External Memory Graph Algorithms and Range Searching Data Structures

Freek van Walderveen

PhD Dissertation

Department of Computer Science
University of Aarhus
Denmark

External Memory Graph Algorithms and Range Searching Data Structures

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

by
Freek van Walderveen
August 9, 2012

Abstract

Every day larger amounts of data are generated that describe our world in terms of networks or graphs. Think for example about maps of roads or rivers, social networks, or the internet (either as a network of computers or as a network of hyperlinks). Besides this, also surface models, such as height models of the Earth, are often represented using graph structures. Traditionally, graphs were easily processed by computers using algorithms taught to any computer science undergraduate. Nowadays, however, graphs are so big that the assumptions underlying traditional algorithms no longer hold: what used to be a simple graph traversal taking no time worth mentioning, suddenly takes years to complete. This increase is not merely a direct result of the bigger amount of data being processed, but mostly a result of the fact that this amount of data does not fit in the computer's main memory anymore and needs to be retrieved from and stored to slow secondary memory.

In the first part of this dissertation, we therefore present a number of contributions in the area of external-memory graph algorithms, a relatively young research area focusing on developing algorithms for efficiently dealing with graphs that do not fit in main memory. Our contributions include results on removing noise from terrain height models, analysing social networks, and processing planar graphs (that is, graphs that can be drawn without edge crossings, such as road and river networks without bridges and tunnels, or height models).

In order to present (for example geographic) data to a user, it is often necessary to select only a relatively small part of a dataset — such as all post offices in the region visible on the user's screen — and return some statictic about this part — such as the distance between the two furthest post offices in the region, which may help a postal company in determining what delivery time they can guarantee for their customers. Even in non-geometric settings, the part of the data that needs to be selected is often easily described geometrically, for example in database queries asking for records matching multiple numerical criteria.

The second part of this dissertation contains contributions relating to these so-called *range searching* problems, both in the external-memory setting and in the traditional internal-memory setting. We obtain results relating to the example above (finding furthest points in a region), as well as for efficiently finding the set of *categories* present in a certain region — post offices for example, may be categorized by the services they offer.

Acknowledgements

First of all, my PhD supervisor Lars Arge has been a great help in finding problems to work on and people to work with, but also in writing and motivating me. Secondly, Norbert Zeh was a great host for my stay in Halifax, from making the preparations until leaving. Herman Haverkort has not only been my master's thesis supervisor and the one that made me aware of the possibility of a PhD at MADALGO, but was also the one that introduced me to the scientific world in the first place. Initially through an interesting student research project, later by writing papers together, going to conferences, and writing journal articles. I am happy we have stayed in touch after I was formally done in Eindhoven.

One of the most important aspects of research nowadays is collaboration, and among others, my excellent co-authors for the papers in this dissertation have been great to work with: Lars Arge, Pooya Davoodi, Michael Goodrich, Kasper Green Larsen, Thomas Mølhave, Michiel Smid, and Norbert Zeh.

With my office mates Casper Kejlberg-Rasmussen and Kasper Green Larsen I have shared many interesting discussions, and they have simply been great office mates. It was only recently that I found out who is responsible for these office assignments, namely Else Magård. As center administrator for MADALGO, she has always been the person to go to for dealing with any issue outside of theoretical computer science. As such, she, and the rest of the helpful administrative staff at the university, made my stay at MADALGO as smooth as possible. At MADALGO I also met many other interesting people, including fellow PhD students Pooya Davoodi, Lasse Deleuran, Thomas Mølhave, Jesper Nielsen, Jakob Truelsen, and Jungwoo Yang.

Last but not least my parents Hans and Gusta supported me in many ways until the very end.

Thank you all!

Freek van Walderveen,
Aarhus, August 9, 2012.

Contents

Chapter 1

Introduction

As the title suggests, this dissertation contains results on graph algorithms and range searching data structures, with a focus on efficiency in external memory. This introduction is aimed at giving a high-level informal overview of what we understand by these topics. More detailed previous work and background can be found in later chapters.

Many of today's computational problems involve extremely large datasets. As the most practically-oriented part of this thesis deals with terrain data, we will now take a closer look at how such data is typically acquired and processed.

Due to the developments in terrain scanning technology of the last twenty to thirty years, very detailed terrain data can now relatively easily be produced at very high rates. Especially light detection and ranging (LIDAR) technology, operating lasers from airplanes, has significantly increased the size of digital terrain models of land areas. For scanning the seabed, vessels have for a long time been equipped with echo sounders. However, the switch from single- to multi-beam echo sounders (MBES) and further technological advances have caused a significant increase in the data collection rate: current echo sounders can make up to 2.2 billion soundings a day [109].

Typically, raw terrain data is converted to a digital terrain model (DTM), forming a height model of for example the Earth's surface that can be used in computer-assisted terrain analysis in diverse areas of scientific and commercial research. Two of the most commonly used types of models are grids (rasters) and triangulated irregular networks (TINs). In both cases, heights are stored for a set of points. In a grid, the points are simply the vertices of a regular grid, so the coordinates of the individual points do not have to be stored, saving space and processing time. In a TIN, horizontal positions are specified for all points, together with a two-dimensional triangulation of the point set, leading to a more accurate representation compared to grids, and allowing for local differences in representational detail. Constructing such a triangulation for massive point sets can be done in different ways, but by far the most widespread is the following three-step procedure. First, we project the three-dimensional input points down to the plane (ignoring their vertical coordinates), then we construct a two-dimensional *Delaunay triangulation* of the projected point set, and finally we lift this triangulation back to the original heights at the vertices (see Figure 1.1).

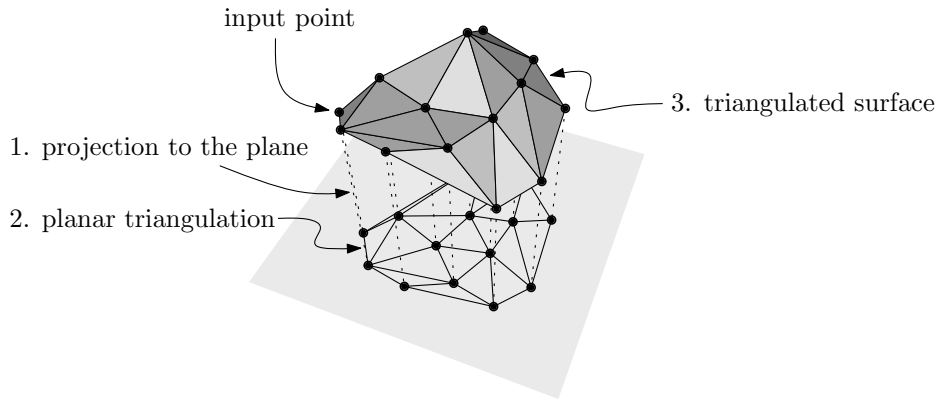

Figure 1.1: Construction of a TIN.

Because of the way DTMs are represented, algorithms working with terrain data are often in essence graph algorithms. The massive amounts of terrain data available today make the area of terrain analysis very interesting from an algorithmic point of view, both at the level of theoretical research and at the level of algorithm engineering. Many very interesting results have already been described in the literature, including efficient algorithms for the generation of TINs, for finding water flow routes and determining where water accumulates, for decomposing a terrain into rivers' watersheds, for topological conditioning to remove minor peaks and depressions from a terrain, and much more. Still, there are many problems left that are not yet solved satisfactorily.

Apart from terrain data, graph algorithms are also useful for other types of geographic data such as road networks, as well as for graphs representing social networks, computer networks, or graphs representing the internet.

Once geographic data has been processed, it needs to be presented to users. Since this type of data is often extremely big and the user may only be interested in a part of the full dataset, efficient data structures need to be developed to store and retrieve such data. In the case of geographic data, the user may for example only be interested in data contained in a certain region. Range searching data structures are made exactly for this purpose: to store a dataset in such a way that all data contained in a (small) *query range* (region) can be retrieved in little time. Of course, many different, often more abstract, applications of range searching data structures exist.

1.1 Models of computation

Just like most other sciences revolve around making, verifying, and updating models of their world in order to make predictions, the science of algorithms revolves around models of computation. In their most elementary form, models of computation only specify a set of operations and their effect on an abstract machine and as such have little predictive value. However, they can also be used to measure and predict the running time of algorithms (or the query time of data structures), as well as to compare the running time of different

algorithms. To support this, models of computation typically focus on what types of operations on data elements in the machine's memory are assumed to be possible in unit time. For example the well-known word-RAM model assumes any basic operation on a data word can be done in unit time. It also assumes that given a (computed) memory address, the machine can access the data at that address in unit time. The pointer machine model is a little more restricted and only assumes constant-time access to data at a known address, disallowing arithmetic on memory addresses. (See Chapter 6 for more details on the word-RAM and pointer-machine model.)

For obvious reasons, the word-RAM model is seen by many as the model that is closest to common hardware, given that the main memory of the machine is large enough to run any algorithm or store any data structure. Another way of putting this is that the word-RAM model (as well as the pointer machine model) essentially assumes the main memory to have infinite size. For certain types of problems though, this assumption is too simplistic and we need a model that also takes bigger and slower *secondary storage* into account for estimating the amount of time an algorithm takes to execute. Specifically, we need to model the access and transfer times for moving data from secondary storage into main memory and vice versa.

The I/O-model

The *I/O-model* (or *external-memory model*) was designed to model the part of the practical running time of algorithms that is due to access and transfer times to and from secondary storage (external memory). It traces back to attempts in the 70's and 80's to analyse the complexity of permuting and sorting data on magnetic hard disks. (Interestingly, this work remains central to both theoretical and practical work in this area, as we shall see later.) The I/O-model as is currently widely used, and adopted in this dissertation, is basically the model Aggarwal and Vitter introduced in 1988 [8]. The model can be seen as a generalization of a model proposed by Floyd in 1972 [59]. Both models consider secondary storage to be divided into blocks forming the unit of data exchange between the bounded-size internal (main) memory and unbounded external memory.

Formally, the model assumes a machine that can load or store any consecutive set of B records from or to external memory in one *I/O*, while the internal memory can hold up to $M = \Omega(B)$ records at a time. Computations can only be done on records in internal memory, but are not restricted in any way. The external memory is assumed to always have sufficient space. To analyse the *I/O complexity* of an algorithm, one proves bounds on the number of I/Os necessary for the execution of an algorithm, given M and B.

For example, scanning an input of N elements (records) takes $\Theta(N/B)$ I/Os. Aggarwal and Vitter [8] showed that a set of N elements can be sorted in $\Theta(\textsc{sort}(N)) = \Theta(N/B \cdot \log_{M/B} N/B)$ I/Os. Many other results on both algorithms and data structures have been obtained since, and more specialized models have been developed to model different types of memory hierarchies.

One of the more successful such models is the so-called *cache-oblivious*

model. The cache-oblivious model captures the possibility of designing I/O-efficient algorithms that do not make explicit use of knowledge of the parameters B and M. Such *cache-oblivious* algorithms work efficiently without explicit tuning for the memory parameters, and are efficient on all levels of even an unknown multilevel memory hierarchy [65]. Cache-oblivious algorithms have been developed for a number of problems, including for sorting [65], where the same $O(\text{SORT}(N))$ I/O-bound can be achieved as in the classic external-memory model.

Other models have also been developed to try to capture current-day hardware more accurately. Actually, the I/O-model has been around for some 24 years now, so it seems obligatory to spend a few words on whether the model and algorithms developed for it will still be usable in the foreseeable future. Therefore, in the remainder of this section we will take a glance at the expiry date of the I/O-model and briefly assess its value in predicting and comparing the practical running times of algorithms.

Future of the I/O-model

Classically, secondary storage is thought of as consisting in a magnetic hard disk, where the access time is the time it takes for the disk head to *seek* to an arbitrary location on the disk. The data is organized in disk blocks (e.g. sectors or pages) containing B bytes, such that the seek time is roughly the same as the time it takes to transfer a block between internal and external memory. The question is to what extent this is still the case in current-day hardware, and whether possible disparities weaken the model's value in predicting practical algorithm running times or not. Finally, we will also discuss the relation between I/O-complexity and internal-memory computation time.

Block size. A correspondence between the seek time t_{seek} of a hard disk and the time B/r_{transfer} to transfer a physical disk block is convenient, since it allows the model to abstract from both values using only one disk-specific parameter — B. Nowadays, however, transfer rates of hard disks have increased by orders of magnitude more than access times [64], while physical disk block sizes have hardly increased at all. To re-balance the equation, one needs to set $B = t_{\text{seek}}/(1/r_{\text{transfer}})$, which on current hard disk drives is on the order of a half to one million bytes [64], while the sector size is at most 4096 bytes [77]. This is of course not a real problem, but one should be aware that the original semantics of the block size is re-interpreted.

Secondary storage technology. Apart from faster magnetic hard disks, other new technologies for secondary storage are also developed. Specifically flash-based solid-state drives are gaining popularity very rapidly. Ajwani et al. [9] investigated the characteristics of flash-based solid-state drives from the perspective of external-memory algorithms and came to the conclusion that many existing algorithms designed for the I/O-model are also efficient on flash-based solid-state drives. For example, the standard I/O-efficient merge sort underlying many I/O-efficient algorithms, as well as the cache-oblivious priority

queue, are also optimal in the *unit-cost flash model* proposed in the paper. The main difference between the I/O-model and the unit-cost flash model is that the latter assumes different block sizes for reading and writing (however still with equal maximum throughput, that is, the same amortized access time per element).

At the same time, hard disk manufacturers do not rest on their laurels and continue developing new technologies to further increase data density, which in the past has consistently lead to higher storage capacity and lower prices [64, 85]. Currently, heat-assisted magnetic recording promises to stretch Moore's law for hard disks another one or two decades [110].

Internal-memory computations. Since the external-memory model *only* models the share of the running time of algorithms caused by external-memory communication, it can only accurately predict and compare practical asymptotic running times in case the internal-memory computations take no more time than external-memory communication. For this to be the case, the amortized time used per transferred data item should be close to $1/r_{\text{transfer}}$. If this is not the case, a decrease in I/O complexity may be bought at the price of increasing the internal-memory computation time, and thereby maybe also the overall practical running time of the algorithm. In the remainder of this thesis, "time" will always refer to internal-memory computation time (independent of the number of I/Os).

In some cases, algorithms can be developed that are both I/O- and time-optimal. For example in the case of sorting, the merge-sort based algorithm of Aggarwal and Vitter [8] can be implemented to achieve optimal $O(N \log N)$ time complexity in the comparison model. Recently, Arge and Thorup [21] show that it is even possible to design a RAM-model sorting algorithm that uses $O(\text{SORT}(N))$ I/Os and $O(N \log_{M/B}(N/B) + \text{RAMSORT}(N))$ time, where $\text{RAMSORT}(N)$ is the best known bound for sorting in the RAM model.

In other cases, I/O-optimal algorithms spend so much time in internal memory that even theoretically, given the relation between B and the external-memory access time mentioned above, they are no better than optimal internal-memory algorithms.

In conclusion, the I/O-model still appears to be usable for some time to come, but other relevant models of computation should not be ignored!

1.2 Contents of this dissertation

This dissertation is based on five papers and manuscripts that have either been published or have been submitted for publication. These contributions are each presented in their own chapter, and are clustered in two parts in this document. Part I contains results on external-memory graph algorithms, and is introduced by means of a survey of known results in this area. Part II describes results on range searching data structures, and starts with a short overview chapter that presents the necessary context for the problems we consider, but does not attempt to give a *full* survey of known results in this vast area of

research. The overview chapter also gives more details on the internal-memory computation models mentioned earlier, since some of our range searching results are developed for those models. In the remainder of this section we give a brief overview of the problems considered and results obtained in each of the chapters. A bibliography of the papers and manuscripts that formed the basis of these chapters is given at the end of this section.

The three main chapters on graph algorithms in Part I contain work ranging from very practical algorithms that have been implemented and tested to more theoretical work that may have a longer way to go before it can have practical impact.

In Chapter 3, we consider a practical problem with terrain data. Apart from using terrain data for the construction of digital elevation models (as sketched earlier), height maps or nautical charts, the data is also used in many more complicated terrain analysis applications such as for analysing the sea floor, for example in the search for oil. Once oil has been found and pipelines have been laid, seabed data (obtained by periodical MBES scannings) is used to maintain the pipeline, for example by controlling the position of the pipe and the movement of the seabed around and below it. However, this type of terrain data is often noisy as a result of scans of (shoals of) fish, multiple reflections, scanner self-reflections, refraction in gas bubbles, and so on. In Chapter 3 we present a new algorithm that avoids the problems of previous local-neighbourhood based algorithms. Our algorithm is theoretically I/O-efficient as well as relatively simple and thus practically efficient, partly due to the development of a new simple algorithm for computing the connected components of a graph embedded in the plane. A version of our cleaning algorithm has already been incorporated in a commercial product. This chapter is based on [16].

Betweenness centrality is one of the most well-known measures of the importance of nodes in a social-network graph. In Chapter 4 we describe the first known external-memory and cache-oblivious algorithms for computing betweenness centrality. We present four different external-memory algorithms exhibiting various tradeoffs with respect to performance. Two of the algorithms are cache-oblivious. We describe general algorithms for networks with weighted and unweighted edges and a specialized algorithm for networks with small diameters, as is common in social networks exhibiting the "small worlds" phenomenon. This chapter is based on [15].

In Chapter 5 we look at a number of external-memory planar graph algorithms exhibiting the problem of excessive internal-memory computation time outlined above. We show how these algorithms can be improved through the development of a new *separator theorem* for cutting planar graphs into regions. Compared to previous separator theorems, ours gives extra guarantees on the shape of regions and the number of disconnected pieces that can make up a region. This allows us to use more efficient internal-memory algorithms for processing these regions in the context of these existing planar graph algorithms, effectively making them optimal both in terms of time and I/Os. This chapter is based on [24].

The two main chapters of Part II deal with specialized types of orthogonal range searching in the plane, for which we develop new internal- and external-

memory data structures and give reductions to establish new lower bounds.

Given a set of points in the plane, range diameter queries ask for the furthest pair of points in a given axis-parallel rectangular range. In Chapter 7, we provide evidence for the hardness of designing space-efficient data structures that support range diameter queries by giving a reduction from the set-intersection problem. The difficulty of the latter problem is widely acknowledged and is conjectured to require nearly quadratic space in order to obtain constant query time, which is matched by known data structures for both problems, up to polylogarithmic factors. We also show that range diameter queries can be answered much more efficiently for the case of points in convex position. This chapter is based on [52].

Range reporting on categorical (or colored) data is a well-studied generalization of the classical range reporting problem in which each of the input points has an associated color (category). A query then asks to report the set of colors of the points in a given rectangular query range, which may be far smaller than the set of all points in the query range. In Chapter 8, we study two-dimensional categorical range reporting in both the word-RAM and I/O-model. For the I/O-model, we present two alternative data structures answering three-sided queries in (1) optimal time using almost-linear space, and (2) near-optimal time using linear space. Both solutions also lead to improved data structures for four-sided queries. For the word-RAM, we obtain optimal data structures for three-sided range reporting, as well as improved upper bounds for four-sided range reporting. Finally, we show a tight lower bound on one-dimensional categorical range *counting* using an elegant reduction from (standard) two-dimensional range counting. This chapter is based on [90].

Bibliography of included work

[16] With Lars Arge, Kasper Green Larsen, and Thomas Mølhave. Cleaning massive sonar point clouds. In *18th ACM SIGSPATIAL GIS*, pages 152–161, Nov. 2010.

[15] With Lars Arge and Michael T. Goodrich. Computing betweenness centrality in external memory. Manuscript submitted for publication, 2012.

[24] With Lars Arge and Norbert Zeh. Multiway cycle separators and I/O-efficient planar graph algorithms. Manuscript submitted for publication, 2012.

[52] With Pooya Davoodi and Michiel Smid. Two-dimensional range diameter queries. In *Proc. 10th Latin American Symposium on Theoretical Informatics (LATIN)*, pages 219–230, Apr. 2012.

[90] With Kasper Green Larsen. Near-optimal range reporting structures for categorical data. Manuscript submitted for publication, 2012.

Part I

External Memory Graph Algorithms

Chapter 2

Survey of External Memory Graph Algorithms

As outlined in Chapter 1, the main motivations behind studying graph algorithms in external memory are in the analysis of networks, such as social networks, computer networks, and graphs representing the internet, as well as in dealing with geographic data, for example in the form of road networks, river networks, and digital elevation models. In each of these areas, many existing datasets do not fit in main memory and have to be stored on disk, causing algorithms that are not optimized for disk I/O to become excessively slow. For algorithms traversing graphs this can be particularly bad if the layout of the graph on disk has no relation to the traversal order. Unfortunately, this discordance is often not resolved by current graph-traversal algorithms, especially those for general graphs (such as directed, non-planar, and/or weighted graphs), resulting in $\Omega(V)$ I/O complexity. The main contributions made in the design of such algorithms are often data structures that record the "state" of the traversal in order to help reduce the dependency of the I/O complexity on the number of edges. For slightly less generic formulations (undirected, planar, and/or unweighted graphs), algorithms are known that reorganize the layout of the graph on disk before starting the actual traversal, resulting in $o(V)$ I/O algorithms.

In a number of cases, the improvement in the I/O-efficiency of a graph algorithm is accompanied by an increase of internal-memory computation time, sometimes by so much that the overall practical running time of the algorithm does not actually improve at all. As discussed in Section 1.1, this is basically the result of ignoring internal-memory models of computation. One could see this as an unforgivable shortcoming of these algorithms, but as is often the case, multiple smaller and bigger steps are necessary before a problem can be considered fully solved.

The remainder of this chapter gives a brief survey of the most important steps that have already been taken in the area of external-memory graph traversal and graph algorithms in general, focusing on results that are relevant for the subsequent chapters. The problems considered and results obtained in those chapters are summarized in the final section.

Connected components and minimum spanning trees. We start our review with a pair of graph problems for which the situation is not as grim as sketched above. In fact, algorithms are known that solve these problems in almost, but unfortunately not quite, as few I/Os as sorting. The problems are, given a graph G with vertex set V and edge set E, to label the connected components of G and to find a minimum spanning forest of G. The first results on these problems were due to Chiang et al. [46], who showed that both can be solved in $O(\text{SORT}(V) + \log \frac{V}{M} \cdot \text{SORT}(E))$ I/Os[1].

The connected component algorithm recursively computes a contracted version of the graph in which the size of the vertex set is halved each time. Since the problem can be solved in internal memory in case $V \leq M$, the recursion is stopped if this is the case. Munagala and Ranade [99] stop the recursion earlier as well as executing multiple recursive steps in one go, resulting in an algorithm using $O(\text{SORT}(V) + \log\log(VB/E) \cdot \text{SORT}(E))$ I/Os. Maybe their most important contribution is a breadth-first search algorithm that requires $O(V + \text{SORT}(E))$ I/Os, which is used for computing the connected components in case $V \leq E/B$. This algorithm will be further discussed below and in Chapter 4.

Arge et al. [14] take the same route to develop a faster algorithm for finding minimum spanning forests, also using a graph contraction procedure to reduce the number of vertices in $O(\text{SORT}(V) + \log\log(VB/E) \cdot \text{SORT}(E))$ I/Os to E/B, and then using an $O(V + \text{SORT}(E))$ I/O algorithm for computing a minimum spanning tree of the contracted graph.

