximal path* in $\mathcal{P}$ is a connected path of edges of $\mathcal{P}$, such that each node except the first and last node has degree two, and such that the first and last have degree larger than two or are the same node, that is, a maximal path is a path that cannot be extended and still only contains nodes of degree two.

Definition 5.1. Let $\mathcal{S} \subseteq \mathbb{R}^2$, $\mathcal{Q}, \mathcal{Q}' \subseteq \mathcal{S}$. $\mathcal{Q}$ is *homotopic* to $\mathcal{Q}'$ in $\mathcal{S}$ if and only if there exists a continuous function $H : \mathcal{Q} \times [0,1] \to \mathcal{S}$ such that

1. $H(\cdot, 0)$ is the identity function
2. $\{H(p,1) \mid p \in \mathcal{Q}\} = \mathcal{Q}'$
3. $H(\cdot, t)$ is injective for all $t \in [0,1]$

We call H a *homotopy function* from $\mathcal{Q}$ to $\mathcal{Q}'$ in $\mathcal{S}$.

Let $\mathcal{Q}$ and $\mathcal{Q}'$ be the sets of points consisting of all points on the edges of planar subdivisions $\mathcal{P}$ and $\mathcal{P}'$, respectively. We say that $\mathcal{P}$ is homotopic to $\mathcal{P}'$ if and only if $\mathcal{Q}$ is homotopic to $\mathcal{Q}'$ in $\mathbb{R}^2$.

Intuitively, the homotopy function H in the above definition transforms the decomposition $\mathcal{Q}$ into $\mathcal{Q}'$ as "time" goes from 0 to 1, that is, $H(p,t)$ defines where a point p in $\mathcal{Q}$ is placed at time t in the homotopic transformation. The continuity of H ensures that paths are not broken, and the injective property ensures that a maximal path P in $\mathcal{P}$ is transformed into a maximal path P' in $\mathcal{P}'$, that faces in $\mathcal{P}$ maintain their neighbors in $\mathcal{P}'$, and thus that no faces appear or disappear.

Definition 5.2. Let $\mathcal{P}$ be a planar subdivision homotopic to a planar subdivision $\mathcal{P}'$. $\mathcal{P}$ and $\mathcal{P}'$ are within distance $\varepsilon_{xy} > 0$ if and only if for each maximal path P in $\mathcal{P}$ and the corresponding maximal path P' in $\mathcal{P}'$, each point on one of these paths is within distance ε_{xy} of a point on the other path.

Throughout the rest of this chapter, $\mathcal{P}$ will be the input planar subdivision and $\mathcal{P}'$ the simplified output subdivision. As discussed in the introduction, we want $\mathcal{P}$ to be homotopic to $\mathcal{P}'$ (Definition 5.1; fulfill the homotopy-constraint), and $\mathcal{P}$ and $\mathcal{P}'$ to be within distance ε_{xy} of each other (Definition 5.2; fulfill the xy-constraint).

The following lemmas about homotopy (needed in the correctness proof in Section 5.4) are easily proved. Intuitively the first lemma states that a homotopic relation in a space S will also hold in any larger space containing S. The second lemma states that two homotopic relations in disjoint spaces can be joined into one homotopic relation in the union of the two spaces.

Lemma 5.1. *Let $\mathcal{S} \subseteq \mathcal{S}' \subseteq \mathbb{R}^2$, $\mathcal{Q}, \mathcal{Q}' \subseteq \mathcal{S}$, such that $\mathcal{Q}$ is homotopic to $\mathcal{Q}'$ in $\mathcal{S}$. Then $\mathcal{Q}$ is homotopic to $\mathcal{Q}'$ in $\mathcal{S}'$.*

Lemma 5.2. *Let $\mathcal{S}_1, \mathcal{S}_2 \subseteq \mathbb{R}^2$. Let H_1 be a homotopy function from $\mathcal{Q}_1$ to $\mathcal{Q}'_1$ in $\mathcal{S}_1$ and H_2 be a homotopy function from $\mathcal{Q}_2$ to $\mathcal{Q}'_2$ in $\mathcal{S}_2$. If $H_1(p,t) = H_2(p,t) = p$ for all $p \in \mathcal{S}_1 \cap \mathcal{S}_2$, $t \in [0,1]$, then $\mathcal{Q}_1 \cup \mathcal{Q}_2$ is homotopic to $\mathcal{Q}'_1 \cup \mathcal{Q}'_2$ in $\mathcal{S}_1 \cup \mathcal{S}_2$.*

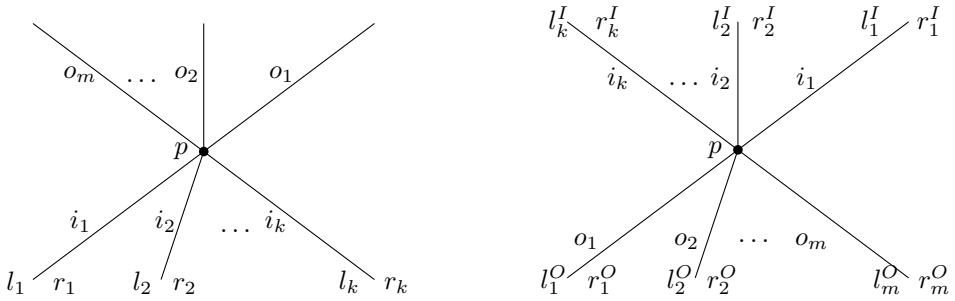
5.3 Algorithm

In this section we describe the external part of our algorithm for simplifying a planar subdivision $\mathcal{P}$. The internal memory part of the algorithm (an algorithm for simplifying a maximal path $\tilde{P}$ under the xy- and homotopy-constraints relative to a set of constraint edges) is described in Section 5.5.

As described for the special case of contour maps in the introduction, our algorithm works in two phases: In the first phase edges of $\mathcal{P}$ are assigned to faces, and in the second phase each face is considered in turn and the actual simplification is performed to obtain $\mathcal{P}'$. Both phases require that every edge e in $\mathcal{P}$ is labeled with the two faces adjacent to e. If this is not the case, we can obtain the labels in $\mathcal{O}(\text{sort}(N))$ I/Os using an algorithm similar to an algorithm due to Arge *et al.* [10] for computing (labeling) connected components in a planar embedded graph, as detailed below.

Labelling. Our labelling algorithm uses two sweepline phases: In the first phase we sweep up (in y direction) while maintaining connectivity below the sweepline, and in the second phase we sweep down while maintaining connectivity above the sweepline. During the second phase we also consider the connectivity information obtained during the first phase in order to compute overall connectivity.

The up sweep uses five different data structures. We have the external stream of outgoing segments O, which is sorted by their minimal y-coordinate, these segments are the input to the algorithm. We have an internal priority queue I with incoming segments and associated left and right labels, of the form (s, l, r). The segments in I are ordered by their maximal y-coordinate (smallest first). We have an ordered set L (red-black tree) of segments, with associated labels of the form (s, l, r). The segments are ordered from left to right (w.r.t. intersection with the sweepline). Both I and L will contain the segments currently intersected by the sweepline. We have a reference



(a) Up sweep (b) Down sweep

Figure 5.2: The node p in a up sweep (a) or a down sweep (b), we have k incoming segments with associated labels, and m outgoing segments. The segments are ordered counter clockwise around p

counted disjoint set structure U (union find) of labels. Any label in I or L is also present in U. The labels in U are reference counted in such a way that once labels go out of I and L they may also in time be removed from U. Specifically at most constant fraction of the elements in U will not be in I and L. We require that the representative of a set in U is its minimal element. Finally we have an external stream R of resulting segments with associated labels, on the form (s, l, r). R contains the result of the up sweep, the segments will by construction be ordered by their maximal y-coordinate (smallest first).

In the up sweep, the nodes of the segment graph (the endpoints of the line segments) are processed in increasing y order. For every node p the outgoing segments $o_1, \ldots, o_m$ are loaded from O, and the incoming segments $i_1, \ldots, i_k$ are loaded from I, together with the associated labels $l_1, \ldots, l_k$ and $r_1, \ldots, r_k$. As shown in Figure 5.2a the segments are ordered counter clockwise around p and the labels represent the left and right faces.

For every j in $1, \ldots, k$, we will find the representatives l'_j and r'_j of l_j and r_j respectively in U, and output (i_j, l'_j, r'_j) to R. For every j in $1, \ldots, k-1$, the labels r_j and l_{j+1} will be unioned in U. If $k = 0$, we will find the predecessor (lower bound) of p in L. This will be some segment on the form (s, l, r) that is to the left of p. The label of the face p is located in will be the representative r' of r in U, so we will set $n_0 := n_m := r'$, for later use. Otherwise if $k \neq 0$, but $m = 0$, we will union l_1 and r_k in U and get a new representative r', again we set $n_0 := n_m := r'$. In the final case where $k \neq 0$ and $m \neq 0$, we will set n_0 to be the representative of r_k in U and n_m to be the representative of l_1 in U. In any case we will create $m - 1$ new labels in U and call them n_1, n_{m-1} these will be the labels assigned to the new faces between each of the outgoing segments. For every j in $1 \ldots m$, we will insert the (o_j, n_{j-1}, n_j) into I and L.

After the up sweep, a down sweep is performed. In this we replay the unions of the up sweep in reverse order. The sweep uses four structures. We have the stream of outgoing segments R on the form (s, l, r), which is the output of the up sweep. We will read this in reverse order to get the elements ordered by their maximal y coordinate in decreasing order. We have an internal priority queue Q of incoming segments on the form (s, l, r), this will be ordered by the minimal y coordinate in decreasing order, and will only contain segments intersected by the sweepline. We have a reference counted dictionary D of labels, mapping temporary labels from the

up sweep, to the final labels. We will only store mappings of labels referenced in Q. Finally we have the output X which contains segments on the form (s, l, r) where l is the label of the face to the left of s and r is the label of the face to the right of s. By construction X is ordered by the minimal y coordinate of the segments in increasing order. That is the segments in X occur in the same order as in O.

In the down sweep, the nodes of the segment graph are processed in decreasing y order. For every node p the outgoing segments $(o_1, l_1^O, r_1^O) \dots (o_m, l_m^O, r_m^O)$ are loaded from R and the incoming segments $(i_1, l_1^I, r_1^I) \dots (i_k, l_k^I, r_k^I)$ are loaded from Q. As show in Figure 5.2b the segments are again ordered counter clockwise around p. The unions preformed for p in the up sweep are replayed. For given l, we define $D[l]$ to be the mapping of l in D or l if l is not mapped by D. For given l, r, the union of l and r in D is performed by setting $D[l] := D[r] := \min(D[l], D[r])$. For every j in $1 \dots m-1$ we union r_j^O and l_{j+1}^O in D. If $k = 0$ we also union l_0^O and r_m^O in D. For every j in $1 \dots k$ we output $(i_j, D[l_j^I], D[r_j^I])$ to the output stream X. For every j in $1 \dots m$ we add (o_i, l_j^O, r_j^O) to Q.

The memory used by the labelling algorithm is linear in the number of segments crossing the sweepline, assuming this also holds for the disjoint set structure. To construct a reference-counted disjoint set with this property, we start with a regular disjoint set structure with path compression. We augment each node with pointers to the left and right siblings and the left and right children, as well as the reference count. Whenever a note has a reference count of zero, has a parent and has less then two children, the node can be deleted and the at most one child can be moved to the parent. This way leafs in the tree are always alive, and the number of dead internal nodes can be bounded by the number of leafs, without affecting the asymptotic running time of any of the operations.

To see that the algorithm is correct we need to prove that any two labels assigned to a face in the output are identical. In the up sweep several labels can be assigned to a given face. New labels are assigned in the case where $m > 1$, here $m - 1$ labels are created. All these labels will at some point during the up sweep be unioned in the case where $k > 1$ or $m = 0$. We note that in the case where $k = 0$, we copy a label already used for the face, such that the labels on inner rings will be the same as that on the outer ring. Once we have swept across a face, all the different labels assigned to it will have been unioned, and every time a union has been performed the labels outputted to R immediately before the union bear witness to it such that it is replayed correctly in the down sweep. Since the down sweep visits the notes and therefore the unions in reverse order, and since the representatives of a union is always the minimal label, all the different labels assigned to a face in the up sweep, will be mapped to the minimal of these in the down sweep.

We note that if we create new labels in increasing order, and if we choose the representative of a union as the smallest of the two representatives, then a face starting below another face will have a smaller label then one starting above it. In particular any face completely contained within another will have a higher label then the one it which it is contained.

Phase one. In this phase, for each face f we collect all face edges (except the ones shared with f) from all smaller neighbor faces of f (faces f' with $|f'| < |f|$). Note that in the contour map case, this corresponds exactly to assigning each contour c to the largest face adjacent to c along with all other contours adjacent to the smaller

face adjacent to c (as described in the introduction).

To collect edges, we first make two copies of each edge in $\mathcal{P}$. More precisely, for an edge e between nodes (points) p_1 and p_2 and with adjacent faces l and r we output (l, r, p_1, p_2) and (r, l, p_1, p_2) to a list S. This can be accomplished in a simple scan of the edges. Next for each face f we compute the size of f and of each neighbor face f' of f. To do so, we first sort S in lexicographical order, such that all face edges of f occur consecutively in S and such that all shared edges with each neighbor f' also occur consecutively. Then we scan through S and for each face f we compute the size $|f|$ of f, as well as the size $|f'|$ of each neighbor face f' while outputting $(f', f, |f|)$ to a list S_n. Then we sort S_n lexicographically, such that it contains the size of all neighbor faces (more precisely, for each face it contains a number of consecutive elements containing the size of neighbor faces). Finally, we perform the actual collection of edges by scanning through S and S_n simultaneously. For each face f we first load all its face edges into memory from S; by assumption they fit in memory. We then obtain the size $|f'|$ of each neighbor from S_n, and if $|f| < |f'|$[1] we output all edges of f not shared with f' to a list S_f in the form (f', f, p_1, p_2). After handing all faces, we sort S_f in lexicographical order, such that for each face it contains all non-shared edges of all smaller neighbor faces as required.

Note that phase one is performed using a constant number of scans and sorts of S, S_n and S_f. S and S_n are of linear size in N, and we will bound the size of S_f in Section 5.4.

Phase two. In this phase we simplify each maximal path $\tilde{P}$ of $\mathcal{P}$ in turn (using the internal algorithm discussed in Section 5.5) with all the face edges of its two adjacent faces as constraints. Note that in the contour map case each contour will be a maximal path, and the edges of the two adjacent faces will be exactly the contours of the parent, siblings and children nodes in the topology tree (as described in the introduction).

To simplify each maximal path we scan through S and S_f simultaneously. For each face f, we first load all its face edges into memory from S, and then we in turn load all non-shared face edges of each smaller neighbor face f' into memory from S_f; by assumption they fit in memory. We simplify each maximal path $\tilde{P}$ shared between f and f' using the internal algorithm (Section 5.5) with constraints $(E_f \cup E_{f'}) \setminus \tilde{P}$ as required. We also simplify maximal paths incident only to f with $E_f \setminus \tilde{P}$ as constraints.

Note that each maximal path is simplified in the above process, since it will either be shared between two neighbor faces (and then simplified when the largest of the faces is considered) or be incident to only one face (and then simplified when that face is considered). Note also that phase two is performed in one scan over S and S_f.

[1] If $|f| = |f'|$ we break ties arbitrarily by outputting edges to the face with the largest label (that is, we output to f if $f > f'$).

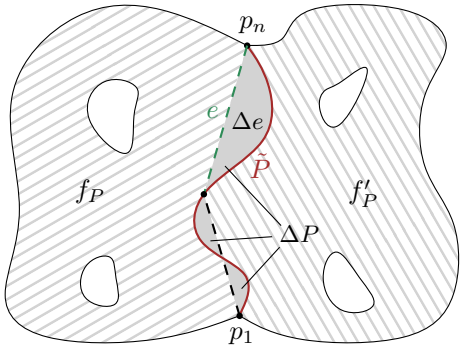


Figure 5.3: Illustration of the definitions used in correctness proof. The solid lines are in $E_{f_P} \cup E_{f'_P}$ and the dashed lines in P.

5.4 Analysis

In this section we show that our algorithm fulfills the homotopy- and xy-constraints, and that it is efficient.