Breadth-first search and depth-first search. Breadth-first search (BFS) and depth-first search (DFS) are two of the most important graph traversals in computer science and have many applications. For general, directed graphs the most I/O-efficient algorithms currently known for these graph traversals are due to Buchsbaum et al. [40] and Chiang et al. [46]. These algorithms use the standard adjacency-list layout of the graph on disk, but use an entirely different approach to dealing with the problem of finding out which vertices have already been visited. The DFS algorithm of Chiang et al. simply rebuilds the graph after M vertices have been visited, resulting in an algorithm using $O(V + \frac{V}{M}\frac{E}{B} + \text{SORT}(E))$ I/Os. Buchsbaum et al. introduce the *buffered repository tree* in order to prevent re-visiting already visited vertices, resulting in algorithms for BFS and DFS using $O((V + \frac{E}{B}) \log \frac{V}{B} + \text{SORT}(E))$ I/Os. For each of the algorithms the actual traversals are done as in internal-memory, using stacks and queues.

For the case of undirected graphs no better DFS algorithms are known, while for BFS faster results have been obtained. As mentioned earlier, Munagala and Ranade [99] present an $O(V + \text{SORT}(E))$ I/O algorithm for this problem. The algorithm constructs the levels of the BFS tree level-by-level from the source vertex. Since the algorithm still uses the standard adjacency-list layout, it needs an I/O for each vertex in each level of the BFS tree for reading its adjacency list. Consecutive levels are constructed directly from the two preceding levels,

[1]The names of sets "V" and "E" are also used to denote the sizes of these sets, whenever the context is clear as to whether we are referencing these as sets or numbers.

so the I/O-complexity is dominated by sorting the adjacency lists for each level. The algorithm is described in more detail in Section 4.2. The most important breakthrough in the area is due to Mehlhorn and Meyer [95], who developed an $O(\sqrt{V(V+E)/B} + \text{SORT}(V+E))$ I/O algorithm for the undirected BFS problem. This algorithm, finally, breaks the $\Omega(V)$ I/O barrier by preprocessing the graph to optimize its layout on disk before performing the actual BFS. (Note that for sparse, connected graphs, the number of I/Os is a factor $\sqrt{B}$ less than before.) Because the input graph is assumed to be undirected (and unweighted), it is possible to partition the graph into disjoint clusters of diameter $O(\sqrt{BV/(V+E)})$. This allows a traversal similar to Munagala and Ranade's, with the difference of maintaining a file containing only the adjacency lists of vertices that are in the same cluster as some vertex in the currently constructed BFS level. This file is scanned and updated for each level, and since the diameter of each cluster is small, the adjacency lists for each cluster do not stay in the file for long. This fact is then used to prove the final I/O bound.

In case BFS trees are required for all vertices in a graph, faster algorithms are known for constructing these trees simultaneously as opposed to running a BFS algorithm from each vertex individually [17, 47]. Specifically, Chowdhury and Ramachandran [47] describe a cache-oblivious algorithm using $O(V \cdot \text{SORT}(E))$ I/Os in total. One of the main building blocks in this algorithm is an "incremental" version of the BFS algorithm of Munagala and Ranade, which is used to more efficiently construct a BFS tree if a BFS tree from a nearby node is already given. We describe this algorithm in Section 4.2. The algorithm can also be used to compute the unweighted diameter of a graph using only $O(E)$ space.

Shortest paths. The single-source shortest-path problem can be thought of as a weighted version of breadth-first search. The currently most I/O-efficient algorithm for this problem is due to Kumar and Schwabe [87] and uses $O(V + \frac{E}{B}\log\frac{E}{B})$ I/Os. For this algorithm, again the graph layout is standard, whereas the interesting part of the algorithm is the external-memory *tournament tree* data structure which is used for maintaining the status of the traversal. More details about this algorithm can be found in Section 4.3.

The best known algorithm for the weighted all-pairs shortest-paths problem is due to Chowdhury and Ramachandran [47] and uses $O(V \cdot \sqrt{VE/B} + V\frac{E}{B}\log\frac{E}{B}))$ I/Os. The algorithm is a slight improvement of an algorithm of Arge et al. [17], and essentially runs V instances of Kumar and Schwabe's algorithm simultaneously, reducing the number of I/Os necessary for accessing adjacency lists. An adaptation of this algorithm is described in Section 4.3. This algorithm computes the weighted diameter using $O(V^2)$ space. It can also be computed using only $O(E)$ space (but $O(V^2 + V\frac{E}{B}\log\frac{E}{B})$ I/Os) by running the Kumar and Schwabe algorithm [87] V times.

Planar graphs. Besides unweighted graphs, the second major graph class for which the single-source shortest-path problem and many others can be solved more efficiently, and in fact often optimally, is planar graphs.

First of all, it should be noted that the contraction-based algorithms for computing connected components and minimum spanning trees run in $O(\text{SORT}(N))$ I/Os[2] for planar graphs since the reduction in the number of vertices results in a similar reduction in the number of edges, such that the recursive costs form a geometrically decreasing series.

For other problems, preprocessing can be used to change the layout of the graph on disk to allow for more efficient graph traversal. This preprocessing consists in constructing a *separator* of $O(N/B)$ vertices dividing the graph into regions of $O(B^2)$ vertices. Maheshwari and Zeh [94] show how to construct a separator with the right properties in $O(\text{SORT}(N))$ I/Os. The standard *tall-cache* assumption, that is, $M = \Omega(B^2)$, guarantees that each of the regions individually fit into main memory. Algorithms for diverse graph problems can now be designed in a three-step framework, by first reducing the graph to only contain separator vertices, then solving the problem on this graph (e.g. single-source shortest paths), and finally using this solution to fill in the regions (e.g. computing shortest-path distances to vertices in the interior of the regions).

A number of problems have been solved I/O-optimally in this framework. Apart from single-source shortest paths [14], also computing strongly connected components and topologically sorting planar directed acyclic graphs (DAGs) [25, 23, 22] takes $O(\text{SORT}(N))$ I/Os. However, they take $\Omega(BN)$ computation time in internal memory. More details about the algorithms can be found in Section 5.4. Finally, for undirected planar graphs, depth-first search can be reduced to breadth-first search in $O(\text{SORT}(N))$ I/Os [18].

Other I/O-efficient algorithms for planar graphs that do not use separators include an algorithm for computing biconnected components in $O(\text{SORT}(N))$ I/Os (using an arbitrary spanning tree as well as a connected-components computation) [46], an alternative algorithm for topologically sorting planar DAGs in $O(\text{SORT}(N))$ I/Os (using an ear decomposition) [23], and an algorithm for directed depth-first search using $O(\text{SORT}(N) \log \frac{N}{M})$ I/Os (using a number of techniques including two-way separators, strongly connected components and topological sorting) [25].

Triangulation. As discussed in Chapter 1, terrain data is mostly represented as grids or TINs, and in both cases basic questions about the terrain can be answered using graph algorithms operating on the graph underlying these models. In order to construct TINs from point sets, we need an algorithm to compute a (Delaunay) triangulation of a two-dimensional point set. This can be done using one of several $O(\text{SORT}(N))$ I/O algorithms for Delaunay triangulation [67, 86, 3].

2.1 Contributions

The contributions in this part of the dissertation range from very practical application-focused work to more theoretical work. In all cases we develop algorithms for dealing with massive graphs in external memory. First, we discuss a topic related to processing terrain data, namely the problem of identifying

[2]For planar graphs, we use N to denote the number of vertices.

noisy points in an MBES dataset. We analyse the problem, develop an I/O-efficient solution, and demonstrate its practicality on a number of real-world datasets. As a by-product we develop a practically efficient connected components algorithm that may be of independent interest. Second, we discuss a topic in social-network analysis, namely the computation of *betweenness centrality* values for all nodes in a graph. We describe a number of algorithms for performing this computation I/O-efficiently both on general, weighted graphs, and on graphs from specific classes, matching the ones commonly found in this area. Third, we revisit the separator-based solutions for planar graph problems. The existing solutions to these problems pay for I/O-efficiency using excessive time in internal memory (mainly for constructing the reduced graphs), thereby negating the performance gain achieved by minimizing the number of disk accesses. We show how to make these algorithms simultaneously efficient in internal and external memory. The key contribution here is the development of a new *multiway simple cycle separator* that allows for a more efficient computation of the reduced graph in all relevant algorithms.

We now give a more detailed overview of the problems and solutions discussed in the following chapters.

Point cleaning

As discussed in Chapter 1, LIDAR and MBES are the most widely used techniques for obtaining detailed terrain data for larger areas. The raw output from these scanners is a *point cloud*: a set of points in $\mathbb{R}^3$ obtained by sending light or sound pulses to determine the distance to the closest object in a certain direction. The pulse may however be reflected before hitting the target: consider for example a bird flying by while an aircraft is making LIDAR measurements or a fish swimming under a ship that is using an echo sounder. Another source of problems for both techniques is that some pulses do not take an ideal path: for LIDAR, light pulses can bounce around multiple times in roof windows before being returned to the scanner, and for echo sounders air bubbles in the water cause similar problems. Actually, MBES data suffers from significantly more noise, and therefore presents a challenging type of data for algorithms removing these spurious measurements. Therefore, in Chapter 3, we look at the problem of cleaning MBES terrain data. Previous work on this problem is also discussed there.

We first analyse what types of noise are commonly found in MBES datasets, and conclude that the type of noise that is most challenging is structural noise in the shape of for example long ribbons. At the same time, the data may also contain features of interest such as pipe lines, that in many ways look just like ribbons of noise, but should under no condition be removed. After the analysis, we describe a cleaning algorithm based on constructing a TIN from the input data, and finding the connected components in a graph constructed from this TIN. We then evaluate the approach and show that removing all but the largest component in the graph is an effective way of removing noise.

As we are interested not only in a theoretically efficient algorithm, but also one that is implementable and efficient in practice, we need practical algorithms

for computing TINs and connected components. For computing a TIN, practical algorithms exist [3, 78], and we use the $O(\text{SORT}(N))$ I/O algorithm of Agarwal et al. [3]. As noted earlier, the current most efficient external-memory algorithm for finding the connected components of a graph is an algorithm by Munagala and Ranade [99]. However, this algorithm is rather involved (the contraction for example requires the computation of Euler tours in trees) and is therefore not of practical interest. Currently the best known practical algorithm for computing connected components is a modified version of an $O(\text{SORT}(N) \log \frac{N}{M})$ union–find algorithm due to Agarwal et al. [4]. This algorithm is unfortunately theoretically less efficient than the algorithm of Munagala and Ranade. Since finding the connected components in a large graph is one of the main components of our cleaning algorithm, we also design a new practical connected component algorithm. The algorithm uses $O(\text{SORT}(N))$ I/Os, assuming a (not necessarily planar) embedding of the graph is given, and the set of edges crossing any horizontal line can be stored in main memory. This assumption holds for many graphs in practice, including the types of graphs constructed by our algorithm.

Overall we obtain a theoretically and practically efficient algorithm for cleaning massive sonar point clouds that works very well in practice. In fact, the algorithm has been further developed by massive data algorithmics start-up SCALGO. In cooperation with marine survey company EIVA they turned the algorithm into a commercial product called SCAN (SCALGO Combinatorial Anti Noise) [56, 57]. At the time of writing, over 30 licenses have been sold for use of the software, on shore as well as on board of ships scanning the seabed.

Computing betweenness centrality

A valuable component of social network analysis involves assigning numerical scores to each vertex in a network based on the "influence" or "importance" of that node in the network, which is commonly referred to as that node's *centrality*. Nodes with high centrality scores are considered to represent entities with high influence or importance. For instance, such nodes are often considered to play a crucial role in the flow of commodities (such as information, drugs, disease, or technology) in their network. Likewise, vertices with low centrality scores are considered to exert relatively less influence and have less of an impact on flow.

One of the most well-known centrality measures is *betweenness centrality*, which is an interesting measure both from the viewpoint of social network analysis and from a computational perspective. Betweenness centrality assigns a value to each node of a network, based on how often that node is on a shortest path between other nodes of the network (an exact definition is given in Chapter 4). The most efficient internal-memory algorithm to compute this measure for all nodes of a graph runs in $O(V^2 \log V + VE)$ time for weighted graphs and $O(VE)$ time for unweighted graphs [37], and shows an interesting relation between computing betweenness centrality and computing the (weighted) diameter of a graph. In Chapter 4, we study this problem in the external-memory setting and find that also here a tight relationship exists between the two problems. In fact, all of our algorithms (optimized for different graph classes) are

either based on all-pairs shortest-path algorithms for computing graph diameters or yield new and improved such algorithms. Specifically, we show the following results for computing betweenness centrality.

- For general unweighted graphs a cache-oblivious algorithm using $O(V \cdot \text{SORT}(E))$ I/Os and $O(E)$ space.
- For general weighted graphs an algorithm using $O(V^2 + V\frac{E}{B}\log\frac{E}{B})$ I/Os and $O(E)$ space.
- For sparse weighted graphs (where $E < VB/\log V$) an algorithm using $O(V \cdot \sqrt{VE/B} + V \cdot \frac{E}{B}\log\frac{V}{B})$ I/Os and $O(V \cdot \sqrt{VE/B})$ space. This algorithm also improves the space of the best known weighted diameter algorithm [47] from $O(V^2)$ to $O(V \cdot \sqrt{VE/B})$.
- For low-diameter weighted graphs a cache-oblivious algorithm using $O(V \cdot E/B \cdot \text{diam}(G))$ I/Os and $O(V^2)$ space on a graph G with (unweighted) diameter $\text{diam}(G)$. This also yields an algorithm with the same bounds for computing the weighted diameter of G.

Multiway cycle separators and planar graph algorithms

As explained earlier in this chapter, many important problems on planar graphs, including single-source shortest paths, topologically sorting DAGs, and computing strongly connected components, have been solved optimally in external-memory with the help of planar graph separators. All of these algorithms achieve an I/O complexity of $O(\text{SORT}(N))$, but the time they use in internal memory is $\Omega(BN)$ in the worst case. At the same time, these problems have linear-time solutions in internal memory [74, 114] that thus directly translate into algorithms with an I/O bound of $O(N)$. Since one I/O corresponds to transferring B elements, these algorithms would effectively also use $O(BN)$ time in the external-memory scenario, and thereby be at least as efficient as the specially-designed external-memory algorithms.

The root of the problem with these algorithms is that after constructing a separator for the graph, they run a BFS or shortest-path computation from each vertex on the boundary of each region, taking $\Omega(B^3)$ time per region. Since there are $\Theta(N/B^2)$ regions, the total time is $\Omega(BN)$. In fact, even the main separator construction itself inherits this problem, as it uses bootstrapping from shortest paths [94].

Klein [82] developed an algorithm that would effectively allow doing these BFS or shortest-path computations in $O(B^2 \log B)$ time per region in case the regions are simple, connected planar graphs. However, existing separator theorems including the external-memory separator of Maheshwari and Zeh [94] in general do not yield such separators.

Our main contribution in Chapter 5 is therefore a separator that guarantees regions to be bounded by a constant number of *boundary cycles*. We then show that with this guarantee, Klein's algorithm can indeed be used, and the problems mentioned above, as well as the separator construction itself, can

be solved in $O(\text{SORT}(N))$ I/Os and $O(N \log N)$ time. In fact, we start out by developing an internal-memory version of our separator that can be constructed in linear time.

Chapter 3

Cleaning Massive Sonar Point Clouds

In order to make further processing feasible, raw multibeam echo sounder data first needs to be cleaned. This cleaning is necessary because the raw data includes a lot of noise, such as (shoals of) fish and other non-permanent objects (see Figure 3.1(a)). Similarly, spurious measurements also create problems. Such measurements appear for example due to multiple reflections, refraction in gas bubbles, influence of the ship's propeller noise, as well as local differences in sound speed due to turbid water [76]. Inaccuracy and miscalibration of measurement devices and the various systems correcting for external influences (such as the pitching and rolling of the ship), can also negatively affect the accuracy of a scan or even result in gross mismeasurements due to scanners detecting their own presence. An example of this type of structural noise is shown in Figure 3.2(a). For most applications, one needs to filter out non-permanent features and gross errors, while for some applications also minor noise has to be filtered out or levelled.

CUBE

Currently, sonar data is often cleaned by hand, possibly with the help of commercially available tools relying on statistical analysis of the data. Often, this is a very tedious and time consuming process. Canepa et al. [43] give a good overview of different types of algorithms and tools that have been proposed in the literature.

Most of the commercially available tools for sonar data cleaning rely on the CUBE algorithm (Combined Uncertainty and Bathymetry Estimator) [41]. The main goal of this algorithm is not so much to remove noise from the data as it is to give depth estimates at the vertices (called *estimation nodes*) of a grid laid over the terrain. The algorithm processes the input points one at a time while maintaining depth estimates at each estimation node, along with information about the accuracy of the estimates based on a statistical analysis of the height and inaccuracy of data points in the neighbourhood of the node. This results in a number of *depth hypotheses* for each estimation node, each supported by a subset of the data points that roughly agree on the depth at that location. After all data points have been processed, a selection needs to be made as to which of these hypotheses are correct and which are formed by

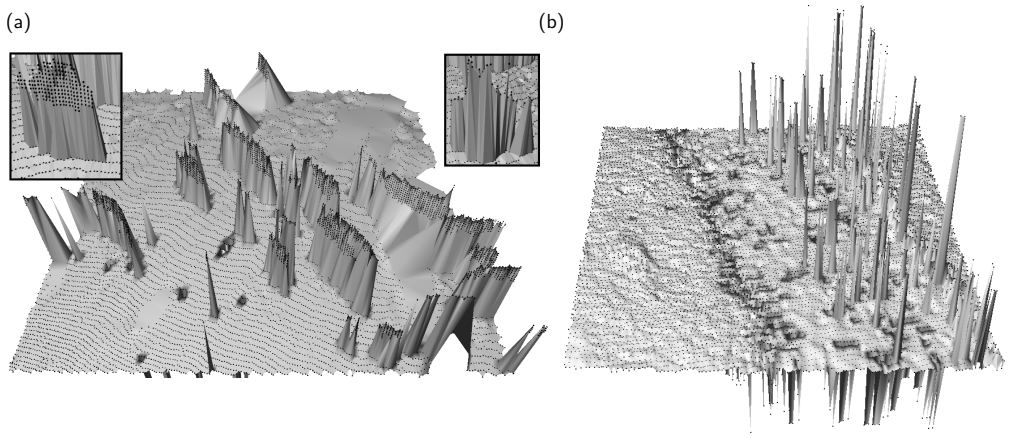

Figure 3.1: (a) Noise caused by fish. Black dots represent the input points. The overlays show the front and back sides of one group of points. (b) Noisy points above and below the seabed (resulting in spikes in the terrain model). Data source: StatoilHydro.

spurious data points. After this is done the data can be cleaned by comparing each input data point to the grid estimates. The hypothesis selection is handled differently by different implementations, but is typically based on one or more of the following methods:

- Selecting the hypothesis that is supported by the most data points.
- Considering a local neighbourhood around each estimation node with multiple hypotheses, finding the closest estimation node v for which there is only one hypothesis, and then choosing the hypothesis at the current node that is closest in depth to the hypothesis at v. The neighbourhood is chosen as an annulus (ring) around the estimation node of a certain pre-defined size (rather than a disc), since it seems to give better results for bursty noise.
- Constructing a lower-resolution approximation of the surface to compare against.
- Using an existing low-resolution, high-quality scan of the area to compare against.

As high-quality scans are often not available for comparison, the last method is not an option in most cases. The other methods work well for single, isolated outliers, but for clustered noise, as in Figures 3.1(a) and 3.2, they often fail to recognize noise, or if they do they often tend to also classify points on the top of pipes lying on (or above) the seabed as noise. The latter is obviously a major problem if the scan was made for pipeline maintenance.

Overall, one main problem with the CUBE method [41] for sonar data cleaning is that only a local neighbourhood around each estimation node is considered, and it may not contain enough information to make the right decision.

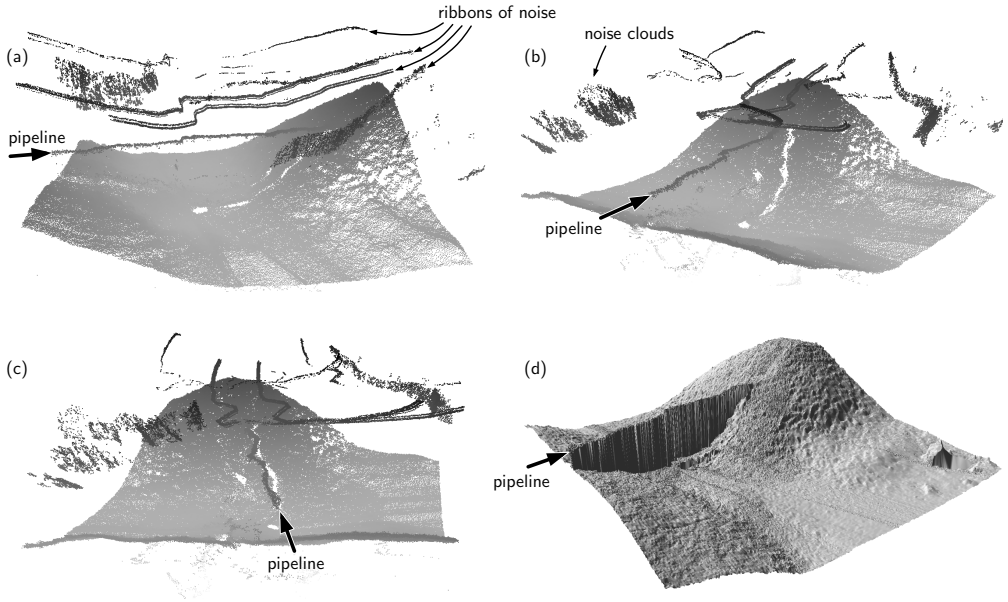

Figure 3.2: Pipeline spanning a valley surrounded by a lot of structural noise. (a-c) Pictures showing the same area from three different angles with a dot for each input point. (The distortion of the pipeline in the horizontal direction is likely the result of a failure to compensate for the motion of the scanner.)
(d) TIN visualization of a cleaned version of the same data.
Data source: StatoilHydro.

Indeed, according to the CUBE User's Manual [42], for CUBE to work properly, separate preprocessing is required to eliminate systematic errors and outliers. Also, determining the best way to use information from the local neighbourhood to select hypotheses is seen as an open problem [41]. Most other methods such as the ones reviewed by Canepa et al. [43] also base their decisions in one way or another on local neighbourhoods, leading to essentially the same problem as with CUBE.

Our contribution

In this chapter we describe a new algorithm for automatic cleaning of massive sonar data, which avoids the problems of local-neighbourhood based algorithms while still allowing an efficient implementation.

In Section 3.1 we first characterize the different types of noise we found in real datasets provided by the companies StatoilHydro, EIVA and others, and discuss why existing local-neighbourhood based algorithms perform poorly for some of these types. Then we describe our new algorithm, and finally we discuss how it performs very well on various real datasets containing all the different noise types. In particular, we show that (unlike existing local-neighbourhood methods) our algorithm is capable of identifying large clusters of noisy points — even clusters with large extent — while for example distinguishing them from points on top of pipes lying on (or, more important, hanging above) the seabed.

In Section 3.1 we also describe how our cleaning algorithm can be implemented to use $O(\text{SORT}(N))$ I/Os under a practically realistic assumption about the input data. The main ingredients in the algorithm are sorting of N elements, triangulation of a set of N points in the plane, and computation of the connected components of a graph of size N embedded in the plane. As discussed, the two first problems can be solved in $O(\text{SORT}(N))$ I/Os with algorithms that are practically efficient. To obtain a practically efficient $O(\text{SORT}(N))$-I/O cleaning algorithm, we in Section 3.2 describe a new simple and practical algorithm for computing the connected components of a graph embedded in the plane. The algorithm uses $O(\text{SORT}(N))$ I/Os under the assumption that any horizontal line intersects at most M edges of the graph. In practice this assumption means that we can handle graphs with $O(M^2)$ edges, as one would expect a horizontal line to hit at most $O(\sqrt{N})$ edges. Thus the assumption certainly holds for practically realistic memory sizes and datasets. We believe that this algorithm is of independent interest since it only requires a sorting step followed by two scans over the graph. Using the practical algorithm of Agarwal et al. [4] instead of our new connected component algorithm would result in a cleaning algorithm without the assumption but using $O(\text{SORT}(N)\log(N/M))$ I/Os. Using the algorithm by Munagala and Ranade [99] we would obtain an $O(\text{SORT}(N)\log\log B)$ algorithm. However, as discussed, this algorithm is not practical.

3.1 Cleaning sonar point clouds

In this section we describe our new theoretically and practically efficient algorithm for cleaning massive sonar point datasets. In Section 3.1.1 we first try to characterize the different types of noise we have experienced in real-life datasets and discuss why existing methods perform poorly for some of these types. Then in Section 3.1.2 we describe our new algorithm, and finally in Section 3.1.3 we discuss how the algorithm performs on various test datasets containing all of the different noise types.

3.1.1 Sonar data noise

Different types of noise can often be found in MBES datasets:

1. Points that appear apparently at random above and below the seabed; sometimes in larger groups. See Figure 3.1(b) for an example.
2. Points resulting from physical objects such as fish, forming larger groups of outliers (as in Figure 3.1(a)).
3. Structural noise, often in the form of ribbons of points appearing along the direction of movement of the echo sounder (as in Figure 3.2).

A main complication in the recognition and removal of noise of the above types is that one typically also finds features on the seabed that are of prime importance for the end user but can be hard to distinguish from noise. Typically such features are objects lying on the seabed or pipelines that either lie

on the seabed or span a "valley" while being supported by "hills" on one or both sides (see Figure 3.2). Existing algorithms can often handle the first type of noise described above, since most points in the neighbourhood of random and spiky noise points have approximately the same height (different from the outliers). Thus it is easy to conclude on statistical grounds that a point is an outlier. For the second type of noise, the effectiveness of traditional cleaning methods typically depends on the size of the local neighbourhood they consider. They typically work well if the neighbourhood is so large that it includes a considerable amount of real data points (clean seabed). If on the other hand the group of outliers is so large that a good fraction or even the majority of points in the neighbourhood is noise, the neighbourhood-based algorithms fail (since they basically need to make an arbitrary decision as to which points are noise and which are from the seabed or an object on the seabed). Finally, in terms of structural noise the neighbourhood-based algorithms face the same problems as for type-2 noise: for example, a pipeline spanning a valley may be locally indistinguishable from a ribbon of structural noise, making it impossible to make a well-founded decision based on local information only.

Since the main problem of most existing methods is the limited size of the considered local neighbourhood, a natural way of improving them is to enlarge the neighbourhood. However, this often makes it harder to compute a good estimate of the terrain (seabed) height at a given position, since the terrain may be very complex in a relatively large neighbourhood. Furthermore, since the running times of the estimation methods are often very dependent on the neighbourhood size, choosing a large neighbourhood may lead to impractical running times.

3.1.2 Our cleaning algorithm

Given a point set P the algorithm first perturbs the horizontal positions of the points in such a way that no two points have the same x- and y-coordinate, resulting in a point set $\tilde{P}$. This perturbation can be done by moving the members of a set of points with the same horizontal position to random positions within a small disc centred around their original location. Next it computes a TIN from $\tilde{P}$ as described in Section 1, by projecting the points onto the plane, constructing a two-dimensional Delaunay triangulation of the projected point set, and lifting this triangulation back to the original heights at the vertices. Next, for each non-boundary edge e of the TIN an edge called e's *diagonal* is added between the two vertices opposite to e in the two triangles incident to e (see Figure 3.3). From this new graph G (embedded in $\mathbb{R}^3$) the algorithm then removes all edges uv where the difference in z-coordinate between u and v is larger than a threshold τ to obtain a graph G_τ. Finally, the connected components of G_τ are computed and all vertices that do not belong to the largest component are marked as noise.

The algorithm can easily be implemented to run in $O(\text{SORT}(N) \log\log B)$ I/Os: in order to give all input points a unique horizontal position we simply sort the points lexicographically by their x- and y-coordinate, and then we can in a simple scan perturb each member of a group of points that share the

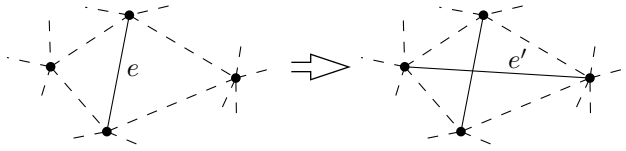

Figure 3.3: Adding the diagonal e' for an edge e in a TIN.

same x- and y-coordinate. As mentioned, the triangulation can be obtained in $O(\text{SORT}(N))$ I/Os [67, 86, 3], and the diagonals can then easily be added and the relevant edges removed using a few sorts and scans. Since both the number of vertices and the number of edges in G is $O(N)$, the connected components can then be computed in $O(\text{SORT}(E) \log\log(VB/E)) = O(\text{SORT}(N) \log\log B)$ I/Os [99]. Finally, after sorting the vertices by their connected component, all noise vertices/points can be marked in two scans: one for counting component sizes and one for marking all points that are not in the largest component. Hence, the total number of I/Os is $O(\text{SORT}(N) \log\log B)$.

Most of the steps of our algorithm can be implemented not only theoretically I/O-efficiently but also practically efficiently, since as discussed in the introduction practical Delaunay triangulation and sorting algorithms have been developed and implemented. As also discussed, the known $O(\text{SORT}(N) \log\log B)$ connected component algorithm [99] is too complicated to be of practical interest, whereas the simpler practical algorithm [4] uses $O(\text{SORT}(N) \log(N/M))$ I/Os. However, in Section 3.2 we describe a new algorithm that is relatively simple and practically efficient, and which under a realistic assumption uses only $O(\text{SORT}(N))$ I/Os. We have implemented and tested the resulting algorithm on a number of massive MBES datasets.

3.1.3 Algorithm performance.

In this section we study the performance of our algorithm in terms of cleaning quality. We also intuitively motivate the different parts of the algorithm and discuss the effect of the threshold τ.

Overall cleaning performance. We first discuss the overall performance of our algorithm in terms of the noise types identified in Section 3.1.1. Noise caused by fish (type 2 above) provides a useful example to get intuition about how and why the algorithm works. Consider the TIN shown in Figure 3.4(a), where a fish above the surface is scanned. It is easy to see what happens when we remove all edges of the TIN with a height difference more than τ: if the fish swims further away from the seabed than this threshold, the edges connecting its points to the points on the seabed are removed and the points on the fish form a small connected component in G_τ and are therefore classified as noise. Even though we do not catch fish swimming too close to the seabed, this is probably the best we can do. If a fish swims too close to the seabed it could just as well be a stone lying on the seabed. It is quite clear however that the fish in Figure 3.4(a) is not an object on the seabed. The reason for this (and a rule of thumb amongst manual data cleaners) is that if an object is resting

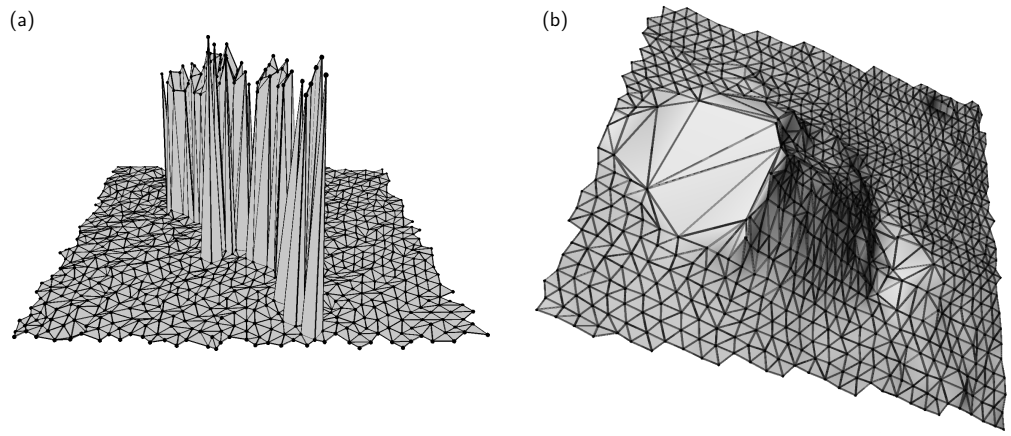

Figure 3.4: (a) TIN of points on a fish that swims approximately 60 cm above the seabed. (b) A stone blocking the view for the echo sounder (scanning from the right). Data source: StatoilHydro.

on the sea floor, also points on the side of the object will be captured by the sonar since it is scanning the object at an angle. A cleaned version of the area around Figure 3.4(a) is shown in Figure 3.8(b).

For the same reasons as for physical objects such as fish, our algorithm also handles random outliers (type 1 above) well, as long as they are far enough from the sea floor. Refer for example to the spikes below the terrain in Figure 3.8(a), visible as lines of dark spots on the top side, that are removed by our algorithm as shown in Figure 3.8(b).

For structural noise (type 3 above) the situation is more interesting. If the noise forms dense ribbons we, as argued previously, get a pattern that is locally indistinguishable from an elevated pipeline (for example Figure 3.2(a)). Still, as illustrated in Figure 3.2 and 3.8(c)–(d), the algorithm correctly removes ribbons of noise while keeping the pipeline intact. The reason is that if one looks along the whole length of the pipeline it will at some point rest on the seabed or a sandbank. This enables our algorithm to distinguish pipes from noise: points on a pipe will be part of the same connected component as the seabed in G_τ because the pipe physically "connects" to the seabed. On the other hand, the points of a ribbon of noise are typically connected to each other but not to the sea floor.

Algorithm intuition. The above discussion of how our algorithm handles the different noise types also to some extend gives the intuition behind the algorithm: defining a "closeness" relation on the input points by constructing a surface graph from them, and then disconnecting parts of the graph that are far away from the predominant surface. As surface graph we use a Delaunay triangulation (a TIN DEM) but we could of course have used other graphs. However, the Delaunay triangulation has a few specific advantages. First of all, it always yields a connected and somewhat regular graph. This property is useful especially in lower-resolution areas of the terrain (see Figure 3.4(b))

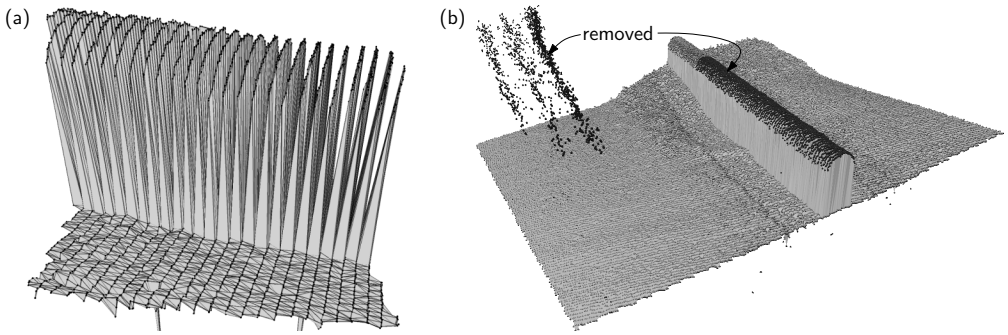

Figure 3.5: (a) TIN of points on a pipe and below it. (b) Pipeline partially separated from the seabed, causing the algorithm *without diagonals* to remove the black points, which are also on the pipeline. Data source: StatoilHydro.

where for example a graph based on fixed-size neighbourhoods would leave holes and might even disconnect the graph (note however, that this is not directly related to the main problem with neighbourhood-based algorithms). A second advantage of the Delaunay triangulation is that practical and efficient construction algorithms are available. A possible alternative to using a TIN (with diagonals) would be to use a 3D Delaunay triangulation. However, the worst-case complexity of this triangulation is $O(N^2)$, and although a worst-case optimal $O(N^2/B)$-I/O algorithm is known [86], the size of the triangulation would make the cleaning algorithm inhibitingly slow both in theory and in practice. Note that a consequence of using a planar triangulation is that it does not allow points at different heights to share the same x- and y-coordinate. Our algorithm works around this limitation by perturbing points with the same horizontal position.

What is maybe less intuitive about our algorithm is the addition of diagonals. This added connectivity is needed to handle some cases of type-3 noise, or rather, to distinguish such noise from a pipeline above the surface. When an echo sounder scans a terrain with overhangs (where at some horizontal positions multiple true surfaces exist at different heights), it usually reports points at the different heights because it is scanning at an angle. The triangulation of such a point set typically consists of many spikes (both up and down), because points on for example a higher surface may happen to only have edges to points on a lower surface (see for example the close-up in Figure 3.5(a), where each line of points on top of the pipe forms a separate component). Removing all these long edges would result in the points being wrongly classified as noise (as in Figure 3.5(b), where the black points are removed by the algorithm when diagonals are not added). By adding the diagonals we make it much more likely that a point is connected to at least one other point on the same surface. This is actually what happens in most practical situations because of the high resolution of MBES point sets and the properties of the Delaunay triangulation. Because the top of the pipe will be more or less evenly sampled, points along the pipe connect to each other in G_τ through the diagonals. The points at the place where the pipe is supported by the seabed then provide a connection, so

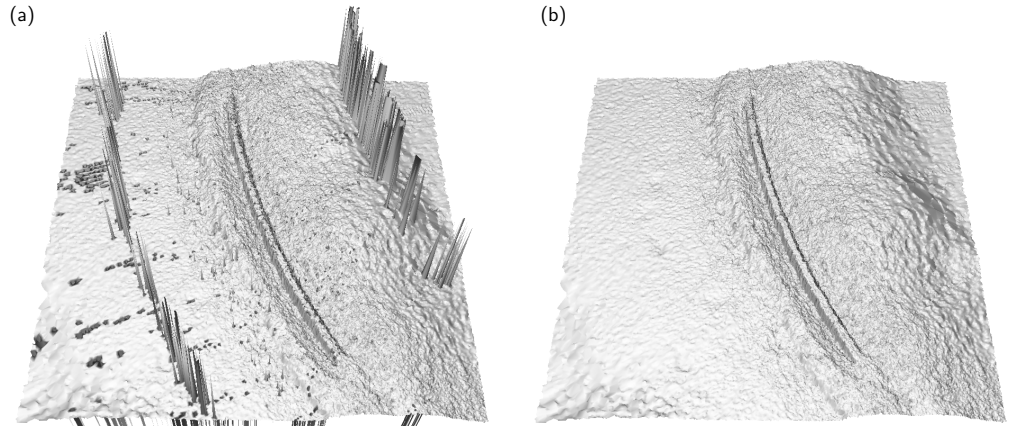

Figure 3.6: Excerpt of dataset I. (a) Raw. (b) Cleaned. Data source: EIVA.

there will be a path of neighbouring points between the top of the pipe and the seabed. Hence, these points and the ones on the seabed form one component in G_τ and the points will not be removed (see for example Figure 3.8(d)).

Detailed cleaning performance (threshold selection). The threshold τ obviously has a major influence on the performance of our cleaning algorithm. In the remainder of this section we discuss the effect of varying τ while also further describing the algorithm's cleaning performance.

In the examples shown in Figure 3.8 we used a threshold τ of 5 cm, which is on the order of the scanner accuracy in those datasets. In general, we expect picking τ a bit higher than the scanner accuracy (a few standard deviations, if that is known) to yield a good result. The intuition behind this rule of thumb is that when the terrain is sampled densely enough, sample points that are close to each other on the surface of the terrain have a height difference that mainly depends on the scanner accuracy. Then, we know that for two points on the same surface, there will be a path of connected points in the triangulation for which each two consecutive points have a height difference lower than τ. We have a similar situation when an object is lying on the sea floor. For any point p on an object of interest there will be a path of close points in G from p to a point on the seabed. Because they are often scanned at an angle, at least on one side there will be a curtain of points reaching from the top of the object to the sea floor, establishing a connection that our algorithm will use to determine that the points forming the object are not noise. Any point for which such a path does not exist is therefore very likely noise.

In some cases a threshold corresponding to the scanner accuracy might be too low and result in parts of the seabed being disconnected from the rest of the seabed (due to low local point resolution). In such cases one would have to use a higher threshold, and our experience is that in many cases it is still possible to remove a large amount of noise using a relatively high threshold. Below we discuss the effect of varying the threshold using four datasets that contain moderate to extreme amounts of noise, as well as different features on

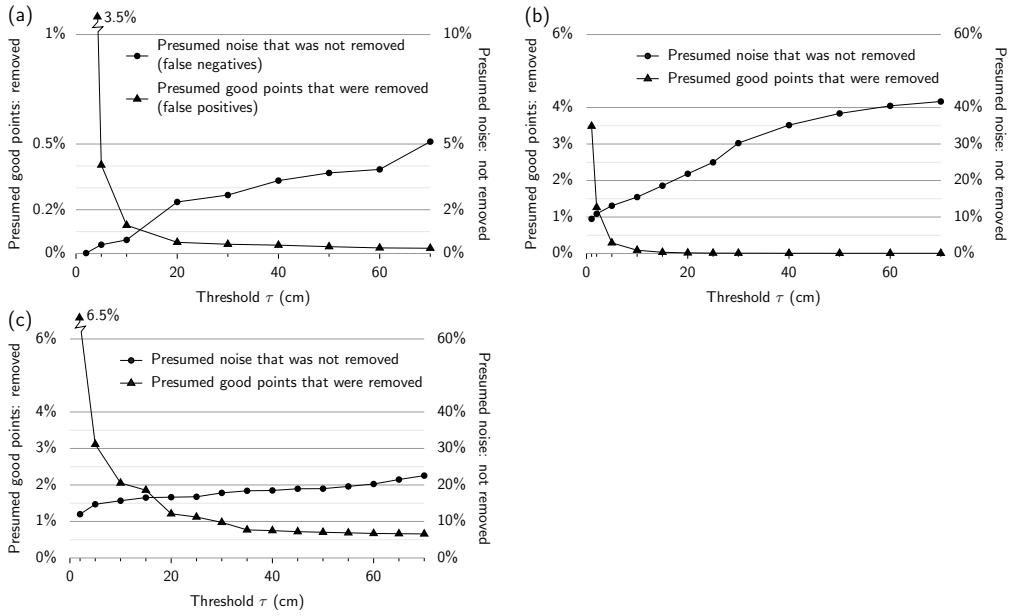

Figure 3.7: Classification differences between manual and automatic cleaning.
(a) Dataset I (see Figure 3.6(a)) of about 7 million points (manual cleaning removed 0.5% of these).
(b) Dataset II of about 7 million points (manual cleaning removed 1.7% of these).
(c) Dataset IV (see Figure 3.2(a)) of about 6 million points (manual cleaning removed 20.5% of these).

the seabed. Excerpts of the raw data are shown in Figures 3.6(a) (dataset I), 3.8(c) (dataset III), and 3.2(a) (dataset IV). Dataset II looks similar to dataset I. For datasets I, II and IV we also had access to a manually cleaned version, allowing for direct comparison. It should be noted though that some of this manual cleaning was rather "rough", so not all outliers were removed. Also for an operator it is not always clear which points should be classified as outliers, so two well-cleaned datasets (by different operators or automated methods) may still have a large number of differently classified points. Nevertheless, we have compared the number of points that are classified differently by our algorithm and the manual process. The results of this comparison are shown in Figure 3.7 (note that the scale on the left indicates the percentage of presumed good points that were removed by the algorithm, while the scale on the right indicates the percentage of presumed outliers that were kept).

For datasets I and II we consider the results to be very convincing: at a threshold of 5 cm 99.6% and 87% of the noise was removed, while only 0.4% and 0.3% of the points that were kept during manual cleaning were also removed. For both datasets threshold values of 2 or even 1 cm still visually give very good results, but as one can see the false-positive rates increase rapidly. For dataset I there also exist some points in isolated parts of the point set at the far end of the viewing angle of the scanner that are removed while they probably

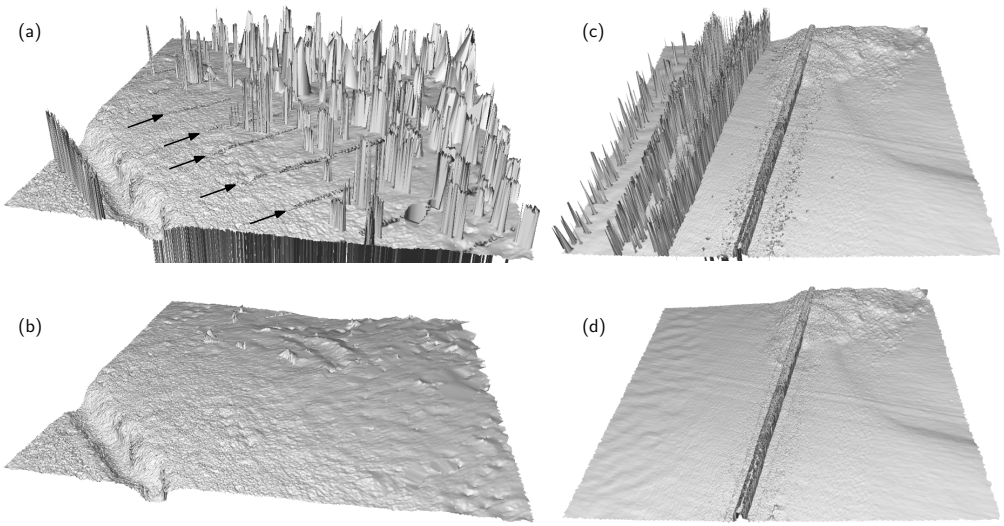

Figure 3.8: (a) Noise caused by fish, with a ribbon of noise in the front left, and some noise below the seabed visible as dark lines on the top (indicated by arrows). (b) Cleaned version of (a).
(c) A pipeline spanning a valley (close-up in Figure 3.5(a)), with structural noise on the left. (d) Cleaned version of (c). Data source: StatoilHydro.

represent the real sea floor. For most applications this would not be a big problem, as such data is considered to be of inferior quality anyway. In both datasets, increasing the threshold results (as one would expect) in an increase of the false-negative rate and a decrease of the false-positive rate. Still, at for example 50 cm the most visible and in that sense important outliers were removed in both datasets.

For dataset III we did not have access to a manually cleaned dataset. However, visual inspection shows that at a threshold of 5 cm (with 5.7% of the points marked as noise), the cleaning is effective and the pipeline is kept intact (see for example Figure 3.8(d)). Like with datasets I and II, setting τ to 2 cm keeps most of the terrain and specifically the pipeline intact, while at $\tau = 1$ cm some parts with poor resolution and parts of the pipe were removed.

Dataset IV is the most noisy dataset we have seen. According to the manual cleaning, 20% of the points are outliers. At the same time the dataset contains a poorly sampled pipeline spanning a valley, which creates problems at low thresholds (where parts of the pipe are classified as noise). Therefore, increasing the threshold to 35 cm results in a visually nicely cleaned dataset where all ribbons of noise are removed, and compared to manual cleaning 81% of all outliers and only 0.8% non-outliers are removed. Up to $\tau = 50$ cm this is still the case, but with higher thresholds more and more ribbons of noise start appearing.