Correctness. Given a maximal path $\tilde{P} = p_1, p_2, \ldots, p_{n-1}, p_n$ of $\mathcal{P}$ with adjacent faces f_p and f'_p (as shown in Figure 5.3), the internal algorithm (Section 5.5) computes a simplified maximal path $P = p_{s_1}, p_{s_2}, \ldots, p_{s_m}$ of $\mathcal{P}'$ where P is a subsequence of $\tilde{P}$, with $s_1 = 1 < s_2 < \cdots < s_m = n$, so that every simplified edge $e = (p_{s_i}, p_{s_{i+1}}) \in P$ replaces a sub-path $\tilde{e} = (p_{s_i}, \ldots, p_{s_{i+1}})$ in $\tilde{P}$. We define Δe as the polygon obtained by joining $\tilde{e}$ and e, and $\Delta P = \bigcup_{e \in P} \Delta e$. Finally we say that e satisfies the xy-constraint if and only if the distance[2] between $\tilde{e}$ and e is at most ε_{xy}. Note that the simplification satisfies the xy-constraint if all edges in $\mathcal{P}'$ satisfy the xy-constraint.

The simplified path P of $\tilde{P}$ computed by the internal algorithm has the following properties (refer to Section 5.5):

(1) $(E_{f_P} \cup E_{f'_P}) \setminus \tilde{P}$ is disjoint from ΔP except for the endpoints of P.

(2) ΔP is contained within $f_P \cup f'_P$.

(3) $\tilde{P}$ is homotopic to P in ΔP.

(4) Any edge in P satisfies the xy-constraint.

Lemma 5.3. *Let P and Q be distinct maximal paths in $\mathcal{P}'$. Then ΔP is disjoint from ΔQ except possibly for the shared endpoints of P and Q.*

Proof. Assume for contradiction that ΔP and ΔQ are not disjoint. Since $\Delta P = \bigcup_{e \in P} \Delta e$ there must exist edges $e \in P$ and $g \in Q$ such that Δe is not disjoint from Δg. By (2) we know that ΔP is contained within $f_P \cup f'_P$ and ΔQ is contained within $f_Q \cup f'_Q$, so $\tilde{e}$ and $\tilde{g}$ must be adjacent to a common face. Since Δe is not disjoint from Δg, either one is within the other or their boundaries must intersect.

[2]The distance between a point p and an edge e is the minimal Euclidean distance between p and a point on e. The distance between an edge e and a path $\tilde{e}$ is the maximal distance between e and any point on $\tilde{e}$.

If we assume that one is within the other, say Δe is within Δg, then also $\tilde{e}$ must be within Δg, but $\tilde{e} \subseteq (E_{f_Q} \cup E_{f'_Q}) \setminus \tilde{Q}$ since $\tilde{P}$ and $\tilde{Q}$ share a common face, thus one polygon cannot be contained within the other since it would violate (1).

The only remaining case is then that the boundary of Δe intersects the boundary of Δg. By (1) e does not intersect $\tilde{g}$, g does not intersect $\tilde{e}$, and by the definition of $\mathcal{P}$, $\tilde{e}$ and $\tilde{g}$ do not intersect (possibly all pairs can intersect on the shared endpoints of P and Q). Therefore, only e and g can intersect. Note that in order for Δe and Δg to share only a point, the shared point must be one of the shared endpoints of P and Q, that is, Δe and Δg must intersect at least twice. Since e and g are edges, they can intersect at most once, which is a contradiction to the fact that they must intersect at least twice. Thus, this contradicts the assumption that ΔP and ΔQ are not disjoint. □

Theorem 5.4. *$\mathcal{P}$ is homotopic to $\mathcal{P}'$ in $\mathbb{R}^2$ and satisfies the xy-constraint.*

Proof. Let $P_1, \ldots, P_k$ be the maximal paths of $\mathcal{P}'$. By (3) we have that $\tilde{P}_i$ is homotopic to P_i in ΔP_i. By Lemma 5.3 all polygons ΔP_i are disjoint from each other except for the shared endpoints, where the points are not moved during the transformation. By repeated applications of Lemma 5.2, we get that $\bigcup_i \tilde{P}_i = \mathcal{P}$ is homotopic to $\bigcup_i P_i = \mathcal{P}'$ in $\bigcup_i \Delta P_i$. By Lemma 5.1, since $\bigcup_i \Delta P_i \subseteq \mathbb{R}^2$ we get that $\mathcal{P}$ is homotopic to $\mathcal{P}'$ in $\mathbb{R}^2$.

By (4) all edges on all paths in $\mathcal{P}'$ satisfies the xy-constraint, so (as noted earlier) $\mathcal{P}'$ also satisfies the xy-constraint. □

Efficiency. The external part of our algorithm performs a constant number of scans and sorts on the lists S, S_n and S_f. Recall that S and S_n are of linear size in N, and that S_f is produced by for each face f of $\mathcal{P}$ collecting all face edges from all smaller neighbor faces of f. To show that our algorithm uses $\mathcal{O}(\text{sort}(N))$ I/Os, we need to show that S_f is of size $\mathcal{O}(N)$.

We consider the dual graph $\mathcal{P}_d$ of $\mathcal{P}$, that is, $\mathcal{P}_d$ contains a node for each face in $\mathcal{P}$, and two nodes in $\mathcal{P}_d$ are connected by an edge if and only if the corresponding faces in $\mathcal{P}$ are neighbors. For a node v in $\mathcal{P}_d$, we define the weight $w(v)$ as the size of the face corresponding to v in $\mathcal{P}$. We define the weight $w(e)$ of an edge e in $\mathcal{P}_d$ to be the minimum of the weights of its endpoints. Observe that $\sum_{v \in \mathcal{P}_d} w(v) = 2N$ and that $|S_f| = \sum_{e \in \mathcal{P}_d} w(e)$.

Lemma 5.5. *Let $F(V, E)$ be a forest such that each node $v \in V$ has non-negative weight $w(v)$. For every $e(u, v) \in E$ we define $w(e) = \min(w(u), w(v))$. Then $\sum_{e \in E} w(e) \leq \sum_{v \in V} w(v)$.*

Proof. We may assume $F(V, E)$ is a tree, since a forest can be transformed into a tree by adding edges between components, which increases $\sum_{e \in E} w(e)$. Now, F can be viewed as a rooted tree by choosing an arbitrary node r in V as the root. For every edge $e \in E$, we define $w'(e)$ to be the weight of the node of e furthest from r. We know $\sum_{e \in E} w'(e) \leq \sum_{v \in V} w(v)$ since the weight of every node is counted at most once. Since $w(e) \leq w'(e)$, we have $\sum_{e \in E} w(e) \leq \sum_{v \in V} w(v)$. □

Theorem 5.6. *The number of I/Os performed by our algorithm is $\mathcal{O}(\text{sort}(N))$.*

Proof. By Nash-Williams' formula [55, 56], every planar graph has *arboricity* at most three, where the arboricity of a graph is the minimum number of forests into

which the edges of the graph can be partitioned. Thus since $\mathcal{P}_d$ is a planar graph, the edges of $\mathcal{P}_d$ can be partitioned into at most three forests. Let $F_i(V_i, E_i)$ be a forest for $1 \leq i \leq 3$ such that $\bigcup_i F_i = \mathcal{P}_d$. By Lemma 5.5,

$$\sum_{e \in \mathcal{P}_d} w(e) \leq \sum_{1 \leq i \leq 3} \sum_{e \in E_i} w(e) \leq \sum_{1 \leq i \leq 3} \sum_{v \in V_i} w(v) \leq 6N \ .$$

Thus, we see that $|S_f| = \mathcal{O}(N)$ and our algorithm uses $\mathcal{O}(\text{sort}(N))$ I/Os. □

5.5 Internal simplification algorithm

Our internal algorithm for simplifying a maximal path is similar to the internal ring simplification algorithm of [8]. We are given a maximal path $\tilde{P} = p_1, \ldots, p_n$ of $\mathcal{P}$ and the set of edges E of the (at most) two faces f and f' adjacent to $\tilde{P}$ (that is, $E = (E_f \cup E_{f'}) \setminus \tilde{P}$). The algorithm computes a simplification P of $\tilde{P}$ satisfying the following properties:

(1) E is disjoint from ΔP except for the endpoints of P.

(2) ΔP is contained within $f \cup f'$.