We conclude that for datasets with moderate noise and relatively good sampling conditions, our cleaning algorithm is both effective and leads to very few false positives with hardly any intervention (and using the same threshold for different datasets). Also for datasets with overhangs in the form of pipelines

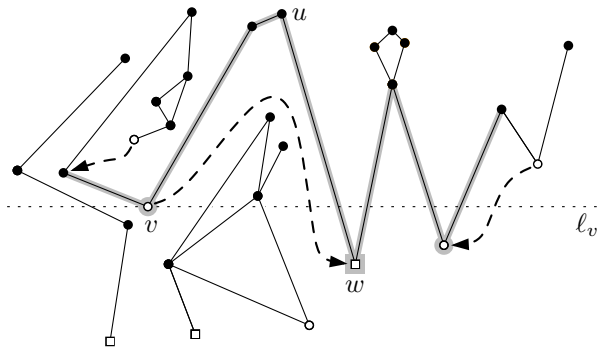

Figure 3.9: Status during the down phase when processing vertex v. Open vertices do not have any lower neighbours; square vertices are the lowest in their component; arrows indicate augmented vertices.

the algorithm works very well. For data with extreme noise and poor sampling conditions the algorithm still works effectively, but may require some work by the operator, either by marking misclassified components or by increasing the threshold.

3.2 Connected components

In this section we describe a practically and theoretically efficient $O(\text{SORT}(N))$-I/O algorithm for computing the connected components of a graph G embedded in the plane (possibly with intersecting edges). The algorithm assigns a label to each vertex of G, such that two vertices have the same label if and only if they are in the same connected component of G. However, it only works under the assumption that any horizontal line intersects at most $O(M)$ edges of G. Our algorithm is inspired by an algorithm due to Danner et al. [50, 51] for computing the connected components in a two-dimensional bitmap.

Our algorithm consists of a *down phase* and an *up phase*. Both phases sweep a horizontal line over the plane while maintaining connectivity information for vertices incident to edges that cross the sweep line. For simplicity, we assume that G does not contain any vertices without incident edges, and that the y-coordinates of all vertices of G are distinct. These assumptions can easily be removed.

Consider a horizontal line ℓ_v through a vertex v of G and let G_v^+ denote the graph induced by all edges of G with at least one vertex on or above ℓ_v. In the down phase, we augment each vertex v with no neighbours below ℓ_v (open vertices in Figure 3.9) with an arbitrary vertex w below ℓ_v that is in the same connected component as v in G_v^+, if existing (arrows in Figure 3.9). In the up phase we then use the augmented information to compute the connected component labels. Below we discuss the two phases in more detail.

Down phase. In the down phase, we sweep a horizontal line from $y = \infty$ to $-\infty$. Whenever the sweep line encounters a vertex v with no neighbour below ℓ_v

we augment v with w as defined above. To be able to find w we maintain a data structure $\mathcal{D}$ in internal memory during the sweep. The structure $\mathcal{D}$ maintains *upwards connected component* information, that is, it maintains information about which of the vertices on or below ℓ_v are in the same connected component in G_v^+ (in Figure 3.9, the vertices with a grey background are in the upwards connected component of v in $\mathcal{D}$). The structure $\mathcal{D}$ is essentially a union–find data structure supporting the following operations:

- Insertion of a vertex.
- Merging of two upwards connected components.
- Extraction of a vertex in $\mathcal{D}$ in the same upwards connected component as the top vertex in $\mathcal{D}$, if such a vertex exists.
- Removal of a vertex.

The assumption that the sweep line intersects $O(M)$ edges ensures that $\mathcal{D}$ fits in internal memory.

More precisely, the sweep proceeds as follows: We scan the vertices of G in decreasing order of y-coordinate. When processing a vertex v, we first insert v in $\mathcal{D}$ if it is not there already. Then we load all edges having v as highest vertex into memory. We do so by scanning the edges of G in order of the y-coordinate of their highest vertex, along with the scan of the vertices. For each edge vw we then check whether w is already in $\mathcal{D}$. If not, we insert w in $\mathcal{D}$ and merge the upwards connected component of v with the newly created component of w. Otherwise, we simply merge the component of v with that of w. If no such edge vw exists, that is, v has no neighbour below ℓ_v, we instead extract a vertex in $\mathcal{D}$ in the same upwards component as v and augment v with this vertex w, if existing. Finally, we remove v form $\mathcal{D}$.

It is easy to realize that $\mathcal{D}$ is maintained correctly during the sweep, that is, when the sweep line is at v, it contains information about which vertices on or below ℓ_v are in the same connected component in G_v^+. It immediately follows that the down phase correctly augments each vertex v that has no neighbour below ℓ_v with a vertex below ℓ_v that is in the same connected component as v in G_v^+ (that is, in the same upward component).

Up phase. In the up phase we compute the connected components of G and assign a label to each vertex. To achieve this, we also sweep a horizontal line, but this time from $y = -\infty$ to ∞. During the sweep, we maintain the invariant that any vertex that has a neighbour below the sweep line has had its component label assigned, that is, any two such vertices have the same label if and only if they are in the same connected component of G. Thus, when the sweep line reaches ∞ we have computed the connected components of G, that is, labelled all vertices correctly.

While performing the sweep, we store all edges of G intersecting the sweep line in internal memory. We also separately store all vertices incident to these edges in internal memory, where each such vertex also stores the label assigned

to it. The assumption that the sweep line intersects $O(M)$ edges ensures that these vertices and edges fit in internal memory.

The sweep now proceeds as follows: We scan the vertices of G in order of increasing y-coordinate, along with the edges of G in order of the y-coordinate of their lowest vertex. From the down phase we know that when processing a vertex v with no lower neighbour, it is augmented with a vertex w below ℓ_v that is in the same component as v in G_v^+, if existing. To determine v's label we first check if v is already stored in internal memory. If it is, that is, if it has a lower neighbour, it has already been assigned a label. If not and v is augmented with a vertex w, which must then be in internal memory, we assign the label of w to v. If v is not augmented with a vertex we assign it a new label. Next we load all edges with v as their lowest vertex into internal memory, while adding v's neighbour vertices to the vertices stored in internal memory and assigning each such vertex the same label as v. Finally, we remove all edges that have v as their highest vertex from internal memory, while also removing vertices from internal memory that are left with no incident edges among the edges in internal memory.

To see that the above algorithm maintains the invariant, first note that if v's label is correctly assigned then so are all other labels assigned when processing v (since they are all assigned v's label and are connected to v). In the case that v is assigned the label of vertex w in internal memory, we know that v is in the same connected component as w in G_v^+ and thus in G, and therefore the label is correctly assigned. In case v is assigned a new label, we know that v is not connected to any vertex below ℓ_v in G_v^+ and thus not to any vertex below ℓ_v in G. Therefore the newly assigned label maintains the invariant.

I/O-complexity. The down phase requires the vertices to be ordered by decreasing y-coordinates and the edges by decreasing y-coordinates of their highest vertices. Similarly, the up phase requires the vertices to be ordered by increasing y-coordinates and the edges by increasing y-coordinate of their lowest vertex. To bring the edges and vertices in this order, we simply sort them, using $O(\text{SORT}(N))$ I/Os. The remainder of the two phases consist only of scanning over sorted lists of vertices and edges taking $O(N/B)$ I/Os. Thus the total cost is $O(\text{SORT}(N))$ I/Os.

Remark. We actually only need to sort the edges and vertices for the down phase. The down phase can then simply push vertices and edges to I/O-efficient stacks after they are processed. This will reverse the order for the up phase and thus save a sorting step. Note that if the vertices and edges are given in the correct order for the down phase, the cost of the algorithm can be reduced to $O(N/B)$ I/Os.

3.3 Conclusion and future work

In this chapter we described an algorithm for removing noisy points from multibeam sonar data that can handle arbitrarily large clusters of noisy points. As

opposed to previous algorithms that base their decisions only on a local neighbourhood around each point, our algorithm can distinguish noise clusters from points on top of physical objects like pipelines. We showed that the algorithm can be implemented to be both theoretically and practically I/O-efficient, in part due to the development of a new practical connected components algorithm. A version of the algorithm has already been incorporated in a commercial product.

Our results open up a number of interesting theoretical questions. First of all, it would be interesting to quantify why our algorithm works so well. To do so, one needs a mathematical model of noise. Existing such models are mostly only concerned with noise caused by scanner inaccuracy and sample points are assumed to be relatively close to the actual surface [53]. An algorithm is then asked not to discard certain points, but to make the best estimate of the surface's position and topology. However, in our case a model would need to consider the addition of high amounts of outliers to a terrain sample and ask for algorithms to classify points to be noise or not. An interesting approach might be to take a simple model of clustered outlier noise that considers samples in a confined region of a smooth terrain to have a given probability of being an outlier (and thus lying relatively far away from the actual terrain). In this model, our algorithm classifies a point p on the terrain within this region correctly, if and only if there is a path in graph G from p to a point on the terrain outside the region such that all consecutive points on the path are on the terrain. One can model this as a colouring of the vertices of graph G: points on the terrain are white and noise points are black. The cleaning quality of our algorithm then depends on the connectivity of the subgraph of G induced by the white points. The connectivity properties of graphs coloured randomly like this are studied in the field of *percolation theory*. Indeed, a result that may shed light on the cleaning quality of our algorithm has been obtained by Bollobás and Riordan [34]. They consider random Voronoi percolation, which corresponds to connectivity in randomly coloured Delaunay triangulations (without diagonals) of infinite random point sets. The result they obtain concerns the *critical probability* in the setting where vertices are coloured white or black independently with a given probability. The critical probability can be thought of as the highest possible probability of vertices being coloured black, before it becomes very likely that the white vertices get separated into small connected components (as opposed to one big component). Bollobás and Riordan prove this critical probability to be $1/2$. In short, percolation theory could be a possible way of quantifying how well an algorithm like ours handles clustered noise. Thus it would open up the possibility of comparing different variations of it, for example based on other types of neighbourhood graphs.

Chapter 4

Computing Betweenness Centrality in External Memory

Betweenness centrality is one of the most well-known measures of the importance of nodes in a social-network graph (see for example [29, 33, 37, 38, 62, 63, 91, 102]). Given an undirected graph $G = (V, E)$, the betweenness centrality of a vertex $v \in V$ is defined as

$$C_B(v) := \frac{1}{2} \cdot \sum_{s \neq t \in V \setminus \{v\}} \frac{\sigma_{st}(v)}{\sigma_{st}} \ ,$$

where σ_{st} is the number of shortest paths between s and t ($\sigma_{ss} = 1$ by convention) and $\sigma_{st}(v)$ is the number of shortest paths between s and t that contain v. Since $\sigma_{st}(v) = 0$ if s and t are in different connected components, we consider connected (and undirected) graphs in this chapter.

There are also other definitions of centrality, ranging from simple statistics, like degree centrality [84] and closeness centrality [30], to sophisticated statistics, like Katz centrality [81] and random-walk/current-flow centrality [39, 103]. In terms of topological insights, however, betweenness centrality is superior to these other statistics in measuring the importance of vertices for the sake of the flow of information (or other commodities) along geodesic paths. For instance, it has been used as a way to identify interdisciplinary journals in scientific collaboration networks [91].

From a large-data algorithmic perspective, betweenness centrality is arguably the most interesting centrality measure as well. Simple centrality statistics, such as degree centrality and closeness centrality, can be computed trivially from a graph's representation or immediately after an all-pairs shortest-paths computation; hence, they do not pose much of a computational challenge. Alternatively, sophisticated centrality measures, like Katz centrality and random-walk centrality, seem to require matrix inversion computations; hence, they are not practically computable for large networks, especially when those networks are sparse, as is often the case. The best known algorithms for betweenness centrality, on the other hand, avoid matrix inversions yet nevertheless involve interesting graph traversals in addition to all-pairs shortest-paths computations (e.g., see [37, 102]). Thus, our interest in this chapter is on efficient algorithms

for computing betweenness centrality for the vertices in large networks, specifically when the space required for our algorithm is larger than our computer's internal memory. Therefore, our focus in this chapter is on external-memory algorithms for computing the betweenness centrality of each vertex in a large network.

Our contribution

The best internal algorithm for computing betweenness centrality is due to Brandes [37] and runs in $O(V^2 \log V + VE)$ time for weighted graphs and $O(VE)$ time for unweighted graphs. Unfortunately, the algorithm, which we review in Section 4.1, does not easily translate into an efficient external-memory algorithm, as it involves a large number of pointer hops (random memory accesses).

Although main memory size is often the primary constraint on the size of the graphs for which one can compute betweenness centrality efficiently in practice [37], the betweenness centrality problem has not previously been considered in the external-memory model.

In this chapter, we provide I/O-efficient and cache-oblivious algorithms for computing betweenness centrality, both in weighted and unweighted graphs. As mentioned above, to the best of our knowledge, no such algorithms were previously known, even though I/O is often a constraint on the size of the graphs one can compute betweenness centrality on in practice.

In Section 4.2, we consider unweighted graphs and describe how to obtain a betweenness centrality algorithm using $O(V \cdot \text{SORT}(E))$ I/Os and $O(E)$ space. This algorithm is cache-oblivious. In Section 4.3, we consider weighted graphs and describe a number of new external-memory algorithms. First, we show how to obtain an $O(V^2 + V\frac{E}{B}\log\frac{E}{B})$ I/O and $O(E)$ space algorithm, which works for general graphs but is admittedly not cache-oblivious. Next we describe an improved $O(V \cdot \sqrt{VE/B} + V \cdot \frac{E}{B}\log\frac{V}{B})$ I/O and $O(V \cdot \sqrt{VE/B})$ space algorithm for sparse graphs (where $E < VB/\log V$). The algorithm performs several phases with a number of concurrent instances of a shortest-path algorithm rather than just one phase with V concurrent instances. Interestingly, this algorithm also improves the space of the best known weighted diameter algorithm [47] from $O(V^2)$ to $O(V \cdot \sqrt{VE/B})$, which may be of independent interest. Finally, we describe a further improved algorithm for low-diameter graphs, which are common for social networks that exhibit the "small world" phenomenon (e.g., see [33]). This algorithm uses $O(VE/B \cdot \text{diam}(G))$ I/Os and $O(V^2)$ space on a graph G with (unweighted) diameter $\text{diam}(G)$; the algorithm is cache-oblivious.

4.1 The betweenness centrality algorithm of Brandes

In this section we review the internal-memory algorithm of Brandes [37] for computing betweenness centrality of a graph $G = (V, E)$. The main idea of the algorithm is to decompose the betweenness centrality of a node v into the contributions of the shortest paths passing through v in G that start in each of the other nodes. The sum of the contributions of the shortest paths starting in

a node $s \in V \setminus \{v\}$ to the betweenness centrality of v is called the *dependency* of s on v, denoted $\delta_s(v)$, and is defined as follows:

$$\delta_s(v) := \sum_{t \in V \setminus \{v\}} \frac{\sigma_{st}(v)}{\sigma_{st}} \ .$$

The betweenness centrality of a vertex v is then equal to the sum of the dependencies of all vertices on v:

$$C_B(v) = \frac{1}{2} \cdot \sum_{s \neq t \in V \setminus \{v\}} \frac{\sigma_{st}(v)}{\sigma_{st}} = \frac{1}{2} \cdot \sum_{s \in V \setminus \{v\}} \delta_s(v) \ .$$

Below we describe an $O(E)$ time and space algorithm for computing the dependency values $\delta_s(v)$ for all $v \in V$ for a particular $s \in V$ in an unweighted graph G (a graph where all edges have weight one). To compute $C_B(v)$ for all $v \in V$ we simply run this algorithm for each $s \in V$, and maintain for each $v \in V$ the sum of all $\delta_s(v)$ computed so far. Thus overall the betweenness centrality is computed for all vertices in an unweighted graph G in $O(VE)$ time using $O(E)$ space. At the end of the section we discuss how to extend the algorithm to the weighted case.

Computing $\delta_s(v)$ for fixed s. Let $P_s(v)$ be the set of *predecessors* of v with respect to s, that is, the neighbours of v that are part of at least one shortest path between s and v, and $S_s(v)$ the set of *successors* of v with respect to s, that is, the set of vertices w that are neighbours of v such that v is on at least one shortest path from s to w. We now have the following:

Lemma 4.1 ([37], Lemma 3) *For $v \neq s$,*

$$\sigma_{sv} = \sum_{u \in P_s(v)} \sigma_{su} \ .$$

Lemma 4.2 ([37], Theorem 6) *For $v \neq s$,*

$$\delta_s(v) = \sum_{w \in S_s(v)} \frac{\sigma_{sv}}{\sigma_{sw}} \cdot (1 + \delta_s(w)) \ .$$

The algorithm of Brandes [37] for computing $\delta_s(v)$ for all $v \in V$ in $O(E)$ time and space first computes $P_s(v)$ and $S_s(v)$ for all $v \in V$ in $O(E)$ time and space using a breadth-first search from s. During the breadth-first search σ_{sv} can also easily be computed when visiting node v using Lemma 4.1, since all vertices $u \in P_s(v)$ are visited before v. Finally, the vertices are visited in reverse breadth-first search order and $\delta_s(v)$ is computed when visiting v using Lemma 4.2; this is possible since all vertices $w \in S_s(v)$ have already been visited when visiting v.

Weighted graphs. The above algorithm can easily be modified to work for weighted graphs, where we are given a weight function $w : E \to \mathbb{R}^+$. To construct $P_s(v)$ and $S_s(v)$ for all v we use Dijkstra's algorithm [54], rather than a breadth-first search. The rest of the algorithm remains unchanged, so the total running time becomes $O(VE + V^2 \log V)$, since Dijkstra's algorithm uses $O(E + V \log V)$ time.

4.2 An I/O-efficient algorithm for unweighted graphs

In this section, we show how to compute betweenness centrality for the vertices of an unweighted graph $G = (V, E)$ in $O(V \cdot \text{SORT}(E))$ I/Os and $O(E)$ space. Similar to the internal-memory algorithm of Brandes [37], our external-memory algorithm computes a breadth-first search (BFS) tree from each vertex s in G while using Lemma 4.1 to compute σ_{sv} for each vertex $v \neq s$ (while also maintaining for each vertex the sum of all $\delta_s(v)$ computed so far). However, unlike the internal case we do not construct the BFS trees for each vertex $s \in V$ independently, since even with the best known external BFS algorithm [95] this would require $\Omega(V \cdot \sqrt{VE/B})$ I/Os. Instead we modify the algorithm by Chowdhury and Ramachandran [47] that constructs BFS trees from all vertices s in $O(V \cdot \text{SORT}(E))$ I/Os in total. The main idea of the algorithm is that a BFS tree from vertex s can be constructed more I/O-efficiently when a BFS tree from a vertex s' close to s is already known. This allows us to perform reverse-order I/O-efficient processing of the BFS trees.

The algorithm first computes an ordering π of the vertices of G such that the distance between consecutive vertices in π is small, and then it constructs the BFS trees from the vertices in π one after the other, utilizing an *incremental BFS* algorithm. Below we first describe how to compute π and then we describe the incremental BFS algorithm. After that we discuss how to modify the BFS algorithm to compute betweenness centrality, and finally we analyse the algorithm.

Computing π. The ordering π of the vertices used in the algorithm by Chowdhury and Ramachandran [47] is constructed as follows. First an (arbitrary) spanning tree T of G is computed and then two directed edges uv and vu are constructed for each undirected edge uv in T to obtain directed graph T'. Then the edges of T' are ordered to form an Euler tour and the order $\pi = \langle s_1, \ldots, s_{|V|} \rangle$ of the vertices in V is defined by the order in which they first occur in the Euler tour.

Note that although the distance $d(s_i, s_{i+1})$ between two consecutive vertices s_i and s_{i+1} in π may be large, the sum of the distances between all consecutive vertices in π is $O(V)$ since the length of the Euler tour is $O(V)$. Furthermore, Chowdhury and Ramachandran [47] also prove the following property of π:

Lemma 4.3 ([47]) *Let $d(u, v)$ be the length of the shortest path between u and v, and $d_i = d(s_{i-1}, s_i)$ for $1 < i \leq V$. Then for any vertex $v \in V$, $d(s_i, v) - d_i \leq d(s_{i-1}, v) \leq d(s_i, v) + d_i$.*

Incremental BFS. The main idea in the incremental BFS algorithm for computing a BFS tree from s_i given a BFS tree from s_{i-1} is to order the adjacency lists of all vertices according to their level in the BFS tree for s_{i-1}. More precisely, assume the vertices in G are labelled with their distance from s_{i-1} in the BFS tree from s_{i-1} and store the adjacency lists of vertices at distance d in a list $\mathcal{A}_d$; within $\mathcal{A}_d$ the adjacency lists are ordered by vertex. Lemma 4.3 above then implies that we are guaranteed to find the adjacency lists for vertices at distance ℓ from s_i in some list $\mathcal{A}_k$ for $\ell - d_i \leq k \leq \ell + d_i$. Using this property we now construct the BFS tree from s_i one level at a time as in the algorithm by Munagala and Ranade [99]: Let $\mathcal{L}_\ell$ denote the sorted list of vertices at level ℓ in the BFS tree from s_i; level $\mathcal{L}_1$ consists of vertex s_i. Given the previous two BFS levels $\mathcal{L}_{\ell-2}$ and $\mathcal{L}_{\ell-1}$, we construct $\mathcal{L}_\ell$ by first scanning $\mathcal{L}_{\ell-1}$ simultaneously with each list $\mathcal{A}_k$, where $\ell - d_i \leq k \leq \ell + d_i$, and inserting an edge uv in a list $\mathcal{E}_\ell$ for each vertex v that is a neighbour of a vertex $u \in \mathcal{L}_{\ell-1}$. We then sort $\mathcal{E}_\ell$ by the second vertex and scan it simultaneously with $\mathcal{L}_{\ell-1}$ and $\mathcal{L}_{\ell-2}$ to remove any edges $uv \in \mathcal{E}_\ell$ for which $v \in \mathcal{L}_{\ell-1}$ or $v \in \mathcal{L}_{\ell-2}$. This means that $\mathcal{E}_\ell$ now contains edges uv connecting a vertex u in level $\ell - 1$ with a vertex v in level ℓ of the BFS tree. Finally, we scan the sorted $\mathcal{E}_\ell$ to aggregate all edges $u_1v, \ldots, u_mv$ for a vertex v and append v to $\mathcal{L}_\ell$.

Computing betweenness centrality. To compute betweenness centrality, we first modify the incremental BFS algorithm above such that it also computes σ_{s_iv} for each vertex v when computing the BFS tree from s_i. To do so we augment each vertex u in the sorted list $\mathcal{L}_{\ell-1}$ of vertices at level $\ell-1$ in the BFS tree from s_i with σ_{s_iu}. We then modify our algorithm for computing $\mathcal{L}_\ell$ from $\mathcal{L}_{\ell-1}$ and $\mathcal{L}_{\ell-2}$ in order to be able to compute σ_{s_iv} for each vertex v in $\mathcal{L}_\ell$. The modification consists of annotating each edge uv in $\mathcal{E}_\ell$ with σ_{s_iu}, which allows us to compute σ_{s_iv} using Lemma 4.1 when aggregating edges $u_1v, \ldots, u_mv$ and inserting v in $\mathcal{L}_\ell$. The annotation is performed simply by sorting the list $\mathcal{E}_\ell$ of edges uv by first vertex and then simultaneously scanning through $\mathcal{E}_\ell$ and $\mathcal{L}_{\ell-1}$ to annotate each edge uv in $\mathcal{E}_\ell$ with σ_{s_iu} from $\mathcal{L}_{\ell-1}$.

After constructing the BFS tree from s_i along with σ_{s_iv} for each vertex v, that is, the lists $\mathcal{L}_1, \mathcal{L}_2, \ldots$ of vertices at each level of the tree and the lists $\mathcal{E}_2, \mathcal{E}_3, \ldots$ of edges between the levels, we compute $\delta_{s_i}(v)$ for each vertex as in Brandes' algorithm [37] by visiting the vertices (levels) in reverse order. To compute $\delta_{s_i}(v)$ for all vertices u in $\mathcal{L}_{\ell-1}$ when $\delta_{s_i}(v)$ has already been computed for all vertices v in $\mathcal{L}_\ell$, we first annotate each edge uv in $\mathcal{E}_\ell$ with $\delta_{s_i}(v)$ and σ_{s_iv} by sorting $\mathcal{E}_\ell$ by second vertex and simultaneously scanning $\mathcal{E}_\ell$ and $\mathcal{L}_\ell$. After that we sort $\mathcal{E}_\ell$ by first vertex and scan the sorted list to obtain all edges $uv_1, \ldots uv_m$ incident to each vertex u in $\mathcal{L}_{\ell-1}$ and compute $\delta_{s_i}(u)$ using Lemma 4.2.

I/O complexity. To analyse our algorithm, first note that the ordering π can be computed in $O(V \cdot \text{SORT}(E))$ I/Os using a minimal spanning tree algorithm by Arge et al. [14] and standard external graph algorithm techniques [46, 116] (see for example Chowdhury and Ramachandran [47]).

To initiate the incremental BFS process, we generate the BFS tree from s_1 in $O(V + \text{SORT}(E))$ I/Os using the algorithm by Munagala and Ranade [99]. Now to construct the BFS tree from s_i and compute $\delta_{s_i}(v)$ for all vertices $v \in V$, we sort the adjacency lists into lists $\mathcal{A}_k$ using $O(\text{SORT}(E))$ I/Os and then we construct the levels of the BFS tree one at a time and traverse them again in reverse order. For level ℓ, we scan lists $\mathcal{A}_{\ell-d_i}, \ldots, \mathcal{A}_{\ell+d_i}$, so in total we scan each list $\mathcal{A}_k$ $O(d_i)$ times, using $O(d_i \cdot E/B)$ I/Os. For each level we also scan $\mathcal{L}_{\ell-1}$ once, so every level of the BFS tree is also scanned $O(d_i)$ times, using $O(d_i \cdot V/B)$ I/Os in total. The remainder of the algorithm consists of scanning and sorting the vertex and edge sets a constant number of times using $O(d_i \cdot E/B + \text{SORT}(E))$ I/Os in total. Overall we use $O(\sum_i (d_i \cdot E/B + \text{SORT}(E))) = O(V \cdot (E/B + \text{SORT}(E))) = O(V \cdot \text{SORT}(E))$ I/Os in total to construct and traverse all of the BFS trees. The amount of space required is $O(E)$, as we only store the set of edges and vertices a constant number of times.

Theorem 4.1 *Betweenness centrality for an unweighted and undirected graph $G = (V, E)$ can be computed in $O(V \cdot \text{SORT}(E))$ I/Os using $O(E)$ space.*

Remark: Using a cache-oblivious minimum spanning tree and Euler tour construction algorithm [13] rather than the I/O-efficient algorithms [14, 46], our algorithm can easily be made cache-oblivious with the same I/O and space bounds, since the incremental BFS algorithm is based only on sorting and scanning of lists.

4.3 I/O-efficient algorithms for weighted graphs

In Section 4.1 we noted that Brandes' internal-memory algorithm can also be used for graphs $G = (V, E)$ with positive edge weights $w : E \to \mathbb{R}^+$, by using a shortest-path algorithm instead of breadth-first search. Thus we can obtain an $O(V^2 + V\frac{E}{B}\log\frac{E}{B})$ I/O and $O(E)$ space algorithm by running a simple modification of the single-source shortest-path algorithm of Kumar and Schwabe [87] from all V vertices. We describe this algorithm in Section 4.3.1. In Section 4.3.2, we then describe an improved $O(V \cdot (\sqrt{VE/B} + (E \log V)/B))$ I/O and $O(V \cdot \sqrt{VE/B})$ space algorithm for sparse graphs where $E < VB/\log V$. The algorithm is a modified version of the previous all-pairs shortest-paths algorithm by Chowdhury and Ramachandran [47] and by Arge et al. [17] that computes shortest paths from multiple sources concurrently. In Section 4.3.3, we consider the case of low-diameter graphs and describe an $O(VE \cdot \text{diam}(G)/B)$ I/O and $O(V^2)$ space algorithm, where $\text{diam}(G)$ is the (unweighted) diameter of G.

4.3.1 The general algorithm

In this section, we sketch the I/O-efficient single-source shortest-path algorithm of Kumar and Schwabe [87], and then show how an additional step allows us to compute betweenness centrality.

The algorithm of Kumar and Schwabe is a modified version of Dijkstra's algorithm that relies on an I/O-efficient priority queue called the *external tournament tree*, which supports a DecreaseKey operation with a slightly unusual semantic: Given an element x and a priority p, the operation updates the priority of x to p only if p is smaller than the current priority of x (inserting x if it does not exist in the queue).

Lemma 4.4 ([87, 17]) *Given a universe of N elements, an external tournament tree using $O(B)$ internal memory can process a sequence of k* DeleteMin, Delete, *and* DecreaseKey *operations in $O(\frac{k}{B}\log\frac{N}{B})$ I/Os in total.*

I/O-efficient Dijkstra. To compute the shortest paths from vertex s to all other vertices, Dijkstra's algorithm [54] processes vertices in order of their distance from s while maintaining a tentative shortest distance $d_t(s,v)$ from s to all non-processed vertices. When processing a vertex v, each neighbour u of v is retrieved and if $d_t(s,u) > d(s,v) + w(v,u)$ the tentative distance to u is updated. To find the next vertex to process, unprocessed vertices are kept in a priority queue with their tentative distance as their priority.

The two main challenges faced by external-memory implementations of Dijkstra's algorithm are to determine which neighbour vertices are already processed when processing a vertex, and which unprocessed neighbours need to have their priority updated (decreased). The I/O-efficient shortest-path algorithm of Kumar and Schwabe [87] solves the second issue using the special semantic of the external tournament tree. The first issue is solved by updating the priority of a vertex irrespective of whether that vertex has already been processed, and using a second tournament tree to take care of removing *spurious updates*, that is, updates to already processed vertices, from the first one. Refer to [87, 116] for details[1]. This results in an algorithm using $O(V+\frac{E}{B})$ I/Os in total for accessing the adjacency lists of the vertices, and $O(E)$ operations on the two tournament trees using $O(\frac{E}{B}\log\frac{E}{B})$ I/Os (Lemma 4.4). Hence, in total the algorithm uses $O(V+\frac{E}{B}\log\frac{E}{B})$ I/Os for computing the shortest paths from one source.

Computing betweenness centrality. For computing betweenness centrality, we first describe how we can compute the number of shortest paths σ_{sv} and then the dependency $\delta_s(v)$ of s on each vertex v after having computed $d(s,v)$ for all vertices using the Kumar and Schwabe algorithm. To do so we first compute the actual directed acyclic graph (DAG) of shortest paths from s by identifying all edges vu where $d(s,u) = d(s,v) + w(v,u)$. We can easily do so by annotating edge vu with $d(s,v)$ and $d(s,u)$ in a few sorting and scanning steps of the edges and vertices, similar to the way we annotated edges in the algorithm in Section 4.2. Next we compute σ_{sv} for each vertex v by traversing the vertices in order of increasing shortest path distance $d(s,v)$, and for a vertex v summing σ_{sw} for each predecessor w of v (edge wv) in the DAG

[1]The algorithm as presented by Kumar and Schwabe only works under the assumption that no two vertices have the same shortest-path distance to s. However, this assumption can be removed by carefully managing the elements in the priority queues to make sure that such vertices are handled at the same time [116].

(Lemma 4.1). We do so I/O-efficiently using the *time-forward processing* technique [46]: Initially we sort the edges by the shortest-path distance of their first vertex (and secondarily by second vertex) and insert s annotated with $\sigma_{ss} = 0$ in an external priority queue [12] with priority $d(s, s) = 0$. Then we visit the vertices in increasing shortest-path order, and repeatedly obtain σ_{sw} for each predecessor w of vertex v by performing DELETEMIN operations on the priority queue, computing σ_{sv}, and then accessing all edges vu in the sorted list of edges and inserting each successor u of v in the priority queue, annotated with σ_{sv}. Overall the algorithm will scan through the sorted list of edges and perform $2E$ INSERT and DELETEMIN operations on the priority queue. After having computed σ_{sv} for each vertex v, we can compute $\delta_s(v)$ for each vertex v in a similar way using Lemma 4.2 by processing the vertices in reverse order of shortest-path distance using time-forward processing. Overall the algorithm uses $O(\text{SORT}(E))$ I/Os to compute σ_{sv} and $\delta_s(v)$, since it performs a constant number of scans and sort steps along with $2E$ priority queue operations, which can be performed in $\text{SORT}(E)$ I/Os [12].

Now to compute the betweenness centrality of each vertex v we simply run the above algorithm with each vertex as source s, and compute $C_B(v)$ by maintaining the partial sum of all $\delta_s(v)$, for all $v \in V$ as in the unweighted case in Section 4.2. Overall we use $O(V \cdot (V + \frac{E}{B} \log \frac{E}{B} + \text{SORT}(E))) = O(V^2 + V \frac{E}{B} \log \frac{E}{B})$ I/Os and $O(E)$ space.

Theorem 4.2 *Betweenness centrality for an undirected graph $G = (V, E)$ with positive edge weights can be computed in $O(V^2 + V \frac{E}{B} \log \frac{E}{B})$ I/Os and $O(E)$ space.*

4.3.2 The algorithm for sparse weighted graphs

The I/O bound in Theorem 4.2 is dominated by the $O(V^2)$ term in case the graph is sparse (when $E < VB/\log V$). The term is a result of each of the V individual shortest-path computations making V accesses to the adjacency lists. In this section we describe an algorithm that reduces the I/O cost when $E < VB/\log V$ at the expense of using more space. More precisely, we show how to compute betweenness centrality in $O(V \cdot (\sqrt{VE/B} + (E \log V)/B))$ I/Os using $O(V \cdot \sqrt{VE/B})$ space.

Our algorithm builds on the all-pairs shortest-paths algorithms of Chowdhury and Ramachandran [47] and of Arge et al. [17]. The idea is to run multiple instances of Kumar and Schwabe's version of Dijkstra's algorithm simultaneously to allow for faster access to the adjacency lists. However, in this case we may not have enough space in main memory for the $\Theta(B)$ space of each of the external tournament trees. Therefore, we below first describe how to modify the external tournament tree to only use $O(B/L)$ space in main memory for a parameter $L < B$. We then run multiple shortest-path algorithms simultaneously, using only L instances of the modified tournament tree at the same time, so we save $O(L)$ accesses to load the single priority queues.

Priority queue buffers. We now show how an $O(B/L)$-sized buffer in internal memory can help reduce the I/O cost for a sequence of operations on an external tournament tree. Essentially the same idea was used for the slim buffer heap of Chowdhury and Ramachandran [47], which achieves the same I/O bound, but the slim buffer heap is based on the buffer heap instead of an external tournament tree, unnecessarily complicating understanding and possible implementation of the algorithm.

Let Q be an external tournament tree as in Lemma 4.4, without using any space in internal memory. We store a *tournament tree buffer* for Q containing a set $s(Q)$ of the $|s(Q)| = O(B/L)$ smallest items in Q as well as an operation buffer of size $O(B/L)$. For any operation on Q we first try to perform it on the buffer of Q, which we assume to be loaded in memory (while the root of Q is still on disk). For a DeleteMin operation, we extract the minimum item from $s(Q)$ and return it. In case we run out of items, we load the root of Q and perform $O(B/L)$ DeleteMin operations on Q to refill $s(Q)$. For a DecreaseKey operation we first check whether the item occurs in $s(Q)$ and update it, otherwise we store the operation in the operation buffer. In case the operation buffer is full, we load the root of Q in memory and perform all $O(B/L)$ buffered operations on Q. A Delete operation is handled in the same way.

Lemma 4.5 *The external tournament tree can be implemented to use $O(B/L)$ internal memory, for $L < B$, such that a sequence of k operations can be processed in $O(k \cdot \frac{L}{B} + \frac{k}{B} \log \frac{N}{B})$ I/Os in total.*

Algorithm. Unlike the previous all-pairs shortest-paths algorithms [17, 47] that run all V instances of the Kumar and Schwabe algorithm [87] simultaneously, we only run K instances simultaneously. We divide these instances into K/L groups using $2 \cdot L$ slim buffer heaps each. Note that this allows us to write the $L \cdot O(\frac{B}{L})$ internal memory used by the L priority queues to disk, and read it again from disk, in a constant number of I/Os. We now perform the V rounds (each processing one vertex) of the Kumar and Schwabe algorithm one round at a time simultaneously for all K instances. A round starts by loading the priority queue blocks for each of the K/L groups in order, and for each group use the priority queues to determine the next vertex to be processed in the round for L of the instances. This results in a list $\mathcal{V}$ with a node to be processed for each of the K instances. To find the adjacency lists of all the vertices in $\mathcal{V}$, we then sort $\mathcal{V}$ by vertex and scan it simultaneously with the adjacency lists (also sorted by vertex). This produces a list $\mathcal{A}$ of the relevant neighbour vertices for all K instances. We then sort the vertices in $\mathcal{A}$ according to the group and instance they belong to. We end the round by scanning $\mathcal{A}$ while loading the priority queue blocks for each of the K/L groups in order, and update the priority queues using the information in $\mathcal{A}$. After finishing a set of K instances, we compute the number of shortest paths σ_{sv} and the dependency values $\delta_s(v)$ for these instances as in Section 4.3.1, that is, we build for each instance the shortest-path DAG from its source s and use time-forward processing on this

DAG to first compute σ_{sv} for all $v \in V$, and then $\delta_s(v)$ for all $v \in V$ in a reverse time-forward processing step.

Analysis. To run all V instances of the Kumar and Schwabe algorithm we perform $\frac{V}{K} \cdot V = \frac{V^2}{K}$ rounds in total. In each of these rounds we use $O(K/L)$ I/Os to read and write priority queue blocks. Sorting and scanning all adjacency lists takes $O(\text{SORT}(K)+E/B)$ I/Os per round. Sorting $\mathcal{A}$ takes $O(V \cdot \text{SORT}(E))$ I/Os in total for all instances, as each edge is used at most twice per instance. We also perform $O(E)$ operations on each tournament tree using $O(VE\frac{L}{B} + V\frac{E}{B}\log\frac{V}{B})$ I/Os (Lemma 4.5). Finally, we use $O(\text{SORT}(E))$ I/Os per instance for computing σ_{sv} and $\delta_s(v)$, while computing the partial sums for $C_B(v)$ at no extra cost. In total we use

$$O\left(\frac{V^2}{L} + \frac{V^2}{K} \cdot \left(\text{SORT}(K) + \frac{E}{B}\right) + V \cdot \text{SORT}(E) + VE \cdot \frac{L}{B} + V \cdot \frac{E}{B}\log\frac{V}{B}\right) =$$

$$O\left(\frac{V^2}{L} + \frac{VEL}{B} + \frac{V^2E}{KB} + \frac{VE}{B}\log\frac{V}{B}\right)$$

I/Os, where $1 \leq L \leq B$, and $L \leq K$. As we need to maintain $2 \cdot K$ tournament trees, we use $O(K \cdot V + E)$ space.

We balance the number of I/Os in the first two terms in the I/O bound above by setting $L = \sqrt{VB/E}$. We can now adjust K to trade space for I/Os. To optimize for I/Os, we set $K = \sqrt{VE/B} \geq L$, in which our algorithm uses $O(V \cdot (\sqrt{VE/B} + \frac{E}{B}\log\frac{V}{B}))$ I/Os and $O(V \cdot \sqrt{VE/B})$ space.

Theorem 4.3 *Betweenness centrality for an undirected graph $G = (V, E)$ with positive edge weights can be computed in $O(V \cdot (\sqrt{VE/B} + \frac{E}{B}\log\frac{V}{B}))$ I/Os using $O(V \cdot \sqrt{VE/B})$ space, given that $E < VB/\log V$.*

Remarks:

(i) For graphs with $E = O(V)$ the I/O improvement over the algorithm of Section 4.3.1 is a factor of $\sqrt{B}$.

(ii) By choosing a value for $K < \sqrt{VE/B}$, one can trade I/Os for space.

(iii) The algorithm can also compute the weighted diameter, improving the space of the best known such algorithm from $O(V^2)$ to $O(V \cdot \sqrt{VE/B})$.

4.3.3 The algorithm for low-diameter weighted graphs

In this section we describe an improved algorithm for weighted graphs with small (unweighted) diameter. Specifically, given a graph G with unweighted diameter $\text{diam}(G)$, our algorithm runs in $O(VE \cdot \text{diam}(G)/B)$ I/Os and uses $O(V^2)$ space. The basic idea is to simulate a distributed variant of the Bellman-Ford algorithm (as used in distance-vector routing algorithms), where each vertex maintains a complete set of distance estimates to all other nodes. The algorithm repeatedly updates these distance estimates of the vertices by comparing

them with those of neighbouring vertices. After all distances are found, we compute the shortest-path counts, dependency values, and betweenness centrality for all vertices. A key feature of our algorithm is that it avoids the repeated sorting and permutation steps required for a naive simulation. Below we first describe the basic distributed Bellman-Ford algorithm for computing all-pairs shortest paths. Then we describe our I/O-efficient variant of this algorithm and the extensions necessary for computing the shortest-path counts, dependency values, and betweenness centrality values for all vertices.

Distributed Bellman-Ford. Consider running the distributed Bellman-Ford algorithm on a physical network consisting of a node for each vertex of G and a connection between node u and node v in case there is an edge $uv \in E$. Each node v maintains a *distance vector* containing values $d_{\mathrm{t}}(v, w) < d(v, w)$ for all $w \in V$. Initially, $d_{\mathrm{t}}(u, v) = 0$ in case $u = v$, and $d_{\mathrm{t}}(u, v) = \infty$ otherwise. Then, the algorithm runs at most $\mathrm{diam}(G) + 1$ rounds in which the distance vector of each node v is sent to each of its neighbours in the network, after which $d'_{\mathrm{t}}(s, v) = \min_{u \in N(v)}\{d_{\mathrm{t}}(s, u) + w(u, v)\}$ for all $s \in V$ is computed, where $N(v)$ is the set of neighbours of v, and then sets $d_{\mathrm{t}}(s, v) = d'_{\mathrm{t}}(s, v)$ for all $s \in V$. Since any shortest path contains at most $\mathrm{diam}(G)$ edges, we are sure that $d_{\mathrm{t}}(s, t) = d(s, t)$ for all $s, t \in V$ after $\mathrm{diam}(G)$ rounds. Since no distances change in the next round, we can detect when to finish without knowing $\mathrm{diam}(G)$.

I/O-efficient shortest paths. For each vertex v we maintain the distance vector $(d_{\mathrm{t}}(v, v_1), \ldots, d_{\mathrm{t}}(v, v_{|V|}))$ as an ordered list in external memory. To simulate a round, instead of making all updates for one *vertex* at a time, which would require copying and sorting the distances, we make the updates involving one *edge* at a time. In more detail, for each round we first set $d'_{\mathrm{t}}(u, v) = \infty$ for all $u, v \in V$. Then, we scan the list of edges once, while for each edge $e = (u, v)$ scanning the distance vectors of u and v simultaneously. For each $1 \leq i \leq V$, we set $d'_{\mathrm{t}}(u, v_i) := \min(d'_{\mathrm{t}}(u, v_i), w(u, v) + d_{\mathrm{t}}(v, v_i))$ and $d'_{\mathrm{t}}(v, v_i) := \min(d'_{\mathrm{t}}(v, v_i), w(u, v) + d_{\mathrm{t}}(u, v_i))$. After finishing each round, we set $d_{\mathrm{t}}(s, t) := d'_{\mathrm{t}}(s, t)$ for all $s, t \in V$. We stop after the first round in which none of the distance estimates change.

Shortest-path counts. Next we compute σ_{st} for all $s, t \in V$. Again, we run $\mathrm{diam}(G)$ rounds, going through all edges once per round. For each $s \in V$, we store estimates of the number of shortest paths from s to all other vertices in a *path-count vector* (list) $(\sigma_{sv_1}, \ldots, \sigma_{sv_{|V|}})$. We maintain for each estimate σ_{st} whether it is (currently known to be) *final*. In each round we finalize the path counts σ_{st} for which all predecessors of t with respect to s had final path counts in the last round. In order to efficiently find out which path counts need to be finalized in a round, we also maintain for each σ_{st} whether it is known to be *incomputable* in the current round (because some predecessor has no final path count yet).

Initially we only set $\sigma_{ss} := 1$ for $s \in V$, and these are final. At the beginning of each round we reset each non-final σ_{st} to 0 and record that it is not incomputable. When processing an edge uv, we scan simultaneously over both the distance and path-count vectors of u and v. We check for each v_i whether uv is *tight* with respect to v_i, that is, whether $d(v, v_i) = d(u, v_i) + w(u, v)$. In case it is, and σ_{vv_i} is not final and σ_{uv_i} is final, we update $\sigma_{vv_i} := \sigma_{vv_i} + \sigma_{uv_i}$. In case σ_{uv_i} is not final we record that σ_{vv_i} is incomputable. If after completing a round we have not found a certain σ_{st} to be incomputable, we are in fact sure its current value is final and record this.

Dependency values. To compute $\delta_u(v)$ for all $u, v \in V$, we again proceed in $\operatorname{diam}(G)$ rounds, but instead of checking for tight edges to predecessors, we check for tight edges to successors with final dependency values. The dependency values for each vertex $v \in V$ are stored as a *dependency vector* (list) $(\delta_{v_1}(v), \ldots, \delta_{v_{|V|}}(v))$; initially $\delta_{v_i}(v) = 0$ for all i. When processing an edge uv, we scan the dependency vectors of u and v as well as their path-count and distance vectors, and update the dependency values of all v_i by setting $\delta_{v_i}(u) := \delta_{v_i}(u) + \sigma_{uv_i}/\sigma_{vv_i} \cdot (1 + \delta_{v_i}(v))$ in case uv is tight with respect to v_i, $\delta_{v_i}(u)$ is not final, and $\delta_{v_i}(v)$ is final.

Finally, we compute $C_B(v)$ for all $v \in V$ by summing all values in the dependency vector of v.

I/O complexity. For each of the three steps in the algorithm we scan the set of edges once per round, taking $O(\frac{E}{B})$ I/Os, and for each edge we scan at most six vectors simultaneously (the distance, path-count, and dependency vectors of both endpoints), taking $O(\frac{V}{B})$ I/Os per edge. It takes $O(\frac{V^2}{B})$ I/Os per round to finalize and reset the distance, path-count, and dependency values. Hence, in total we use $O(\operatorname{diam}(G) \cdot (\frac{E}{B} + E \cdot \frac{V}{B} + \frac{V^2}{B})) = O(VE/B \cdot \operatorname{diam}(G))$ I/Os. The algorithm uses $O(V^2)$ space because it needs to store the three lists of length V for each vertex.

Theorem 4.4 *Betweenness centrality for a directed graph $G = (V, E)$ with diameter* $\operatorname{diam}(G)$ *and positive edge weights can be computed in $O(VE/B \cdot \operatorname{diam}(G))$ I/Os using $O(V^2)$ space.*

Remarks:

(i) The algorithm uses a factor $O(B/\operatorname{diam}(G))$ fewer I/Os than the algorithm of Section 4.3.1 in case of graphs where $E = O(V)$. For graphs where $E \geq VB/\log V$, the improvement is only a factor $O(\log \frac{E}{B}/\operatorname{diam}(G))$.