(3) $\tilde{P}$ is homotopic to P in ΔP.

(4) Any edge in P satisfies the xy-constraint.

Algorithm. The simplification is performed by a simple recursive algorithm on a sub-path $\tilde{P}_{ij} = p_i, \ldots, p_j$, based on Douglas-Peucker's algorithm [29], where $\tilde{P}_{ij}$ initially is $\tilde{P}$ itself. Let $\Delta\tilde{P}_{ij}$ be the polygon obtained by closing $\tilde{P}_{ij}$ with the edge $e = (p_j, p_i)$. We replace $\tilde{P}_{ij}$ by e, if it satisfies three conditions: 1) the furthest point p_k from e in $\tilde{P}_{ij}$ is within distance ε_{xy}, (ε_{xy}-*condition*), 2) E is disjoint from $\Delta\tilde{P}_{ij}$ (*disjoint-condition*) and 3) e does not intersect any edge in $\tilde{P} \setminus \tilde{P}_{ij}$ (*non-intersection condition*). If e violates any of these conditions, we recurse on both $\tilde{P}_{ik}$ and $\tilde{P}_{kj}$. In order to prevent the "collapse" of faces (which would violate the homotopy-constraint)[3], our algorithm will not simplify a maximal path to lengths less than 2; rings will not be simplified to lengths less than 3.

What remains is to describe how to decide if a given edge e satisfies the three conditions.

We check the ε_{xy}-condition simply by scanning through all points in $\tilde{P}_{ij}$.

To check the disjoint-condition, that is, to check if E is disjoint from $\Delta\tilde{P}_{ij}$ except possibly for the endpoints of $\tilde{P}$, we navigate the space inside $f \cup f'$ by computing a trapezoidal decomposition $\mathcal{D}_o$ of edges in E using a sweepline algorithm. In addition, we add both endpoints of $\tilde{P}$ to $\mathcal{D}_o$ as empty edges. As in [8], we define the *trapezoidal sequence* $t(Q)$ of a path Q to be the sequence of trapezoids traversed by Q, sorted in the order of traversal. If $t(Q)$ contains the partial sequence $tt't$ for some trapezoids $t, t' \in \mathcal{D}_o$, we will replace this by t. Performing this contraction repeatedly until no more contractions are possible, we obtain a new sequence $t_c(Q)$ called the *canonical trapezoidal sequence* of Q. We trace $\tilde{P}_{ij}$ and e in $\mathcal{D}_o$ to obtain $t(\tilde{P}_{ij})$

[3]Simplifying a path to a single edge will normally not cause problems. However, if the boundary of a face consists of exactly two maximal paths and both of them were simplified to a single edge, the face would collapse to an edge, violating the homotopy-constraint.

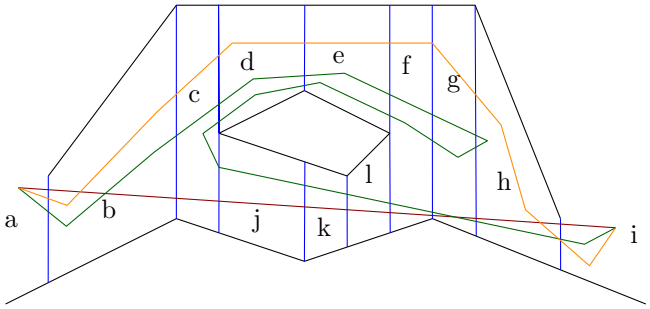


Figure 5.4: Example of using a trapazoidal decomposition to verify homotopy. Here the red line has a trapazoidal sequence of *abcjklfghi* which is also the canonical trapazoidal sequence. The green path has a trapazoidal sequence of *abcdefghgfedcjklfghi* and a canonical trapazoidal sequence of *abcjklfghi*, so the green path can be replaced by the red line, in the simplification. On the other hand the orange line has a trapazoidal sequence of *abcdefghi* which is also its canonical trapazoidal sequence, so the orange path cannot be replaced by the red line.

and $t(e)$ respectively, and verify that $t_c(\tilde{P}_{ij})$ is equal to $t_c(e)$. If they are the same, E is disjoint from $\Delta\tilde{P}_{ij}$ and the endpoint of $\tilde{P}$ is not inside of $\Delta\tilde{P}_{ij}$ (possibly on the boundary). This can be proved by an argument similar to the ones in [23]. See Figure 5.4 for an example.

Finally, we check if e does not intersect $\tilde{P}\backslash\tilde{P}_{ij}$, the non-intersection condition, in a similar way to the disjoint-condition. We first build a trapezoidal decomposition $\mathcal{D}_s$ of all edges in $\tilde{P}$. Then instead of checking the trapezoidal sequence, we check if e crosses any edge $\tilde{P} \setminus \tilde{P}_{ij}$ during the tracing e in $\mathcal{D}_s$. Whenever the trace crosses an edge in $\tilde{P}$, we check if the edge is in $\tilde{P}_{ij}$. If this is not the case, e is an invalid edge.

Correctness. To prove correctness, we need to show that the simplification computed by our algorithm satisfy the above four properties. Property (1) is satisfied since all the edges in E are in $\mathcal{D}_o$, and we explicitly check the disjoint-condition for all edges in P. Since $\tilde{P}$ is contained within $f \cup f'$, the only way to violate Property (2) is for P to intersect the boundary of $f \cup f'$. This is impossible by Property (1). Property (4) is also explicitly checked by the algorithm, when the ε_{xy}-condition is considered. For Property (3), we show that for any P generated by the algorithm, there exists a homotopy function from $\tilde{P}$ to P within ΔP. We first introduce the following lemma.

Lemma 5.7. *Let h and h' be two simple paths that share both endpoints and form a simple polygon Δh. Then, h is homotopic to h' in Δh.*

Proof. The Jordan–Schoenflies theorem [24] states that for any simple closed curve C in the plane there is a homeomorphism $f : \mathbb{R}^2 \to \mathbb{R}^2$ such that $f(C)$ is the unit circle in the plane[4]. Thus, we know that there exists a homeomorphism f that maps Δh to the unit circle in the plane such that h and h' are mapped to the boundary of the unit circle. It is then easy to find a homotopy function H' from $f(h)$ to $f(h')$ on the unit disk; simply moving uniformly along straight lines within the disk will

[4] A homeomorphism is a function $f : X \to Y$ where f is a bijection and f and f^{-1} are continuous.

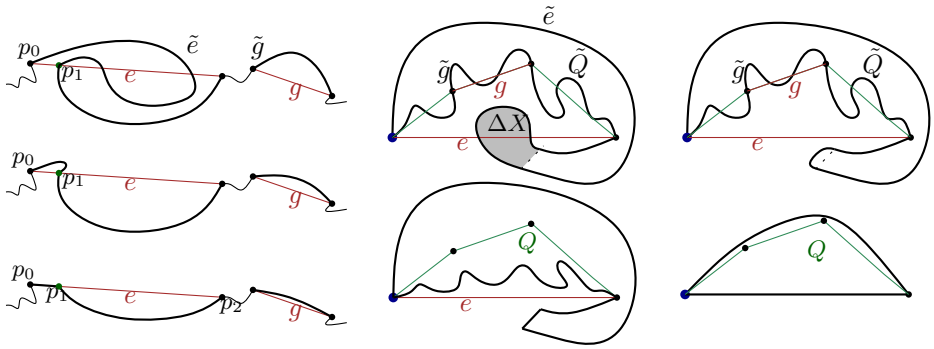


Figure 5.5: Illustration of the two cases (left and right, respectively). (Left) For each simple polygon, we simply deform one to another by finding intersection points. The green point is a stationary point during the deformation. (Right) The blue point is the endpoint of $\tilde{P}$, and it must be an endpoint of e. We first make $\Delta e_{\tilde{Q}}$ empty and deform $\tilde{Q}$ into e and then $\tilde{e}$ into Q.

do. Since f^{-1} is also a homeomorphism, $H(p,t) = f^{-1}(H'(f(p),t))$ will provide a homotopy function from h to h'. □