(ii) As opposed to the other algorithms discussed in this chapter, the current algorithm can easily be made to work for directed graphs.

(iii) The algorithm is cache-oblivious, since it only scans over lists.

Chapter 5

Multiway Cycle Separators and I/O-Efficient Planar Graph Algorithms

As explained in Chapter 2, the existing I/O-efficient solutions to single-source shortest paths [14], topological sorting [23, 22], and computing strongly connected components [25] achieve an I/O complexity of $O(\text{SORT}(N))$, which is substantially less than N I/Os. However, the time they use in internal memory is $\Omega(BN)$ in the worst case which is comparable to that of the linear time internal-memory algorithms applied naively to large graphs. This negative impact of the internal-memory computation on the total running time of these algorithms has indeed been observed in several implementations of separator-based algorithms. In order to see *any* speed-up over naive algorithms, a much smaller block size had to be used than for other algorithms whose internal-memory computation is independent of the block size.

Our contribution

We present algorithms for computing single-source shortest paths in planar graphs with non-negative edge lengths, topological sorting of planar directed acyclic graphs (DAGs), and computing the strongly connected components of planar graphs using $O(\text{SORT}(N))$ I/Os and $O(N \log N)$ time. Thus, in contrast to previous algorithms for these problems, the amount of internal-memory computation performed by our algorithms is independent of the block size.

The high-level idea in our algorithms is simply to follow the framework of the existing algorithms and use Klein's multi-source shortest-path algorithm for planar graphs [82] to reduce the amount of internal-memory computation to $O(N \log N)$. For this to work, however, we need a separator partition of the graph that guarantees, in addition to the standard properties of planar graph partitions, a constant number of boundary cycles per region in the partition. No existing internal- or external-memory algorithm for computing separator partitions guarantees this. Our main technical contribution therefore is an algorithm for computing a multiway variant of Miller's simple cycle separators [96] in linear time in internal memory (Section 5.2) or using $O(\text{SORT}(N))$ I/Os and $O(N \log N)$ time in external memory (Section 5.3). These algorithms do not use RAM-specific operations and therefore also work on a pointer ma-

chine. Specifically, given a parameter ε, we show how to partition a graph with weights on its vertices, edges, and faces that sum to 1 and a bound s on the size (number of vertices on the boundary) of every face, into $O(1/\varepsilon)$ regions (collections of faces and their boundaries) such that each region has weight at most ε and $O(1)$ boundary cycles of size $O(\sqrt{\varepsilon s N})$.

The internal-memory algorithm for computing the above separators combines existing techniques for computing separators, such as separation trees [10], simple-cycle separators [96], and recursive partitioning of regions to reduce their boundary size [60], in novel and intricate ways. The external-memory algorithm employs the bootstrapping technique of [94] to obtain an I/O-efficient separator algorithm from the internal-memory algorithm. The basic idea of this bootstrapping approach is to apply graph contraction to obtain a constant factor smaller graph G' from the original graph G, recursively compute a separator partition $\mathcal{P}'$ of G', recover a suboptimal but "good enough" separator partition $\mathcal{P}''$ of G from $\mathcal{P}'$, use $\mathcal{P}''$ to perform BFS in G, and use the computed BFS tree to obtain an optimal partition $\mathcal{P}$ of G. In contrast to the separator algorithm of [94] (and similar to Miller's algorithm [96]), our separator algorithm requires a BFS tree of the face-incidence graph of G, which requires the bootstrapping framework to alternate between the current graph and its dual. This introduces a few complications.

Computing graph separators in internal memory

There exists a large body of work on computing planar separators, especially in internal memory. In order to put our results in context, we review some of that work here. Lipton and Tarjan [92] were the first to show that in linear time every planar graph G can be partitioning into two subgraphs of size at most $2N/3$ each and without edges between them by removing a set S of at most $O(\sqrt{N})$ vertices from G. Such a set S is called a *vertex separator*. With the motivation to speed up shortest-path computations on planar graphs, Frederickson [60] extended this result to obtain a *multiway separator*: given a parameter r, G can be partitioned into $O(N/r)$ subgraphs of size at most r and with boundary size $O(\sqrt{r})$, where the boundary size of a region is the number of vertices it shares with other regions. Frederickson's algorithm to compute such a partition takes $O(N \log N)$ time, by applying Lipton and Tarjan's algorithm recursively. Aleksandrov and Djidjev [10] showed that essentially the same partition can be computed in linear time. Their main technique is the use of a *separation tree*: given a spanning tree T of G, the graph T^* containing all edges dual to non-tree edges of G is a spanning tree of the dual of G, and an appropriate partition of T^* induces the desired partition of G. Miller [96] extended Lipton and Tarjan's result in a different direction by showing that, given that the maximum face size is s, it is in fact possible to compute a simple cycle C of length at most $O(\sqrt{sN})$ in G such that at most $2N/3$ vertices lie on either side of the cycle. C is called a *simple cycle separator* of G. To the best of our knowledge, no work has been done on multiway simple cycle separators, probably mostly because it is not clear what structural properties to require of such a separator: a single simple cycle cannot partition the graph into more than two regions, and if more than

one cycle is required/allowed, what are the structural properties one should ask the separator to have?

Our definition in this chapter is a natural one: Miller's simple cycle separator partitions the graph into two subgraphs with a boundary size equivalent to the one guaranteed by Lipton and Tarjan's algorithm, and each subgraph has exactly one boundary cycle. Our multiway separator partitions the graph into $O(N/r)$ subgraphs of size at most r and boundary size $O(\sqrt{sr})$, the equivalent of Frederickson's partition, and each region has $O(1)$ boundary cycles. As we show, this is also the right partition to guarantee that we can solve a number of problems on planar graphs efficiently in both internal and external memory at the same time. Previous results relied on Maheshwari and Zeh's Frederickson-style graph partition to compute single-source shortest paths, topological sorting, and strongly connected components using $O(\text{SORT}(N))$ I/Os but $O(N \log N + BN)$ time in internal memory. By guaranteeing a bound on the number of boundary cycles of each region, we can replace the naive implementation of the internal-memory computation of these algorithms with Klein's multi-source shortest-path algorithm (see Section 5.4) and thereby reduce the internal-memory computation to $O(N \log N)$.

5.1 Preliminaries

In this section, we introduce the necessary notation and terminology for embedded planar graphs and their duals as used in this chapter, including a formal definition of multiway cycle separators.

Basic definitions. For a graph G, we use $V(G)$ and $E(G)$ to denote its vertex and edge sets, respectively. If G is clear from the context, we simply write V and E for $V(G)$ and $E(G)$, respectively. A *planar embedding* $\mathcal{E}(G)$ of G is a drawing of G in the plane such that no two edges intersect, except in their endpoints. G is *planar* if it has a planar embedding. A planar graph G equipped with a given embedding $\mathcal{E}(G)$ is often referred to as a *plane graph*. We use $\mathcal{E}(v)$ and $\mathcal{E}(e)$ to denote the image of vertex v or edge e under this embedding, and $\mathcal{E}(S) = \bigcup_{x \in S} \mathcal{E}(x)$, for any set of vertices and edges of G. The *faces* of an embedding are the connected components of $\mathbb{R}^2 \setminus (\mathcal{E}(V(G)) \cup \mathcal{E}(E(G)))$. We denote the set of faces of G by $F(G)$. The *boundary* ∂f of a face f is the set of vertices and edges contained in f's closure. The *size* of a face is the number of edges on its boundary. A *region* is defined as a collection R of faces and their boundaries, excluding all vertices and edges that are on the boundary of at least one face not in R. The latter constitute the boundary ∂R of R. We use $\bar{R} := R \cup \partial R$ to denote the closure of a region. We call a region R connected if it is connected in the topological sense. Given a weight function w assigning weights to the vertices, edges, and faces of a plane graph G, the weight of a region or subgraph of G is the sum of the weights of the vertices, edges, and faces it contains. A *simple cycle* C is a collection of vertices and edges of G such that $\mathcal{E}(C)$ is homeomorphic to S^1 (a closed loop). $\mathbb{R}^2 \setminus \mathcal{E}(C)$ has two connected components, one bounded and one unbounded. We call the former the *interior*

of C and the latter the *exterior* of C. The *region enclosed by a cycle* C is the collection of vertices, edges, and faces embedded in C's interior. The *weight enclosed by* C is the weight of the region enclosed by C. The boundary ∂R of a region R is the union of a set of simple cycles, which we call *boundary cycles* of R. For exactly one of these cycles $C \subseteq \partial R$, R is in C's interior. We call this cycle the *outer boundary* of R and the other boundary cycles of R *holes*.

Dual-related structures. The *dual* G^* of a plane graph G contains one vertex f^* for each face f of G. For every edge e of G on the boundary of faces f_1 and f_2, G^* contains an edge e^* with endpoints f_1^* and f_2^*. We call f^* the dual of face f and e^* the dual of edge e. G^* is planar, and an embedding of G^* is obtained naturally from the given embedding of G. Given a spanning tree T of G, the graph T^* containing every edge of G^* whose dual does not belong to T is a spanning tree of G^*. We call T^* the *dual* of T. Every non-tree edge e of G w.r.t. T defines a *fundamental cycle* $\mathrm{fc}(e)$, which consists of e and the path between e's endpoints in T. If T^* is rooted in the vertex representing G's outer face, then the descendant nodes of e's dual edge e^* are the dual vertices of the faces in the region enclosed by $\mathrm{fc}(e)$. The *face incidence graph* G_{f} of G is closely related to the dual. It has one vertex f^* per face f of G and an edge $f_1^* f_2^*$ if and only if the boundaries of faces f_1 and f_2 are non-disjoint. The *vertex-on-face graph* G_{vf} contains every vertex of G and one vertex f^* per face f of G and contains an edge vf^* if vertex v is on the boundary of face f.

Simple cycle ε-separators. Given a parameter $0 < \varepsilon < 1$ and a function that assigns weights to the vertices, edges, and faces of G such that the total weight of G is 1 and any face has weight at most ε, a *simple cycle ε-separator* of a 2-edge-connected planar graph G is a 2-edge-connected subgraph S of G together with a grouping of the faces of G into $O(1/\varepsilon)$ regions with the following properties:

(i) all faces of G embedded inside the same face of S belong to the same region;

(ii) the weight of each region is $O(\varepsilon)$;

(iii) the boundary of each region has size $O(\sqrt{\varepsilon s N})$;

(iv) every region is either connected or shares boundary edges with at most two other regions, which must be connected; and

(v) the boundary of each connected portion of each region consists of $O(1)$ simple cycles.

We use $G - S$ to denote the set of connected regions into which S divides G. The *boundary sets* of S are the maximal sets of boundary vertices incident to the same regions.

In general, we use *multiway (simple) cycle separator* to refer to a simple cycle ε-separator in case ε is clear from the context or unspecified.

5.2 Computing multiway simple cycle separators in linear time

This section presents our main result in this chapter: a linear time algorithm for computing multiway simple cycle separators, as summarized in the following theorem.

Theorem 5.1 *A simple cycle ε-separator can be computed in linear time.*

Our algorithm to compute such a separator follows the standard framework used in almost all planar separator algorithms starting with Lipton and Tarjan's seminal result, but it has a few non-trivial twists. We proceed in three steps.

Step 1 computes a BFS tree and chooses appropriate levels of the tree as the first part S_1 of the separator. Just like Miller's algorithm, we use BFS in the face-incidence graph of G instead of G and then choose S_1 to consist of the boundaries between faces on appropriate pairs of consecutive levels. S_1 partitions G into two types of regions: *light* ones of weight at most ε and containing at most εN vertices, and *heavy* ones of weight greater than ε or with more than εN vertices but of small diameter. There are two complications here. As in Miller's algorithm, we cannot actually guarantee low diameter of the heavy regions. We can, however, identify a low-diameter subgraph R' of each heavy region R such that each face of R' contains a subgraph of R of weight at most ε, as well as construct a low-diameter spanning tree of R'. A multiway simple cycle separator of R' is also a multiway simple cycle separator of R, so Step 2 focuses on partitioning each such subgraph R' into subregions of weight at most ε and with at most εN vertices. The second complication is that S_1 may consist of too many simple cycles. We remove simple cycles between adjacent light regions as appropriate to reduce the number of light regions and simple cycles to $O(1/\varepsilon)$ while keeping the resulting regions light.

Step 2 partitions each heavy region R into light subregions and is divided into two substeps. Step 2.1 chooses a number of fundamental cycles of the spanning tree T of R' to partition R. We cannot yet guarantee that the resulting subregions are light because this may require too many fundamental cycles. (Aleksandrov and Djidjev's application of separation trees used the fact that T^* has degree three if G is a triangulation, in which case a greedy partition of T^* into subgraphs of weight at most ε induces a collection of $O(1/\varepsilon)$ fundamental cycles partitioning R into subregions of weight at most ε. We cannot use this strategy because T^* does not have constant degree in our algorithm.) What Step 2.1 produces is a partition of T^* into $O(1/\varepsilon)$ subtrees T_i^* (thus corresponding to $O(1/\varepsilon)$ fundamental cycles) such that the subtrees of T_i^* rooted in children of the root each have weight at most ε and represent regions with at most εN vertices. Each such subtree of T_i^* spans a subregion of weight at most ε of the region R_i' spanned by T_i^*, and each such subregion shares at least one edge with the face f corresponding to the root of T_i^*. Step 2.2 computes a "nesting tree", which captures the attachment of these subregions to f and then applies a procedure very similar to Step 2.1 to partition this nesting tree into subtrees of weight at most ε. Once again, we cannot guarantee that this pro-

duces only $O(1/\varepsilon)$ regions, but this time the regions can be grouped naturally to reduce their number to $O(1/\varepsilon)$. This is what Step 3 does.

The procedure outlined so far produces a partition into the right number of regions of the right size, with the right weight, and with the right *total* boundary size. The boundary of an individual region, however, may be too large and may consist of too many simple cycles. Step 3 therefore subdivides regions further to ensure each individual region has a small boundary consisting of only $O(1)$ simple cycles. We do this in a manner very much like the final step in Frederickson's algorithm, but using Miller's simple cycle separator algorithm instead of Lipton and Tarjan's algorithm. The final step is to group connected regions in a manner very similar to Frederickson's algorithm to reduce the number of regions to $O(1/\varepsilon)$.

5.2.1 Step 1: Partition into small or low-diameter regions

In this section, we compute the first part S_1 of the separator, which partitions the graph into few regions that are either light or of low *weighted diameter*, as summarized in the following lemma. Given the current separator, we define the weighted diameter of a graph to be the maximum number of non-separator edges of any path in the graph.

Lemma 5.1 *Given a parameter ε, $0 < \varepsilon < 1$, and a 2-edge-connected plane graph G with maximum face size s and maximum face weight at most ε, it takes linear time to compute (i) a 2-edge-connected subgraph G' of G whose faces have weight at most ε and boundary size at most $\max(s, \sqrt{\varepsilon s N})$ each and (ii) a set $S_1 \subseteq E(G')$ consisting of $O(1/\varepsilon)$ edge disjoint simple cycles that partition G' into $O(1/\varepsilon)$ connected regions, each either of weight at most ε and containing at most εN vertices or of weighted diameter at most $O(\sqrt{\varepsilon s N})$. The total length of the cycles in S_1 is $O(\sqrt{sN/\varepsilon})$.*

As already explained, we implement this step in two substeps. First we find a superset $S_{1,1} \supseteq S_1$ of size $O(\sqrt{sN/\varepsilon})$ that splits G into regions with weighted diameter $O(\sqrt{\varepsilon s N})$. Then we remove all but $O(1/\varepsilon)$ cycles from $S_{1,1}$, which may increase the weighted diameter of some regions beyond $O(\sqrt{\varepsilon s N})$. If the region is light, this is no problem. For a heavy region, we merge some of its faces to obtain G' and ensure that the portion of G' contained in each region has weighted diameter $O(\sqrt{\varepsilon s N})$.

Step 1.1: Partition into low-diameter regions

To compute $S_{1,1}$, we construct a breadth-first search (BFS) tree T_{f} of the face incidence graph G_{f} of G, starting from an arbitrary root face. To compute this BFS tree in linear time (and I/O-efficiently in Section 5.3), we construct the vertex-on-face graph G_{vf} of G, and compute a BFS tree T_{vf} of G_{vf} rooted in an arbitrary face vertex of G_{vf}. T_{f} can be obtained from T_{vf} by making every face vertex a child of its grandparent in T_{vf} and then deleting all vertices of G from T_{vf}. Let L_i be the vertices of T_{f} at distance i from the root, and let $\mathcal{F}_i$ be the set of vertices and edges that are each incident to a face in L_i and a face in L_{i+1}

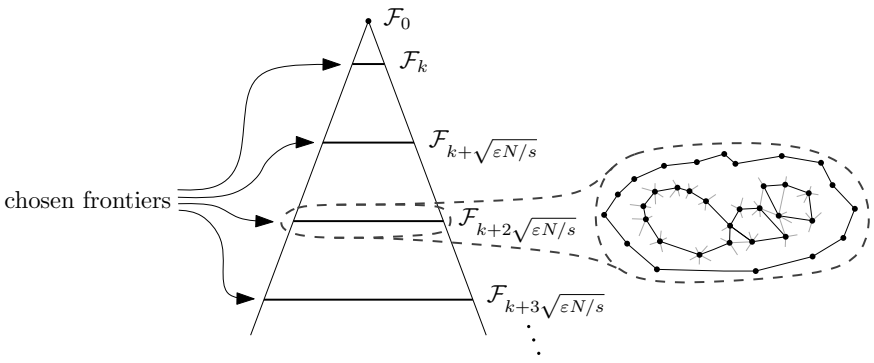

Figure 5.1: BFS and frontiers in Step 1.1.

(see Figure 5.1). $\mathcal{F}_i$ forms a set of edge-disjoint simple cycles that separate the faces in $L_0, L_1, \ldots, L_i$ from the faces in $L_{i+1}, L_{i+2}, \ldots$. We call $\mathcal{F}_i$ the *level-i frontier*.

Each vertex or edge belongs to at most one frontier, so by the pigeon hole principle we can choose $0 \leq k < \sqrt{\varepsilon N/s}$ such that there are at most $N/\sqrt{\varepsilon N/s} = \sqrt{sN/\varepsilon}$ edges in $S_{1,1} := \mathcal{F}_k \cup \mathcal{F}_{k+\sqrt{\varepsilon N/s}} \cup \mathcal{F}_{k+2\sqrt{\varepsilon N/s}} \cup \mathcal{F}_{k+3\sqrt{\varepsilon N/s}} \cup \cdots$. These frontiers can clearly also be selected in linear time. It is not hard to prove that each region of $G - S_{1,1}$ has weighted diameter at most $\sqrt{\varepsilon s N}$. We do not prove this here because what we really require is that every region of $G' - S_1$ has this diameter. We construct G' and S_1 in the next step.

Step 1.2: Reducing the number of boundary cycles

The reduction of the number of boundary cycles is achieved in two steps. We use a *nesting tree* T of the regions of $G - S_{1,1}$ (faces of $S_{1,1}$). In the first step, we ensure this tree has only heavy leaves by merging leaves into their parents. This step also modifies G to obtain the subgraph $G' \subseteq G$ whose regions have low weighted diameter. This ensures that there are only $O(1/\varepsilon)$ regions corresponding to leaves or heavy regions and, hence, only $O(1/\varepsilon)$ regions corresponding to internal nodes with at least two children. The second step merges pairs of adjacent light regions corresponding to internal nodes with only one child, in order to reduce their number to $O(1/\varepsilon)$ while keeping their weight and size bounded by ε and εN, respectively.

Computing a nesting tree with few leaves. To compute the nesting tree, we compute the dual G^* of G and remove all edges dual to edges in $S_{1,1}$. The regions of $G - S_{1,1}$ correspond to the connected components of the resulting subgraph of G^*, and we create one vertex in T for each region. Then we add one edge to T for every pair of regions that share a boundary edge, which ensures that T is a tree since $S_{1,1}$ is a set of edge-disjoint simple cycles. We root T in the node representing the region containing the outer face of G. Next we assign weights to the vertices of T that capture the weights of their corresponding regions. In particular, the weight of every face f of G is assigned to the node of T representing the connected component of G^* that contains f^*. For every

vertex or edge x of G, we choose an arbitrary incident face f and add x's weight to the node of T that received f's weight. We also define a *region size* of each node x of T, which is the number of vertices of G that assigned their weights to x. Note that this ensures that the total weight and the total region size of the nodes in any subtree $T' \subseteq T$ are upper bounds on the weight and the number of vertices in the region of G represented by T'. Every edge of T represents a simple cycle in $S_{1,1}$ that separates the regions corresponding to the subtrees on either side of the edge. We call an edge of T *heavy* if the subtree below it has weight greater than ε or region size greater than εN. Otherwise the edge is *light*. First we traverse T and identify all light edges whose parent edges are heavy. We call these edges *critical edges* and their corresponding cycles in $S_{1,1}$ *critical cycles*. We remove all critical edges and their descendant edges from T, and we remove the cycles corresponding to the removed edges from $S_{1,1}$, thus obtaining a reduced separator $S_{1,2} \subseteq S_{1,1}$. This ensures that every leaf of T represents a heavy region of $G - S_{1,2}$, but the weighted diameters of heavy regions may be large. Next we show how to reduce the diameter of heavy regions to $O(\sqrt{\varepsilon s N})$ again while keeping the face weight and face size bounded by ε and $\max(s, \sqrt{\varepsilon s N})$, respectively.

Constructing G' from G. Consider a heavy region R of $G - S_{1,2}$, and let R'' be the subgraph of R obtained by removing all vertices and edges enclosed by a critical cycle in R. Each face of R'' is either a face of G or a critical cycle. Let C be such a critical cycle, and let R_C be the region of R enclosed by C. Since R_C has weight at most ε, we can arbitrarily merge faces in R_C without creating a face of weight greater than ε. Recall that C is part of a frontier $\mathcal{F}_i$. We consider the portions of frontiers $\mathcal{F}_i, \mathcal{F}_{i+1}, \ldots, \mathcal{F}_{i+\sqrt{\varepsilon N/s}-1}$ that belong to R_C. By the pigeon hole principle and because R_C has at most εN vertices, there exists one frontier $\mathcal{F}_j$ among them that contributes at most $\sqrt{\varepsilon s N}$ vertices to R_C. We remove all vertices and edges of R_C enclosed by cycles of $\mathcal{F}_j$ contained in R_C to obtain a subgraph $R'_C \subseteq R_C$. We call the resulting faces of R'_C *pruning faces*. By performing this transformation on each portion R_C of R enclosed by a critical cycle C in R, we obtain a *pruned subgraph* $R' \subseteq R$. We obtain G' by replacing each heavy region R of $G - S_{1,2}$ with its pruned subgraph.

Lemma 5.2 *For every heavy region R of $G - S_{1,2}$, the pruned subgraph $R' \subseteq R$ has weighted diameter at most $4\sqrt{\varepsilon s N}$. Every face of R' has boundary size at most $\max(s, \sqrt{\varepsilon s N})$ and weight at most ε.*

Proof. We already argued that every face of R' has weight at most ε because it is either a face of G or is enclosed in a critical cycle. If it is a face of G, its size is at most s. Otherwise it is a pruning face and, thus, has size at most $\sqrt{\varepsilon s N}$. To bound the diameter of R', it suffices to show that every vertex of R' has a path of length at most $2\sqrt{\varepsilon s N}$ to the outer boundary of R' because the outer boundary of R' does not count towards the diameter (it is part of $S_{1,2}$). Let the outer boundary of R' be part of a frontier $\mathcal{F}_i$. Every vertex of R' is on the boundary of a face f of R' that is a face of G and such that f^* belongs to some level L_j of T_{f} with $j - i \leq 2\sqrt{\varepsilon N/s}$. Let $(f')^*$ be f^*'s ancestor in L_i. Then

$\text{dist}_{R'}(u, v) \leq s \cdot (j - i) \leq 2\sqrt{\varepsilon s N}$, for every vertex u on f and every vertex v on f', because the dual of every vertex on the path in T_{f} from f^* to $(f')^*$ is a face of R' and has size at most s. One of the vertices of f' must belong to $\mathcal{F}_i$ and, hence, to the outer boundary of R'. Thus, every vertex of f has distance at most $2\sqrt{\varepsilon s N}$ from the outer boundary of R'. □

Even though we will further reduce $S_{1,2}$ to obtain a separator $S_1 \subseteq S_{1,2}$, this reduction will only affect light regions. Thus, Lemma 5.2 shows that the heavy regions of $G' - S_1$ have the required structure. For Step 2 it is important that we can find a particular spanning tree of each heavy region of $G' - S_1$, as stated in the following lemma.

Lemma 5.3 *Every heavy region R' of $G' - S_{1,2}$ has a spanning tree T' of weighted diameter at most $4\sqrt{\varepsilon s N}$ and such that every boundary cycle of R' contributes all but one of its edges to T'.*

Proof. As shown in the proof of Lemma 5.2, every vertex of R' has distance at most $2\sqrt{\varepsilon s N}$ from the outer boundary of R' (or from the frontier $\mathcal{F}_0$ if R' is the region including the outer face of G'). Thus, a shortest-path tree of R' rooted in a vertex on this outer boundary (or on $\mathcal{F}_0$) has weighted diameter at most $4\sqrt{\varepsilon s N}$. Such a tree T'' can be computed in linear time using a straightforward adaptation of standard BFS because the edges have weight 0 or 1.

All the edges of T'' are directed away from the root. For every boundary cycle C of R', we remove all edges with head in C from T'', except one edge whose tail does not belong to C and whose head has the smallest distance from the root of T'' among these edges. If C is the outer boundary of R', we remove all edges with head in C. Then we add all edges of C, except one, to T''. It is easily verified that the resulting graph T' is a tree, has diameter no greater than that of T'', and includes all edges of each boundary cycle, except one. □

Merging light regions. The final step is to merge light regions to reduce their number to $O(1/\varepsilon)$. We consider maximal paths in T such that all nodes on such a path P represent light regions and have only one child each. We traverse each such path from one end and find the longest prefix P' such that the total weight of the nodes in P' is no more than ε and the total vertex count of these nodes is no more than εN. We remove all cycles corresponding to edges in P' from $S_{1,2}$ and then proceed to the first node in P after P'. Starting with this node, we once again find a prefix of the remainder of P as above and remove the cycles corresponding to its edges from $S_{1,2}$. We continue until we have consumed all of P in this fashion. The cycles not removed from $S_{1,2}$ once we are done processing all such paths P in T in this manner constitute the final separator S_1 produced by Step 1 of our algorithm.

Clearly, this reduction of $S_{1,2}$ to S_1 does not affect heavy regions, and we explicitly ensure that the regions produced by merging light regions are themselves light. Thus, all regions of $G' - S_1$ have the properties stated in Lemma 5.1. To bound the number of regions, consider the tree T' obtained from T by replacing each subpath P' in the above procedure with a single node. There are at most

$4/\varepsilon$ nodes in T' that are of weight greater than $\varepsilon/2$ or have vertex count greater than $\varepsilon N/2$. Since every leaf of T' is heavy and the number of internal nodes with at least two children is bounded by the number of leaves, there are at most $8/\varepsilon$ nodes of weight greater than $\varepsilon/2$, vertex count greater than $\varepsilon N/2$ or with at least two children. We call these nodes *semi-heavy* and all other nodes *semi-light*. Every semi-light node v must be the parent of a semi-heavy node because otherwise the processing of paths of degree-1 nodes above would have merged v and its child. Thus, there are at most $8/\varepsilon$ semi-light nodes, and the total number of regions of $G' - S_1$ is at most $16/\varepsilon$.

5.2.2 Step 2: Splitting heavy regions

The second step of our algorithm produces a separator S_2 that partitions G into regions of size at most εN and weight at most ε, as stated in the following lemma.

Lemma 5.4 *Given a parameter ε, $0 < \varepsilon < 1$, and a 2-edge-connected plane graph G with maximum face size s and maximum face weight at most ε, it takes linear time to compute a subgraph $S_2 \subseteq G$ such that every region of $G - S_2$ has at most εN vertices and weight at most ε, S_2 has $O(\sqrt{sN/\varepsilon})$ edges, and every connected component of S_2 is 2-edge-connected. The total number of holes of the regions of $G - S_2$ is $O(1/\varepsilon)$.*

We obtain S_2 by augmenting the separator S_1 produced by Step 1. Furthermore, it is sufficient to obtain a separator S_2 that partitions G' into regions because $G' \subseteq G$ and every face of G' has weight equal to the total weight of the portion of G embedded in it. The light regions of $G - S_1$ do not need to be partitioned further. We partition every heavy region R of $G - S_1$ into light subregions in two steps. In Step 2.1, we use the *separation tree technique* by Aleksandrov and Djidjev [10] to partition R into $O(1/\varepsilon)$ parts composed of subregions of weight at most ε and size at most εN that each share at least one edge with a "root face" f. This partition is obtained by adding $O(1/\varepsilon)$ fundamental cycles w.r.t. the low-diameter spanning tree produced in Step 1 to the separator. In Step 2.2, we use a *nesting forest* that captures how certain subregions combined with f enclose other subregions, in order to obtain a partition in which all regions are small. As already stated in the introduction of this section, the reason why we need to follow this two-step approach is that the separation tree technique can be used to produce a partition into small regions using few fundamental cycles only if the dual spanning tree has constant degree. Since the faces of G' do not necessarily have constant size, we cannot guarantee this here.

Step 2.1: Separation tree decomposition

Recall that every heavy region R of $G' - S_1$ has a low-diameter spanning tree T. Let T^* be the dual spanning tree of T. Note that T^* also includes vertices corresponding to faces bounded by the boundary cycles of R and which as such do not belong to R. We assume that these faces and the boundary vertices and

edges of R have weight 0 so that R viewed as an embedded planar graph has the same weight as R viewed as a region, that is, as a collection of faces of G' and their boundaries. We assume that T^* is rooted in the vertex corresponding to the outer boundary of R. Slightly extending the separation tree technique of Aleksandrov and Djidjev [10], we assign the weights of the vertices, edges, and faces of G' to the vertices of T^* as follows: The weight of every face f is assigned to f^*. For every vertex or edge x, we choose a face f that has x on its boundary and add x's weight to the weight of f^*. This guarantees that the weight of a subtree T' of T^* is an upper bound on the weight of the region consisting of the faces of G' dual to the vertices in T'. As in Step 1, we assign a *vertex count* to each node f^* of T^*, denoted $v(f^*)$, which is the number of vertices of G that assigned their weight to f^*. Now we identify a set C of at most $w(T^*)/\varepsilon - 1$ edges of T^* whose removal partitions T^* into at most $(w(T^*) + v(T^*)/N)/\varepsilon = (w(R) + |R|/N)/\varepsilon$ subtrees $T_1^*, T_2^*, \ldots, T_k^*$ such that, for all $1 \leq i \leq k$, any subtree of T_i^* that does not include the root of T_i^* has weight at most ε and total vertex count at most εN. Every edge $e \in C$ induces a fundamental cycle $\mathrm{fc}(e)$ in T, and the set of fundamental cycles $\mathrm{fc}(C) = \{\mathrm{fc}(e) \mid e \in C\}$ partitions R into connected subregions corresponding to the trees $T_1^*, T_2^*, \ldots, T_k^*$. Since the total weight of G' is 1 and G has N vertices, the total number of fundamental cycles introduced in all heavy regions of $G' - S_1$ is $O(1/\varepsilon)$. By Lemma 5.3, adding these fundamental cycles to S_1 therefore produces a separator $S_{2,1} \supseteq S_1$ of size $|S_1| + O(\sqrt{\varepsilon s N}/\varepsilon) = O(\sqrt{sN/\varepsilon})$.

To compute the set C, we process T^* bottom-up (e.g., using a postorder traversal) and compute for each node f^* of T^* the total weight and vertex count of all its descendant nodes. When visiting a non-root node f^*, if the total weight of the descendants of f^* exceeds ε or their total vertex count exceeds εN, we add the parent edge of f^* to C. This removes the subtree below f^* from T^*, and the nodes in this subtree do not contribute to the total weight or vertex count of any ancestor of f^*. This is essentially the same approach chosen by Aleksandrov and Djidjev, but, as we already argued, it does not guarantee a bound on the weight or size of any region yet because T^* does not necessarily have bounded degree. The bound on the number of edges added to C is easily seen as claimed because every edge added to C can be charged to a subtree of weight greater than ε or vertex count greater than εN that is pruned from T^* by adding this edge to C.

To identify the edges in $\mathrm{fc}(C) \setminus S_1$ that need to be added to $S_{2,1}$, we label every node in T^* with the index i of the subtree T_i^* it belongs to. An edge e needs to be added to $S_{2,1}$ if it is not already in S_1 and the two endpoints of e^* in R^* have different labels.

For each subregion R_i of R corresponding to a tree T_i^*, the root face is the one corresponding to the root r^* of T_i^*. For each child f^* of r^*, the portion of R_i corresponding to the descendants of f^* in T_i^* by definition has weight at most ε and size at most εN. r^* is a face of G' and, thus, also has weight at most ε. Finally observe that each such portion of R_i is connected because its dual is spanned by a subtree of T_i^* and all nodes in T^* not corresponding to faces of R must be leaves, by Lemma 5.3. Thus, $S_{2,1}$ defines the partition we set out to compute in Step 2.1.

Step 2.2: Nesting forest decomposition

Now consider the partition of G' defined by $S_{2,1}$, and let R' be one of its regions. If R' is light, there is no need to partition it further. So assume R' is heavy. In this case, it is one of the subregions produced in Step 2.1. Let $(T')^*$ be the rooted spanning tree of $(R')^*$, and let r^* be the root of $(T')^*$. Let R'' be the subgraph of R' obtained by replacing the faces dual to the nodes in each subtree of $(T')^*$ rooted in a child of r^* with a single face (see Figure 5.2(a)). By letting the weight and vertex count of each face of R'' be the weight and vertex count of its corresponding portion of $(T')^*$, we ensure that the weight and vertex count of each such face upper bounds the weight and number of vertices in the portion of G embedded inside this face, and no face of R'' has weight greater than ε nor vertex count greater than εN. Thus, a separator partitioning R'' into subregions of weight at most ε and vertex count at most εN also partitions the portion of G embedded inside R' into regions with these properties.

To compute such a partition of R'' that does not add too many edges to the final separator, we construct a *nesting forest* $\mathcal{T}$ of R''. The vertex set of $\mathcal{T}$ includes one vertex f^* per face $f \neq r$ of R''. To construct the edge set of $\mathcal{T}$, we number the edges on the boundary of the root face r clockwise around r, starting with an arbitrary edge. Every face $f \neq r$ of R'' shares at least one edge with r: the dual edge of the edge connecting its corresponding subtree of T^* to r^*. By considering the first and last edge in the ordering of the edges around r shared between r and f, we obtain an edge interval I_f associated with f. Node f_1^* is the parent of node f_2^* in $\mathcal{T}$ if $I_{f_2} \subseteq I_{f_1}$ and the boundaries of f_1 and f_2 share an edge. Given $\mathcal{T}$, we assign weights and vertex counts to its nodes in a manner analogous to Step 2.1, and we compute a set C of edges of $\mathcal{T}$ whose removal partitions $\mathcal{T}$ into subtrees of weight at most ε and vertex count at most εN. We do this similarly to the computation of the edge set C in Step 2.1, with the following modification: when we encounter a node f^* whose subtree of $\mathcal{T}$ has weight greater than ε or vertex count greater than εN, we do not only add its parent edge to C but also all edges connecting f^* to its children. We add all boundary edges of r not already in $S_{2,1}$ to $S_{2,1}$, label every node f^* of $\mathcal{T}$ with the component of $\mathcal{T} - C$ it belongs to, and once again add every edge e of R'' to $S_{2,1}$ that does not belong to $S_{2,1}$ and such that the endpoints of its dual edge belong to the same component of $\mathcal{T}$, but a different component of $\mathcal{T} - C$. We then only need to separate the components of $\mathcal{T}$, which we do by grouping consecutive components along the boundary of r into maximal sets of combined weight at most ε and vertex count at most εN. We separate consecutive sets by adding a single path to $S_{2,1}$ between their shared vertex with r and the boundary of R' (see Figure 5.2(b)). We obtain the final separator S_2 produced by Step 2 by augmenting $S_{2,1}$ in this fashion for every heavy region R' of $G' - S_{2,1}$. Since every component of $\mathcal{T} - C$ has weight at most ε and vertex count at most εN, S_2 partitions G' and, thus, G into regions of weight at most ε and vertex count at most εN. Since each region of $G - S_2$ is a collection of faces of G, S_2 is obviously 2-edge-connected. It remains to prove that S_2 has the size claimed in Lemma 5.4 and that the total number of holes

(a) (b)

Figure 5.2: Illustrating R'' in Step 2.2. The fat black edges are part of the spanning tree.
(a) Region f with its parent and children in the nesting forest. The gray path indicates the edges added to the separator in case the subtree of f^* in $\mathcal{T}$ has weight greater than ε or vertex count greater than εN.
(b) Path separating sets of components in the nesting forest.

in the regions of $G - S_2$ is at most $O(1/\varepsilon)$.

Lemma 5.5 *$|S_2| = O(\sqrt{sN/\varepsilon})$ and the total number of holes of the regions of $G - S_2$ is $O(1/\varepsilon)$.*

Proof. Consider a heavy region R' and a node f^* of the nesting forest $\mathcal{T}$ constructed for R' such that we add the edges incident to f^* in $\mathcal{T}$ to C. There are only $O(1/\varepsilon)$ such nodes over all heavy regions of $G' - S_{2,1}$. Since $|S_{2,1}| = O(\sqrt{sN/\varepsilon})$, it suffices to prove that each such node f^* adds at most $O(\sqrt{\varepsilon sN})$ edges to S_2. Note that R' is part of a heavy region $R \supseteq R'$ partitioned in Step 2.1, which has a spanning tree T of weighted diameter $O(\sqrt{\varepsilon sN})$. Since T is a spanning tree of R, and I_f contains the intervals of all descendants of f^* in $\mathcal{T}$, the part of the boundary of f not included in $S_{2,1}$ is part of a path in T; see Figure 5.2(a). Since the edges added to S_2 by f^* are exactly f's boundary, this proves that f^* adds at most $O(\sqrt{\varepsilon sN})$ edges to S_2. In addition, every heavy region R' adds the boundary of its root r to S_2. Since the size of each such face is $O(\sqrt{\varepsilon sN})$ and there are only $O(1/\varepsilon)$ heavy regions, this adds only $O(\sqrt{sN/\varepsilon})$ additional vertices to S_2. Finally, each of the in total $O(1/\varepsilon)$ paths separating sets of components of $\mathcal{T}$ adds $O(\sqrt{\varepsilon sN})$ vertices to S_2.

It remains to bound the number of holes in the regions of $G - S_2$. Since S_1 is composed of $O(1/\varepsilon)$ simple cycles, there cannot be more than $O(1/\varepsilon)$ holes in the regions of $G - S_1$. Step 2.1 partitions the heavy regions of $G - S_1$ using $O(1/\varepsilon)$ fundamental cycles. Every such cycle that touches the boundary of the partitioned region in one vertex or not at all increases the total number of holes by at most one. Any cycle that touches the boundary in at least two vertices does not create any holes. In Step 2.2, the boundary of the root face r of each heavy region R' contributes at least one edge to $S_{2,1}$. Thus, adding the whole boundary of r to S_2 amounts to adding a number of paths that start and end in distinct vertices that are already part of the separator. This cannot create any new holes. The same argument holds when adding the boundary of a face f to S_2 in Step 2.2 because f shares at least one edge with the root face r of R', whose boundary we added to S_2 previously. □

5.2.3 Step 3: Limiting the number of regions, boundary size and boundary cycles

Steps 1 and 2 of our algorithm together produce a separator S_2 of the desired total size which partitions the graph into regions of weight at most ε and size at most εN each, but there may be too many regions and individual regions may have too many holes or a too big boundary. The final step of our algorithm first ensures each region has only $O(1)$ holes and boundary size $O(\sqrt{\varepsilon s N})$ and adds only $O(\sqrt{sN/\varepsilon})$ to the size of the separator to do so. This done, we merge regions to reduce their number to $O(1/\varepsilon)$ without increasing their boundary size or number of holes.

Reducing boundary sizes. While there is a region R whose boundary size is $b > c\sqrt{\varepsilon s N}$, for an appropriate constant $c > 0$, we split it into two or more subregions and partition these subregions recursively to obtain regions with boundary size no more than $c\sqrt{\varepsilon s N}$. Slightly extending Frederickson's approach [60, Lemma 2], we triangulate the exterior and holes of R and assign weight 1 to the boundary edges of R and weight 0 to all other edges and to the vertices and faces of R. We then use Miller's simple cycle separator theorem [96] to split R into two subregions of weight at most $2b/3$ each. Since each face of R has size at most s and R has size at most εN, Miller's algorithm produces a simple cycle C of length at most $c'\sqrt{\varepsilon s N}$. Thus, the boundary of R and C, excluding edges introduced when triangulating the exterior and the holes of R, together partition R into two or more subregions of boundary size at most $2b/3 + c'\sqrt{\varepsilon s N}$, which is at most $3b/4$, for c sufficiently larger than c'. This ensures that $O(|S_2|/\sqrt{\varepsilon s N}) = O(1/\varepsilon)$ such splits suffice to produce a partition into regions of boundary size at most $c\sqrt{\varepsilon s N}$ each. Each application of Miller's algorithm to a region of size $O(\varepsilon N)$ takes $O(\varepsilon N)$ time. Since we do this $O(1/\varepsilon)$ times, the total cost of this step is linear.

Reducing the number of holes. We use a similar approach to limit the number of holes. We recursively split regions with more than a holes, for an appropriate constant $a > 0$, using Miller's simple cycle separator theorem. For such a region R, we once again triangulate its exterior and its holes. If the outer boundary of R has k edges, this splits the exterior of R into $k-2$ triangles, and we assign weight $1/(k-2)$ to each of them. We similarly weight the triangles of each hole so that their total weight is 1. We assign weight 0 to all other faces of R and to the vertices and edges of R. The two regions produced by the simple cycle C obtained using Miller's algorithm have at most two thirds of the holes of R. Thus, R's boundary and C, once again excluding edges that were introduced by triangulating the exterior and holes of C, define a partition of R into subregions that each contain at most two thirds of the holes of R. Note that every hole of R not completely on one side of C merges with C as part of the subregions' boundaries and thus does not contribute to the hole counts of the subregions. Therefore, if R has h holes, every region in the partition of R defined by R's boundary and C has at most $2(h+1)/3+1$ boundary cycles. The addition of C to the separator may result in some subregions having boundary

size greater than $c\sqrt{\varepsilon sN}$. For each such region, a constant number of splits as in the previous paragraph suffice to split it into subregions with boundary size at most $c\sqrt{\varepsilon sN}$ and with at most $2(h+1)/3+O(1)$ holes. By choosing a large enough, we can once again ensure that only $O(1/\varepsilon)$ such splits are necessary to reduce the number of holes per region to $O(1)$ while maintaining the bound on the boundary size of each region and maintaining a bound of $O(\sqrt{sN/\varepsilon})$ on the total size of the separator. Since each of these $O(1/\varepsilon)$ splits once again operates on a region of size $O(\varepsilon N)$, the total cost of these splitting steps is $O(N)$.

Merging regions. Let $S_{3,1}$ be the separator obtained so far. $S_{3,1}$ has all properties of a simple cycle ε-separator, except that we may have created too many regions. The total number of holes is $O(1/\varepsilon)$ because this was true for the regions of $G-S_{2,2}$ and each of the $O(1/\varepsilon)$ splits we have have performed to reduce the boundary size or number of holes in a region can create at most one new hole. In this final step we merge regions to reduce their number to $O(1/\varepsilon)$ while maintaining the bounds on each region's size, weight, number of holes, and boundary size.

We merge regions using the uniform graph contraction procedure of Maheshwari and Zeh [94, Section 3]. First we construct the dual of $G-S_{3,1}$, which contains a vertex per region of $S_{3,1}$ and an edge between two regions if they share a boundary edge. Let us call this graph H. We assign four types of weights to the vertices of H as follows: The weight $w_1(v)$ of a vertex $v \in H$ is the weight of the region represented by v. In addition, every boundary vertex or edge x of $G-S_{3,1}$ adds its weight to $w_1(v)$, for an arbitrary vertex v representing a region incident to x. Weights $w_2(v)$, $w_3(v)$, and $w_4(v)$ respectively are the boundary size, number of holes, and vertex count of the region represented by v. Again, every boundary vertex of $G-S_{3,1}$ increases $w_4(v)$ by one, for an arbitrary vertex v representing an incident region.

The uniform graph contraction procedure proceeds in two phases. The *contraction phase* repeatedly contracts edges in H to reduce H's size. When contracting an edge vw, the weights of the resulting vertex are the sums of the corresponding weights of v and w. At the end of this contraction phase, every vertex is either heavy or light (see below for a definition), and every light vertex is adjacent only to heavy vertices. The *bundling phase* greedily merges pairs of light degree-1 and degree-2 vertices with the same set of neighbours. Again, the weights of the vertex resulting from such a merge are the sums of the corresponding weights of the merged vertices. Both phases ensure that the weights of the vertices they produce do not exceed certain upper bounds given as parameters to the algorithm. In our application of this technique, we require that $w_1(v) \le \varepsilon$, $w_2(v) \le c\sqrt{\varepsilon sN}$, $w_3(v) \le a$, and $w_4(v) \le \varepsilon N$, for appropriate constants $a > 0$ and $c > 0$. We call a vertex *heavy* if $w_1(v) > \varepsilon/2$, $w_2(v) > (c/2)\sqrt{\varepsilon sN}$, $w_3(v) > a/2$ or $w_4(v) > \varepsilon N/2$. Otherwise the vertex is *light*. Let H^C be the graph produced from H using the uniform contraction procedure. The total number of heavy vertices in H^C is $O(1/\varepsilon)$ because their total w_1-, w_2-, w_3-, and w_4-weights respectively are 1, $O(\sqrt{sN/\varepsilon})$, $O(1/\varepsilon)$, and N. Maheshwari and Zeh proved that the total number of light vertices in H^C

is at most six times the number of heavy vertices in H^C. Thus, H^C has $O(1/\varepsilon)$ vertices.