We consider the polygon Δe for an edge $e \in P$. Note that from Property (1), Δe does not intersect any edge in E. We have two cases to consider: Either Δe is disjoint from Δg for any $g \in P \setminus \{e\}$ (possibly sharing an endpoint) or it is not. In the first case, we define a homotopy function as follows. We trace along e from an endpoint p_0 (say the 0-th intersection) of e to the other endpoint. Whenever we reach the i-th intersection point p_i between e and $\tilde{e}$ that has not been mapped (including the other endpoint of e), we can find a homotopy function H_i from the part of $\tilde{e}$ (from p_{i-1} to p_i) to the part of e (from p_{i-1} to p_i) within the simple polygon Δi they form by Lemma 5.7. Then, we regard all points on $\tilde{e}$ from p_{i-1} to p_i as being mapped. Note that we might ignore some intersections that has been mapped already (refer to Figure 5.5). Let k be the number of intersections found during the mapping. Then, we can obtain a homotopy function H from $\tilde{e}$ to e in Δe as follows:

$$H(p,t) = \begin{cases} p & \text{if } t = 0 \\ H(p, \frac{i-1}{k}) & \text{if } \frac{i-1}{k} < t \leq \frac{i}{k} \wedge p \notin \Delta i \\ H_i(p, tk - i + 1) & \text{if } \frac{i-1}{k} < t \leq \frac{i}{k} \wedge p \in \Delta i \end{cases}.$$

In the other case where Δe and Δg are not disjoint for some g, we note that e does not intersect $\tilde{g}$, g does not intersect $\tilde{e}$ (by the non-intersection condition) and $\tilde{e}$ does not intersect $\tilde{g}$ (by the definition of $\mathcal{P}$). Thus, we know that e and g cannot intersect (as in Lemma 5.3). This means that either Δe is contained within Δg or Δg in Δe, assume without loss of generality that Δg is contained within Δe. Note that the endpoints of $\tilde{P}$ cannot be inside of Δe by the disjoint-condition. Hence there must be a sub-path $\tilde{Q}$ of $\tilde{P}$ going from one endpoint of $\tilde{e}$ to the other endpoint through the interior of Δe containing $\tilde{g}$. Let Q be the simplification of $\tilde{Q}$, $\Delta e_{\tilde{Q}}$ the polygon obtained by closing $\tilde{Q}$ with e, and $\Delta \tilde{e}_Q$ the polygon obtained by closing $\tilde{e}$ with Q. Note that $\Delta e_{\tilde{Q}}$ and $\Delta \tilde{e}_Q$ are simple polygons. If $\tilde{e}$ intersects e we first find a simple polygon ΔX completely contained in $\Delta \tilde{e}_Q$ such that ΔX contains the intersection between $\Delta e_{\tilde{Q}}$ and $\Delta \tilde{e}_Q$ incident to e (refer to Figure 5.5). By Lemma 5.7, we can

find a homotopy function such that all points on $\tilde{e}$ in ΔX map to the boundary of ΔX outside of $\Delta e_{\tilde{Q}}$. Then we can find a homotopy function from $\tilde{Q}$ to e in $\Delta e_{\tilde{Q}}$ by Lemma 5.7. After mapping $\tilde{Q}$ to e, we also find a homotopy function from (modified) $\tilde{e}$ to Q in $\Delta \tilde{e}_Q$ by Lemma 5.7.

5.6 Experiments

We have implemented our algorithm and in this section we present the results of experimenting with it on real-life data. We both compare its performance to the previous algorithm for contour map simplification [8] and investigate its performance on general planer subdivisions by using it to simplify a so-called catchment (or watershed) decomposition of a terrain.

Implementations. We implemented our algorithms using the TPIE library for efficient implementation of I/O-efficient algorithms, utilizing the libraries pipelining functionality [1].

Our *simplification algorithm* was implemented as described in Section 5.3 and 5.5 except for one major optimization, which resulted in a speed-up of an order of magnitude. The optimization is based on the observation that when simplifying a maximal path P we do not actually need to consider all the face edges E of the two faces adjacent to P. Recall that in the internal algorithm in Section 5.5 we simplified P by constructing a trapezoidal decomposition on all edges in E. However, it is easy to realize that only the subset of E within the bounding box B of P can actually constrain the simplification of P to P'. We used this observation in our implementation of the external algorithm in Section 5.3, where we only used the internal algorithm on the edges inside the bounding box of the path when simplifying a maximal path. More precisely, we modified the algorithms such that when considering a face f in phase two, we first built an internal memory Hilbert R-tree T [46] on all face edges of f. Then when loading non-shared face edges of each smaller neighbor face f' into memory in turn and simplifying every maximal path P on the boundary between f and f', we retrieved the necessary face edges inside the bounding box B of P by querying T with B and scanning through the face edges of f' and collecting only edges inside B. As mentioned, this resulted in a significant runtime speedup.

Similarly to the previous *contour map simplification* algorithm of Arge *et al.* [8], we implemented our algorithm to work on a grid terrain model, where the terrain is represented as a regular grid of elevation values, and where input parameters δ, ε_{xy} and ε_z are used to specify that the algorithm should produce a contour map with equi-spaced contours a distance of δ apart and simplify it under the xy- and z-constraints (as discussed in the introduction). As the previous algorithm [8], we fulfill the z-constraint by introducing additional contours at level $h-\varepsilon_z$ and $h+\varepsilon_z$ for each contour at level h,[5] and the input contour map is constructed simply by adding diagonals to the grid terrain model to obtain a triangulation, and then obtaining the contour edges by intersecting each triangle with horizontal planes at the relevant heights. After this preprocessing the contours are simplified with our algorithm as described (using the sweeping based face labeling algorithm discussed in Section 5.3). Since the preprocessing only requires scanning the input grid, it does not dominate the total running time.

[5] The additional contours are not simplified and are not part of the output contours.

	Arge *et al.* [8]	Our algorithm
Input edges	4,786,277,840	4,786,277,840
Running time (hours)	43	26
Output edges (% of input edges)	8.43	8.56
ε_{xy} violations	111,065,821	112,074,000
ε_z violations	262,798,580	263,424,320

Table 5.1: Comparison of results for contour simplification of Denmark. Here ε_{xy} violations is the number of times we recurse in the internal algorithm because of violation of the xy-constraint, and ε_z violations is the number of times we recurse because of violation of the z-constraint (but not the xy-constraint).

Our implementation of an algorithm for *simplifying a catchment decomposition* also works on a grid (terrain model) and takes a simplification parameter ε_{xy}. However, now the grid is interpreted such that neighbor grid cells with the same value are in the same catchment (decomposition face). Our algorithm first constructs the decomposition edges by scanning over the grid and constructing edges between neighbor cells with different values, where edges are merged such that there are no endpoints (nodes) of degree two where both edges are either horizontal or vertical. We also directly augment each edge with the face (catchment value) on each side of the edge (and thus avoiding the sweeping based face labeling algorithm). After this preprocessing the decomposition is simplified with our algorithm as described. Again the preprocessing only requires scanning the input grid and does therefore not dominate the total running time.

Experimental setup and data. We performed all our experiments on a machine with an 8-core Intel Xenon CPU running at 3.2GHz and with 32GB of RAM out of which 13GB were available for our experiment. The machine had a 20 disk raid with a maximal I/O speed of roughly 600 MB/s.

For our contour map simplification experiments we used the same data as in the paper by Arge *et al.* [8], that is, a 2 by 2 meter grid model of Denmark with roughly 12.4 billion grid cells. As in [8], the model was simplified using topological persistence [4, 31] before the experiments, such that depressions and hills with a depth/height less than 0.5 meter were removed.

For our catchment decomposition simplification experiments we used three different grid datasets, all obtained from the 2 by 2 meter grid model of Denmark. The datasets were obtained by running commercial catchment delineation software from SCALGO with thresholds 100, 500, and $500,000$, respectively (in number of grid cells). Intuitively, the software assigns a flow direction for each cell to the steepest downslope neighbor, computes river networks by identifying cells with a number of upstream cells larger than the threshold, and assigns all cells that flow into the same stream junction to the same catchment.

Experimental results. We first compared the practical performance of our algorithm with the previous algorithm for the *contour map simplification* problem. As in Arge *et al.* [8] we constructed a 0.5-meter contour map of all of Denmark (that is, we used $\delta = 0.5m$) and used simplification parameters $\varepsilon_z = 0.2$ meters and $\varepsilon_{xy} = 5$ meters; the ε_z value was chosen to roughly correspond to the z-accuracy of the input

dataset, and the ε_{xy} value was chosen to allow a xy-variation of more than 2 grid cells (and because results in [8] showed that choosing a larger values does not lead to significant further simplification because of the z-constraint). An example of the unsimplified and simplified contours computed in the experiment is given in Figure 5.6, and detailed results are given in Table 5.1 (comparison to the previous algorithm) and Figure 5.7 (resource use of our algorithm). In Table 5.1 it can be seen that the preprocessing phase, identical for the two algorithms, resulted in approximately 4.8 billion edges. The simplification factor of the two algorithms are comparable, both producing an output of size only approximately 8% of the input size. However, our algorithm is significantly faster than the previous algorithm, running in 26 hours versus the previous 43 hours. We believe this is due to its simplicity.

A closer look at the simplification number in Table 5.1 reveals that our algorithm simplifies slightly less than the previous algorithm. We believe the explanation is that our algorithm, unlike the previous one, partitions a contour into several maximal paths (due to degeneracies) and simplifies each path individually, leading to a higher chance of violating the constraints. This is confirmed by Table 5.1 that shows the number of edges added during the algorithms as a result of violating the xy-constraint or the z-constraint (when the xy-constraint was already satisfied).

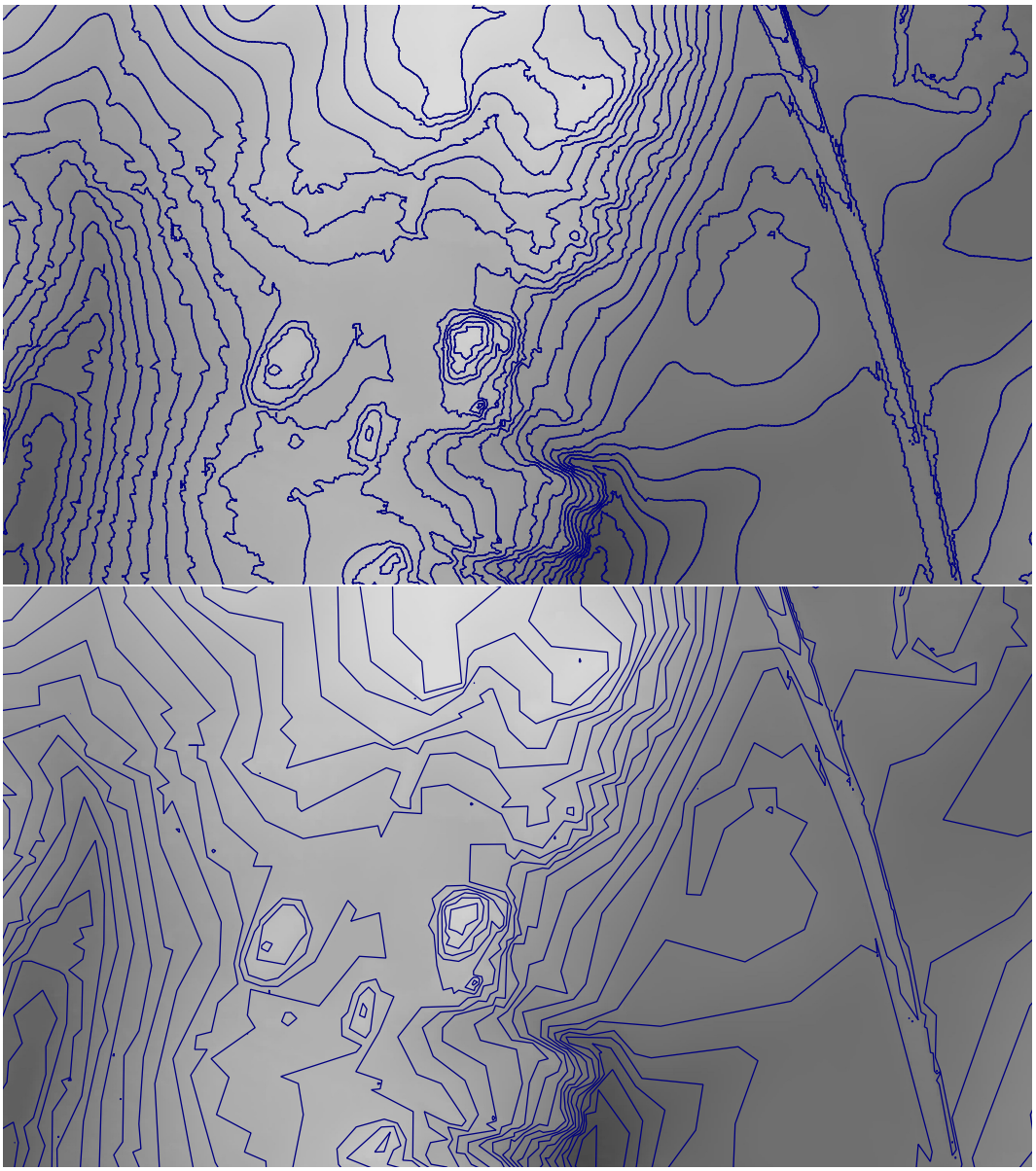


Figure 5.6: Example of unsimplified (top) and simplified (bottom) contours in contour simplification experiment.

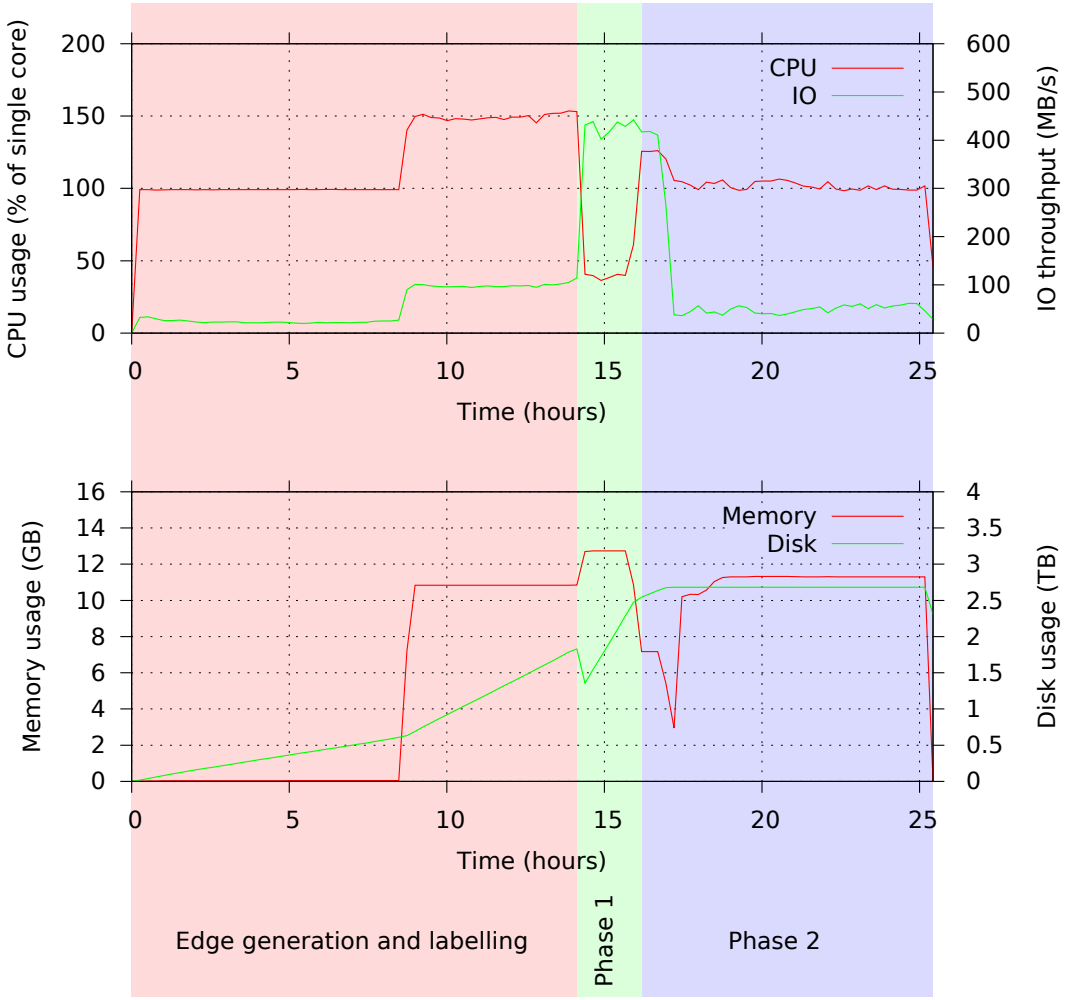


Figure 5.7: Resource usage graphs for the contour simplification experiment. The CPU utilization and I/O throughput are illustrated in the top graph, and the usage of main- and external memory in the bottom graph.

Catchment Threshold	100	500	500000
Input subdivision edges	3,212,696,822	1,239,228,951	102,512,883
Input subdivision faces	387,212,706	75,342,930	787,428
Running time (minutes)	425	117	10
Output edges (% of input edges)	45.30	26.73	8.00

Table 5.2: Results for catchment decomposition simplification of Denmark with different thresholds.

Path Length	1	$[2^1, 2^2)$	$[2^2, 2^3)$	$[2^3, 2^4)$	$[2^4, 2^5)$	$[2^5, 2^6)$	$[2^6, 2^7)$	$[2^7, 2^8)$	$[2^8, \infty)$
Dist. in 100	1.9%	35.3%	7.7%	16.2%	20.8%	11.1%	4.3%	2.1%	0.7%
Output	100%	98.1%	38.6%	18.6%	9.8%	5.5%	2.8%	1.5%	0.5%
Dist. in 500	1.0%	19.9%	2.4%	7.0%	18.0%	27.1%	13.9%	5.1%	5.6%
Output	100%	99.0%	39.7%	18.0%	9.3%	5.6%	3.5%	1.6%	0.6%
Dist. in 500k	0.3%	3.8%	0.1%	0.1%	0.4%	1.7%	6.4%	20.4%	66.8%
Output	100%	99.8%	72.8%	18.2%	9.1%	5.6%	4.5%	4.2%	3.9%

Table 5.3: Distribution of maximal path lengths, along with simplification factor for the various path length intervals, in catchment decomposition simplification of Denmark.

In the *catchment decomposition simplification* experiments designed to investigate the practical performance of our algorithm on general subdivisions, we simplified the three grid decompositions using $\varepsilon_{xy} = 10$ meter; the ε_{xy} was chosen to be somewhat larger than the 2 meter input grid size but without being significantly larger. An example of the unsimplified and simplified catchments computed in the experiment is given in Figure 5.8, and detailed results are given in Table 5.2 and 5.3. In Table 5.2 it can be seen that the preprocessing phase resulted in three subdivisions with approximately 102 million, 1.2 billion and 3.2 billion edges, respectively, and with approximately 800 thousand, 75 million and 387 million faces, respectively. The first interesting thing to note is that the time used on the threshold 100 input is much smaller than the time used on the only slightly larger contour simplification input discussed above. This is due to face labeling being avoided in the catchment decomposition simplification algorithm. It is also interesting to note that the simplification factor is significantly lower than in the contour case, and that it decreases significantly as the catchment threshold decreases (as the number of faces increases). We believe this is due to a large number of maximal paths being too short to allow for a significant simplification. This is confirmed by Table 5.3 that shows the distribution of maximal path lengths for the three datasets, along with the simplification factor for various path length intervals. As it can be seen, there is a significant number of short paths in the two largest datasets and they are not simplified significantly.

Figure 5.8: Example of unsimplified (top) and simplified (bottom) catchments in the catchment decomposition simplification experiment; Colors indicate catchments in the input.

Bibliography

[1] Tpie: The Templated Portable I/O Environment, `http://www.madalgo.au.dk/tpie`

[2] Adelson-Velskii, Landis: An algorithm for the organization of information. In: Proceedings of the USSR Academy of Sciences (1962)

[3] Afshani, P.: On dominance reporting in 3D. In: Proc. 16th European Symposium on Algorithms. pp. 41–51 (2008)

[4] Agarwal, P.K., Arge, L., Yi, K.: I/O-efficient batched union-find and its applications to terrain analysis. Proc. 22nd Symposium on Computational Geometry (2006)

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

[6] Arge, L.: External memory data structures. In J. Abello, P.M. Pardalos, and M. G. C. Resende, editors. Kluwer Academic Publishers (2002)

[7] Arge, L., Brodal, G.S., Truelsen, J., Tsirogiannis, C.: An optimal and practical cache-oblivious algorithm for computing multiresolution rasters. In: Proc. 21st European Symposium on Algorithms. LNCS, vol. 8125, pp. 61–72. Springer (2013)

[8] Arge, L., Deleuran, L., Mølhave, T., Revsbæk, M., Truelsen, J.: Simplifying massive contour maps. Proc. 20th European Symposium on Algorithms pp. 96–107 (2012)

[9] Arge, L., Haverkort, H.J., Tsirogiannis, C.: Fast generation of multiple resolution instances of raster data sets. In: Proc. 21st European Symposium on Algorithms. LNCS, vol. 8125, pp. 52–60 (2013)

[10] Arge, L., Larsen, K.G., Mølhave, T., van Walderveen, F.: Cleaning massive sonar point clouds. Proc. 18th International Conference on Advances in Geographic Information Systems pp. 152–161 (2010)

[11] Arge, L., Truelsen, J., Yang, J.: Simplifying massive planar subdivisions. In: Proc. 16th Algorithm Engineering and Experiments. pp. 20–30. SIAM (2014)

[12] Bădoiu, M., Cole, R., Demaine, E.D., Iacono, J.: A unified access bound on comparison-based dynamic dictionaries. Theoretical Computer Science 382(2), 86–96 (2007)

[13] Bayer, R.: Symmetric binary b-trees: Data structure and maintenance algorithms. Acta Informatica 1, 290–306 (1972)

[14] Bocher, P.K., McCloy, K.R.: The fundamentals of average local variance - part I: detecting regular patterns. IEEE Transactions on Image Processing 15(2), 300–310 (2006)

[15] Borodin, A., Fich, F.E., Meyer auf der Heide, F., Upfal, E., Wigderson, A.: A tradeoff between search and update time for the implicit dictionary problem. In: Proc. 13th International Colloquium on Automata, Languages and Programming, LNCS, vol. 226, pp. 50–59. Springer (1986)

[16] Bose, P., Douïeb, K., Langerman, S.: Dynamic optimality for skip lists and B-trees. In: Proc. 19th Symposium on Discrete Algorithms. pp. 1106–1114. SIAM, Philadelphia, PA, USA (2008)

[17] Bose, P., Howat, J., Morin, P.: A distribution-sensitive dictionary with low space overhead. In: Proc. 11th Workshop on Algorithms And Data Structures, LNCS, vol. 5664, pp. 110–118. Springer-Verlag (2009)

[18] Bose, P., Kranakis, E., Morin, P., Tang, Y.: Approximate range mode and range median queries. In: Proc. 22nd Symposium on Theoretical Aspects of Computer Science. pp. 377–388 (2005)

[19] Brodal, G.S., Kejlberg-Rasmussen, C., Truelsen, J.: A cache-oblivious implicit dictionary with the working set property. In: Proc. 21st International Symposium on Algorithms and Computation, Part II. LNCS, vol. 6507, pp. 37–48. Springer (2010)

[20] Brodal, G.S., Nielsen, J.S., Truelsen, J.: Finger search in the implicit model. In: Proc. 23st International Symposium on Algorithms and Computation. LNCS, vol. 7676, pp. 527–536. Springer (2012)

[21] Brodal, G.S.: Finger search trees. In: Mehta, D., Sahni, S. (eds.) Handbook of Data Structures and Applications, chap. 11. CRC Press (2005)

[22] Brodal, G.S., Kejlberg-Rasmussen, C.: Cache-oblivious implicit predecessor dictionaries with the working set property. In: Proc. 29th Symposium on Theoretical Aspects of Computer Science. vol. 14. Dagstuhl Publishing (2012)

[23] Cabello, S., Liu, Y., Mantler, A., Snoeyink, J.: Testing homotopy for paths in the plane. Proc. 18th Symposium on Computational Geometry pp. 160–169 (2002)

[24] Cairns, S.T.: An elementary proof of the Jordan-Schoenflies theorem. Proceedings of the American Mathematical Society 2(6), 860–867 (1951)

[25] Carr, H., Snoeyink, J., van de Panne, M.: Flexible isosurfaces: Simplifying and displaying scalar topology using the contour tree. Computational Geometry: Theory and Applications 43, 42–58 (2010)

[26] Chan, T.M., Durocher, S., Larsen, K.G., Morrison, J., Wilkinson, B.T.: Linear-space data structures for range mode query in arrays. In: Proc. 29th Symposium on Theoretical Aspects of Computer Science. vol. 14. Dagstuhl Publishing (2012)

[27] Chan, T.M.Y., Chen, E.Y.: Optimal in-place algorithms for 3-d convex hulls and 2-d segment intersection. In: Proc. 25th Symposium on Computational Geometry. pp. 80–87. ACM (2009)

[28] Clark, D.R., Munro, J.I.: Efficient suffix trees on secondary storage. In: Proc. 7th Symposium on Discrete Algorithms. pp. 383–391. ACM/SIAM (1996)

[29] Douglas, D.H., Peucker, T.K.: Algorithms for the reduction of the numbfer of points required to represent a digitized line or its caricature. Cartographica: The International Journal for Geographic Information and Geovisualization 10, 112–122 (1973)

[30] Driscoll, J.R., Sarnak, N., Sleator, D.D., Tarjan, R.E.: Making data structures persistent. Journal of Computer and System Sciences 38(1), 86–124 (1989)

[31] Edelsbrunner, H., Letscher, D., Zomorodian, A.: Topological persistence and simplification. Proc. 41th Foundations of Computer Science pp. 454–463 (2000)

[32] Fisher, P., Wood, J., Cheng, T.: Where is helvellyn? fuzziness of multi-scale landscape morphometry. In: 29th Transactions of the Institute of British Geographers. pp. 106–128 (2004)

[33] Franceschini, G., Grossi, R.: Optimal worst-case operations for implicit cache-oblivious search trees. In: Proc. 8th Workshop on Algorithms And Data Structures. LNCS, vol. 2748, pp. 114–126. Springer-Verlag (2003)

[34] Franceschini, G., Grossi, R., Munro, J.I., Pagli, L.: Implicit B-Trees: New results for the dictionary problem. In: Proc. 43rd Foundations of Computer Science. pp. 145–154. IEEE (2002)

[35] Frederickson, G.N.: Implicit data structures for the dictionary problem. Journal of the ACM 30(1), 80–94 (1983)

[36] Fredman, M.L.: A lower bound on the complexity of orthogonal range queries. Journal of the ACM 28(4), 696–705 (1981)

[37] Frigo, M., Leiserson, C.E., Prokop, H., Ramachandran, S.: Cache-oblivious algorithms. In: Proc. 40th Foundations of Computer Science. pp. 285–297. IEEE (1999)

[38] Garland, M., Heckbert, P.S.: Surface simplification using quadric error metrics. Proc. 24th Special Interest Group on GRAPHics and Interactive Techniques pp. 209–216 (1997)

[39] Greve, M., Jørgensen, A.G., Larsen, K.D., Truelsen, J.: Cell probe lower bounds and approximations for range mode. In: Proc. 37th International Colloquium on Automata, Languages and Programming, Part I. LNCS, vol. 6198, pp. 605–616. Springer (2010)

[40] Harel, D., Tarjan, R.E.: Fast algorithms for finding nearest common ancestors. SIAM Journal on Computing 13(2), 338–355 (1984)

[41] Heckbert, P.S., Garland, M.: Survey of polygonal surface simplification algorithms. Tech. rep., Department of Computer Science, Carnegie Mellon University (1997)

[42] Iacono, J.: Alternatives to splay trees with $\mathcal{O}(\log(n))$ worst-case access times. In: Proc. 12th Symposium on Discrete Algorithms. pp. 516–522. SIAM (2001)

[43] Jacobson, G.J.: Succinct static data structures. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA, USA (1988)

[44] Jacobson, G.J.: Space-efficient static trees and graphs. In: Proc. 30th Foundations of Computer Science. IEEE Computer Society (1989)

[45] JáJá, J., Mortensen, C.W., Shi, Q.: Space-efficient and fast algorithms for multidimensional dominance reporting and counting. In: Proc. 15th International Symposium on Algorithms and Computation. pp. 558–568 (2004)

[46] Kamel, I., Faloutsos, C.: Hilbert r-tree: An improved r-tree using fractals. Proc. 20th International Conference on Very Large Data Bases pp. 500–509 (1994)

[47] Krizanc, D., Morin, P., Smid, M.H.M.: Range mode and range median queries on lists and trees. pp. 517–526 (2003)

[48] Krizanc, D., Morin, P., Smid, M.H.M.: Range mode and range median queries on lists and trees. Nordic Journal of Computing 12(1), 1–17 (2005)

[49] LaMarca, A., Ladner, R.E.: The influence of caches on the performance of sorting. Journal of Algorithms 31(1), 66–104 (1999)

[50] Larsen, K.G.: On range searching in the group model and combinatorial discrepancy. SIAM Journal on Computing 43(2), 673–686 (2014)

[51] Miltersen, P.B., Nisan, N., Safra, S., Wigderson, A.: On data structures and asymmetric communication complexity. Journal of Computer and System Sciences 57(1), 37–49 (1998)

[52] Mortensen, C.W., Pettie, S.: The complexity of implicit and space-efficient priority queues. In: Proc. 9th Workshop on Algorithms And Data Structures. LNCS, vol. 3608, pp. 49–60. Springer-Verlag (2005)

[53] Munro, J.I.: An implicit data structure supporting insertion, deletion, and search in $\mathcal{O}(\log^2 n)$ time. Journal of Computer and System Sciences 33(1), 66–74 (1986)

[54] Munro, J.I., Suwanda, H.: Implicit data structures for fast search and update. Journal of Computer and System Sciences 21(2), 236–250 (1980)

[55] Nash-Williams, C.S.J.A.: Edge-disjoint spanning trees of finite graphs. Journal of the London Mathematical Society pp. 445–450 (1961)

[56] Nash-Williams, C.S.J.A.: Decomposition of finite graphs into forests. Journal of the London Mathematical Society p. 12 (1964)

[57] Pătraşcu, M.: Lower bounds for 2-dimensional range counting. In: Proc. 39th Symposium on Theory of Computing. pp. 40–46 (2007)

[58] Pătraşcu, M.: (Data) STRUCTURES. In: Proc. 49th Foundations of Computer Science. pp. 434–443 (2008)

[59] Pătraşcu, M.: Succincter. In: Proc. 49th Foundations of Computer Science. pp. 305–313. IEEE Computer Society (2008)

[60] Patrascu, M., Thorup, M.: Higher lower bounds for near-neighbor and further rich problems. In: Proc. 47th Foundations of Computer Science. pp. 646–654 (2006)

[61] Petersen, H.: Improved bounds for range mode and range median queries. In: Proc. 34th International Conference on Current Trends in Theory and Practice of Computer Science. pp. 418–423 (2008)

[62] Petersen, H., Grabowski, S.: Range mode and range median queries in constant time and sub-quadratic space. Information Processing Letters 109(4), 225–228 (2008)

[63] Ramer, U.: An iterative procedure for the polygonal approximation of plane curves. Computer Graphics and Image Processing 1(3), 244–256 (1972)

[64] Rosser, J.B., Schoenfeld, L.: Approximate formulas for some functions of prime numbers. Illinois Journal of Mathematics 6(1), 64–94 (1962)

[65] Sleator, D.D., Tarjan, R.E.: Self-adjusting binary search trees. Journal of the ACM 32(3), 652–686 (1985)

[66] Vitter, J.S.: Algorithms and data structures for external memory. Foundations and Trends in Theoretical Computer Science 2(4), 305–474 (2006)

[67] Willard, D.E.: Log-logarithmic worst-case range queries are possible in space theta(n). Information Processing Letters 17(2), 81–84 (1983)

[68] Williams, J.W.J.: Algorithm 232: Heapsort. Communications of the ACM 7(6), 347–348 (1964)

[69] Woodcock, C.E., Strahler, A.H.: The factor of scale in remote sensing. Remote Sensing of Environment 21(3), 311–332 (1987)

[70] Yao, A.C.C.: Should tables be sorted? Journal of the ACM 28(3), 615–628 (1981)