Our final partition contains one region per vertex in H^C. The separator consists of all vertices and edges whose incident faces do not all belong to the same region of the partition. Each vertex of H^C represents a group of vertices of H that were merged into it in the two phases of the contraction procedure. Its corresponding region is the union of the regions of $G - S_{3,1}$ represented by these vertices of H. The weights assigned to the vertices of H ensure that each region in the final partition has weight no more than ε, boundary size $O(\sqrt{\varepsilon s N})$, $O(1)$ holes, and size at most εN. Since there is one region per vertex in H^C, the number of regions is $O(1/\varepsilon)$. Finally, observe that merging the regions of $G - S_{3,1}$ corresponding to vertices of H that were merged into a single vertex during the contraction phase produces a connected region because these vertices induce a connected subgraph of the dual of $G - S_{3,1}$. The bundling phase merges regions only if they are adjacent to the same set of at most two neighbours. Thus, the bundling phase produces potentially disconnected regions, but all connected components of these regions are adjacent to at most two other regions. Since these neighbouring regions correspond to heavy vertices produced by the contraction phase, they are connected. Thus, the regions in the final partition are either connected or are adjacent to at most two other regions, which are connected. It is easily verified that the construction of the graph H, the application of the uniform contraction procedure, and the construction of the final partition from H^C can each be carried out in linear time.

This proves Theorem 5.1.

5.3 Computing multiway simple cycle separators I/O-efficiently

In this section, we discuss how to combine the separator algorithm from Section 5.2 with the contraction procedure of Maheshwari and Zeh [94] to obtain an I/O-efficient algorithm for computing multiway cycle separators.

Theorem 5.2 *A simple cycle ε-separator can be computed using $O(\text{SORT}(N))$ I/Os and $O(N \log N)$ time.*

It is easy but tedious to verify that each step of the algorithm in Section 5.2 can be implemented using $O(\text{SORT}(N))$ I/Os and $O(N \log N)$ time using sorting and scanning primitives, as well as primitives to compute the connected components of a planar graph in these bounds [46] and to manipulate the dual, faces, etc. of an embedded planar graph [115]. The only exception is the computation of BFS trees in Step 1 and in all applications of Miller's simple cycle separator algorithm in Step 3. The only known method to compute BFS or shortest paths in the same I/O bound is the algorithm of [14]. In Section 5.4.1, we discuss how to reduce its internal-memory computation time to $O(N \log N)$, but the algorithm requires the input graph to be planar, which the face incidence graph is not, and it requires a multiway (cycle) separator to be given as part of the input. We overcome these two problems as follows.

BFS in the face incidence graph G_{f} of G reduces to BFS in the vertex-on-face graph G_{vf} of G (see Section 5.1): a BFS tree T_{f} of G_{f} can be obtained from a BFS tree T_{vf} of G_{vf} by making every face vertex f^* a child of its grandparent in T_{vf} (which is also a face vertex).

To overcome the circular dependency between computing separators and computing BFS or shortest paths, we use a bootstrapping approach from [94, Section 7]. The basic idea is to apply graph contraction to G to obtain a compressed graph G', recursively compute a separator partition $\mathcal{P}'$ of G' and then undo the contraction to obtain a partition $\mathcal{P}''$ of G. Due to the expansion, this partition guarantees only a region size and a region boundary size a constant factor larger than we could guarantee by computing a partition for G directly, but this is good enough for computing shortest paths in G. Once we have a shortest-path tree for G, we can use it to compute our final partition $\mathcal{P}$ of G using an I/O-efficient implementation of the algorithm from Section 5.2.

Applying this approach to compute multiway *simple cycle* separators requires two non-trivial changes, which we describe in detail in this section. First, the bootstrapping procedure from [94] only maintains a *vertex* separator, a set of vertices whose removal breaks the graph into connected components of the desired size. We need to carry out the expansion of G' to G more carefully to ensure we recover the regions of a multiway simple cycle separator from $\mathcal{P}'$. Second, our multiway simple cycle separator algorithm requires a BFS tree not of G but of the face-on-vertex graph G_{vf} of G. Thus, the partition $\mathcal{P}'$ we compute needs to be a partition of G_{vf}, not of G. Since G_{vf} is a constant factor larger than G, the parameters of the contraction procedure need to be adjusted to ensure the recursive call operates on a graph that is a constant factor smaller than G.

The notion of cycle separators is meaningful only if the graph is embedded. Thus, we have so far assumed the input graph is embedded. The applications we discuss in Section 5.4, on the other hand, make sense even in the absence of an embedding. Thus, if no embedding of the graph is given, the goal of the algorithm in this section is to compute an arbitrary embedding of G along with a multiway cycle separator based on this embedding. Since the I/O-efficient planar embedding algorithm of [93] requires a vertex separator, we cannot simply compute an embedding before computing separators, but the computation of the embedding needs to be incorporated in the recursive computation of the separator as in [94, Section 7].

In the remainder of this section, we recall the contraction procedure from [94], discuss how to recover a multiway cycle separator when undoing this contraction, and finally put things together to obtain a planar embedding and a multiway cycle separator in the I/O and time bounds stated in Theorem 5.2.

Graph contraction. The graph contraction procedure of [94] proceeds in two phases. Both phases are driven by a bound β on the *weight* of each vertex in the contracted graph, which is the number of vertices in G it represents. Initially, we set the contracted graph to be G, and every vertex has weight 1. The first phase repeatedly computes a matching of the current graph and contracts the

edges in the matching. The vertex resulting from the contraction of a matching edge uv represents the vertices in G represented by u and v. Thus, its weight is the sum of the weights of u and v. The first phase ensures that no vertex in the current graph has weight greater than β and continues contracting edges until each vertex of weight less than $\beta/2$ is adjacent only to vertices of weight at least $\beta/2$. The second phase groups degree-1 and degree-2 vertices based on the vertices they are adjacent to. Each such group of vertices with the same set of neighbours is greedily divided into maximal subgroups such that the total weight of the vertices in each group is at most β. The vertices in each such subgroup are merged into a single vertex of weight equal to the total weight of the subgroup.

As shown in [94], these two contraction phases can be implemented using $O(\text{SORT}(N))$ I/Os and $O(N \log N)$ time. The final graph G' obtained using this procedure is planar, has no vertex of weight greater than β, and has at most $14N/\beta$ vertices, where N is the number of vertices in G.

Expanding the separator. Since a multiway cycle separator is essentially defined by grouping the faces of G into regions, it is tempting to apply the above contraction procedure to the dual of G, compute a multiway simple cycle separator of the graph that has the contracted graph as its dual, and then replace each face in the resulting partition with the corresponding faces of G. The reason we do not choose this strategy is that edge contraction does not preserve vertex degrees, that is, face sizes in the primal, a property of the graph every simple cycle separator algorithm relies on. Instead, we contract the primal (which cannot increase face sizes) and then recover the boundary cycles of the regions in $\mathcal{P}''$ from those of the regions in $\mathcal{P}'$. Here we discuss how to construct $\mathcal{P}''$ from $\mathcal{P}'$.

So let $\mathcal{E}$ be an embedding of G with respect to which we want to obtain a multiway simple cycle separator, let G_2 be the graph obtained by applying the contraction procedure to G, and let $\mathcal{E}_2$ be the embedding of G_2 constructed as part of the contraction. Recall that G_2 is obtained from G in two phases. The contraction phase obtains a graph G_1 and an embedding $\mathcal{E}_1$ of G_1 by repeatedly contracting matching edges. The bundling phase groups degree-1 and degree-2 vertices with the same neighbours in G_1 to obtain G_2. We first construct a cycle separator $\mathcal{P}_1$ of G_1 from a cycle separator $\mathcal{P}_2$ of G_2 and then construct the final cycle separator $\mathcal{P}''$ of G from $\mathcal{P}_1$.

In $\mathcal{P}_2$, every edge is labelled with the two regions on each of its sides and every face is labelled with the name of the region it belongs to. Our goal is to first compute a similar labelling of the edges of G to represent $\mathcal{P}''$. Given this labelling, the faces of G can be labelled with the regions they belong to by computing the dual of G, removing all edges dual to region boundary edges (edges that do not have the same region on both sides) and then computing the connected components of the resulting graph.

The partition $\mathcal{P}_1$ of G_1 we compute from $\mathcal{P}_2$ has the same regions as $\mathcal{P}_1$. The bundling of degree-1 and degree-2 vertices in G_1 to obtain G_2 amounts to removing such vertices and their incident edges from G_1. In constructing

$\mathcal{P}_1$, we re-introduce these vertices and edges and embed them according to $\mathcal{E}_1$. This splits each face f of G_2 into a collection of faces of G_1, which should be assigned to the region of $\mathcal{P}_1$ corresponding to the region of $\mathcal{P}_2$ that contains f. To achieve this assignment of faces of G_1 to regions in $\mathcal{P}_1$, we do not alter the region labels of all edges of G_1 that also belong to G_2. For each edge re-introduced in constructing G_1 from G_2, one endpoint belongs to G_2 and the other does not. We consider each vertex in G_2 in turn and process its incident edges in G_1 in clockwise order, starting with an edge in G_2. For each edge of G_1 not in G_2, we assign the same region label to both its sides, and this region label is chosen to be the same as the region label of the clockwise side of the most recent edge in G_2 we have inspected. This reconstruction of $\mathcal{E}_1$ from $\mathcal{E}_2$ and the corresponding assignment of region labels to the edges of G_1 can be done using $O(\text{SORT}(N))$ I/Os and $O(N \log N)$ internal-memory computation time. The details are straightforward but tedious. (See [115] for an overview of the types of primitive operations on planar graphs that can be implemented using these I/O and time bounds.)

Next we construct an edge labelling of G representing the partition $\mathcal{P}''$ from the edge labelling of G_1 we just constructed. Every edge e of G_1 corresponds to a unique edge in G. This edge in G inherits e's region labels. Every vertex v of G_1 represents a connected subgraph G_v of G of constant size because v's weight is at most β and G_1 is obtained from G using only edge contractions. If v belongs to the boundary of some regions in $\mathcal{P}_1$, we need to ensure the replacement of v with G_v leaves these boundary cycles intact. We do this by labelling the edges in G_v as follows. Let $e_1, e_2, \ldots e_k$ be the edges in G corresponding to edges in G_1 incident to v, and let $v_1, v_2, \ldots, v_k$ be the endpoints of these edges in G_v. We may have $v_i = v_j$, for some of these indices. We compute a spanning tree T_v of G_v and let T'_v be the tree obtained by adding edges $e_1, e_2, \ldots, e_k$ to T_v. We compute an Euler tour of T'_v that respects the embedding of G. Note that each edge in the Euler tour represents a side of an edge of T'_v. We traverse this Euler tour, starting with one side of edge e_1. Every edge that is not a side of an edge e_i inherits its region label from the last side of an edge e_i visited by the Euler tour. This labels the edges of T_v in a manner that represents the boundary cycles of $\mathcal{P}''$. It remains to compute the labels of the edges of G_v not in T_v. This can be done in a manner identical to the construction of $\mathcal{P}_1$ from $\mathcal{P}_2$.

Bootstrapping. Now consider a planar graph G for which we want to compute a planar embedding $\mathcal{E}$ and a multiway cycle separator $\mathcal{P}$. First we compute $\mathcal{E}$ unless an embedding of G is already given. This can be done using $O(\text{SORT}(N))$ I/Os and $O(N \log N)$ time, given an $O(B^2/N)$-vertex separator of G [93]. To compute such a vertex separator $\mathcal{P}'$, we apply the above contraction procedure with a parameter $\beta = O(1)$ to be determined later. We recursively compute an embedding and a simple cycle B^2/N-separator of the resulting graph G^C, use the algorithm from Section 5.4.1 to compute a shortest-path tree T^C of G^C, and then use this shortest-path tree of T^C together with the vertex separator algorithm of [94] to obtain a B^2/N-vertex separator $\mathcal{P}^C$ of G^C.

$\mathcal{P}'$ now contains one region per region of $\mathcal{P}^C$. The region R' in $\mathcal{P}'$ corresponding to a region R in $\mathcal{P}^C$ contains all vertices of G represented by vertices in R, that is, that were contracted into vertices in R during the contraction step. The separator vertices of $\mathcal{P}'$ are all vertices of G represented by separator vertices of $\mathcal{P}^C$. Since each vertex of G^C expands into at most β vertices of G, $\mathcal{P}'$ is an $O(B^2/N)$-vertex separator of G. The cost of this procedure is $O(\text{SORT}(N))$ I/Os and $O(N \log N)$ internal-memory computation time plus the cost of recursively computing an embedding and a cycle separator of G^C. If $I(N)$ and $T(N)$ respectively denote the I/O and time bounds of our algorithm on an N-vertex graph, this recursive call uses at most $I(14N/\beta)$ I/Os and $T(14N/\beta)$ time because G^C has at most $14N/\beta$ vertices.

With an embedding $\mathcal{E}$ of G in hand, we now use a similar approach to obtain a simple cycle ε-separator of G. As already stated, this takes $O(\text{SORT}(N))$ I/Os and $O(N \log N)$ time with the exception of the cost of computing BFS trees. We need to compute BFS trees of the vertex-on-face graph G_{vf} of G, of the regions produced in Step 1 of our algorithm, and of the vertex-on-face graphs of the regions we partition in Step 3. For each such graph H, we apply the contraction procedure to obtain a compressed graph H^C along with an embedding $\mathcal{E}^C$ of H^C consistent with $\mathcal{E}$. We use the same parameter β as when constructing G^C from G. We compute a simple cycle B^2/N-separator $\mathcal{P}^C$ of H^C w.r.t. embedding $\mathcal{E}^C$ by invoking our algorithm recursively on H^C. Then we use the expansion procedure above to obtain a simple cycle $O(B^2/N)$-separator $\mathcal{P}''$ of H and use this cycle separator to compute a BFS tree T of H. Thus, the cost of the recursive call involved in computing a BFS tree for H is $I(14|H|/\beta)$ I/Os and $T(14|H|/\beta)$ time. Since $I(N)$ and $T(N)$ are convex functions, we have $\sum_H I(14|H|/\beta) \leq I(\sum_H 14|H|/\beta)$ and $\sum_H T(14|H|/\beta) \leq T(\sum_H 14|H|/\beta)$. Now it suffices to observe that the total size of all graphs for which we need to compute BFS trees in Step 1 and Step 3 of our algorithm is $O(N)$. Thus, we can choose β to be a large enough constant so that $14N/\beta \leq N/4$ (the size of G^C in computing an embedding $\mathcal{E}$ of G) and $\sum_H 14|H|/\beta \leq N/4$. For this choice of β, we obtain recurrences for the I/O complexity and running time of our algorithm bounded by $I(N) \leq 2I(N/4) + O(\text{SORT}(N))$ and $T(N) \leq 2T(N/4) + O(N \log N)$, which solve to $I(N) = O(\text{SORT}(N))$ and $T(N) = O(N \log N)$. This proves Theorem 5.2.

5.4 Applications

In this section, we discuss how to use a separator obtained using the algorithm from Section 5.3 to design algorithms for planar graphs that are efficient in internal and external memory at the same time. In particular, we prove the following result.

Theorem 5.3 *The single-source shortest-path problem on a planar graph with non-negative edge lengths, topological sorting of a planar DAG, and computing the strongly connected components of a planar graph can be solved using $O(\text{SORT}(N))$ I/Os and $O(N \log N)$ time, where N is the number of vertices in the graph.*

Algorithms for these problems with the same I/O bounds as in Theorem 5.3 were presented in [22, 25, 14], but these algorithms were not efficient in internal memory. Here we exploit the structure of the graph partition from Section 5.2 to obtain internal-memory efficient variants of these algorithms. All three algorithms use the same four-step procedure and assume that $M \geq cB^2$, for some constant $c > 0$. We recall this procedure in our discussion of the shortest-path algorithm in Section 5.4.1 and then discuss the changes necessary to use this approach to compute a topological ordering (Section 5.4.2) or strongly connected components (Section 5.4.3). Since the algorithms require a separator partition with a small number of boundary sets, we need to assume the input graphs have bounded degree. In the case of shortest paths and strong connectivity, this assumption is easily satisfied by replacing each high-degree vertex with a directed cycle of degree-3 vertices. This does not change the strong connectivity of the graph and, if the edges in each cycle are chosen to be of length 0, neither the distances between vertices. In the case of topological sorting, we are not aware of any such simple transformation that preserves planarity and acyclicity and reduces the degree of each vertex to $O(1)$. We discuss at the end of Section 5.4.2 how to obtain a topological sorting algorithm for planar DAGs without a bound on their vertex degrees.

5.4.1 Single-source shortest paths

The shortest-path algorithm of [14] proceeds in four steps. In the first step, it computes a partition $\mathcal{P}$ of the graph into $O(N/B^2)$ subgraphs (or regions) $G_1, G_2, \ldots, G_r$ of size at most cB^2, boundary size $O(B)$, and with $O(N/B^2)$ boundary sets. Using our algorithm from Section 5.3, such a separator partition can be obtained using $O(\text{SORT}(N))$ I/Os and $O(N \log N)$ time. In contrast to previous separator algorithms, our algorithm also ensures that each connected subregion of each region G_i has only $O(1)$ boundary cycles. This is the key to making the shortest-path algorithm efficient in internal memory.

Step 2 computes a compressed graph G^{R} on the boundary vertices of all regions. This graph is obtained by replacing each region G_i with a graph G_i^{R} over its boundary vertices. Let $G_{i,1}, G_{i,2}, \ldots, G_{i,k_i}$ be the connected subregions of G_i. Then G_i^{R} contains an edge uv if and only if u and v are on the boundary of the same subregion $G_{i,j}$ and its length is $\ell(uv) := \text{dist}_{G_{i,j}}(u, v)$ in this case. It is easily verified that this implies that $\text{dist}_{G^{\text{R}}}(u, v) = \text{dist}_G(u, v)$, for any two vertices $u, v \in G^{\text{R}}$.

Step 3 computes $\text{dist}_{G^{\text{R}}}(s, v) = \text{dist}_G(s, v)$, for all vertices $v \in G^{\text{R}}$. As shown in [14], this can be done using $O(\text{SORT}(N))$ I/Os and $O(N \log N)$ time because G^{R} has $O(N/B)$ vertices and $O(N)$ edges, and the partition of G it was derived from has $O(N/B^2)$ boundary sets.

Step 4 finally computes the distances $\text{dist}_G(s, v)$, for all vertices $v \in G$. To do this, it inspects each region G_i in turn and computes $\text{dist}_G(s, v) = \min_u(\text{dist}_G(s, u) + \text{dist}_{G_i}(u, v))$, for each non-boundary vertex $v \in G_i$, where the minimum is taken over all boundary vertices of G_i. Since each region fits in memory, this can be done by loading each region into memory in turn and running Dijkstra's algorithm on this region in memory. Thus, this step takes

$O(N/B)$ I/Os and $O(N \log N)$ time.

It remains to discuss the cost of Step 2 above. Its I/O complexity is $O(N/B)$, for the same reason Step 4 achieves this I/O complexity: it suffices to load each region G_i into memory, generate G_i^{R} in memory and then write G_i^{R} back to disk. By summing the I/O complexities of the four steps, we obtain an I/O bound of $O(\textsc{sort}(N))$ I/Os for the whole algorithm. This is the same argument as in [14]. The construction of G_i^{R} from G_i in memory requires the computation of distances between all boundary vertices of each connected subregion $G_{i,j}$ of G_i. A naive implementation runs $O(B)$ single-source shortest-path computations on G_i, one per boundary vertex. Since G_i has $O(B^2)$ vertices, this takes $O(B^3)$ time using the linear time single-source shortest-path algorithm of [74]. Using an all-pairs shortest-path algorithm instead is no faster. Since we have to repeat this step for each of the $O(N/B^2)$ regions in $\mathcal{P}$, the total time used in Step 2 is $O(N/B^2 \cdot B^3) = O(NB)$. Next we discuss how to reduce the internal-memory cost of constructing G_i^{R} from G_i to $O(B^2 \log B)$. This reduces the internal-memory cost of Step 2 to $O(N \log B)$ and the internal-memory cost of the whole algorithm to $O(N \log N)$, which proves the cost of computing shortest paths claimed in Theorem 5.3.

To compute the distances between all boundary vertices of each connected subregion $G_{i,j}$, we use the fact that $G_{i,j}$ has only $O(1)$ boundary cycles, each of which bounds a face of $G_{i,j}$. For each such cycle C, we can use an algorithm by Klein [82] to compute a shortest-path tree T_u for each vertex $u \in C$. This algorithm takes $O(n_{i,j} \log n_{i,j})$ time to construct all these trees, where $n_{i,j}$ is the number of vertices in $G_{i,j}$. The computed representation of each such shortest-path tree T_u supports distance queries between u and any vertex $v \in G_{i,j}$ in $O(\log n_{i,j})$ time. Since $G_{i,j}$ has $O(1)$ boundary cycles, it thus takes $O((n_{i,j} + b_{i,j}^2) \log n_{i,j})$ time to compute the distances between all boundary vertices of $G_{i,j}$, where $b_{i,j}$ is the number of these boundary vertices. Now it suffices to observe that $\sum_{j=1}^{k_i} n_{i,j} = O(B^2)$ and $\sum_{j=1}^{k_i} b_{i,j} = O(B)$ to conclude that the total cost of computing G_i^{R} is $O(B^2 \log B)$, as claimed.

5.4.2 Topological sorting

To obtain a topological ordering of the vertices of a planar DAG G, Arge and Toma [22] start with the following observation.[1] Let G^s be a DAG obtained by adding a new source vertex s to G and connecting s to each source v of G using an edge sv. Let the *level* $\ell_{G^s}(v)$ of each vertex v be the length of the longest path from s to v in G^s. A topological ordering of G can be obtained by sorting the vertices of G by their levels. To compute these levels, Arge and Toma employ the same 4-step strategy as in the shortest-path algorithm discussed in Section 5.4.1. Step 1 computes a graph partition of G with the same properties as in Section 5.4.1. Step 2 constructs a graph G^{R} whose vertex set contains the source s of G^s and all boundary vertices of G. To construct G^{R}, each region G_i in the partition is replaced with a graph G_i^{R} whose vertex set contains s

[1] We provide a somewhat different description of the algorithm here based on the same ideas.

and the boundary vertices of G_i. For every pair of boundary vertices (u, v) of G_i, G_i^{R} contains an edge uv if and only if there exists a connected subregion $G_{i,j}$ that has u and v on its boundary and there exists a path from u to v in $G_{i,j}$. The length of edge uv is the length of the longest path from u to v in $G_{i,j}$ in this case. In addition, let G_i^s be the subgraph of G^s containing s and all vertices of G_i. Then G_i^{R} contains an edge sv, for every boundary vertex v of G_i that is reachable from s in G_i^s. The length of this edge is the length of the longest path from s to v in G_i^s. Once again, it is not difficult to see that $\ell_{G^s}(v) = \ell_{G^{\mathrm{R}}}(v)$, for every vertex $v \in G^{\mathrm{R}}$. Step 3 computes $\ell_{G^{\mathrm{R}}}(v)$, for all $v \in G^{\mathrm{R}}$, by iteratively removing sinks and updating the levels of their out-neighbours. Step 4 computes $\ell_{G^s}(v)$, for all non-boundary vertices of G. To this end, it inspects each region G_i in turn and computes the levels of all vertices in a weighted supergraph $\bar{G}_i^s$ of G_i^s. In addition to the vertices and edges of G_i^s, $\bar{G}_i^s$ contains an edge sv, for each boundary vertex v of G_i. All edges of G_i^s have weight 1 in $\bar{G}_i^s$, while each edge sv not in G_i^s has weight $\ell_{G^{\mathrm{R}}}(v) = \ell_{G^s}(v)$. These weights ensure that $\ell_{\bar{G}_i^s}(v) = \ell_{G^s}(v)$, for every vertex $v \in G_i^s$. These levels can be computed for all vertices in $\bar{G}_i^s$ by once again iteratively removing sinks from $\bar{G}_i^s$ and updating the levels of their out-neighbours.

Arge and Toma [22] argue that the I/O complexity of this algorithm is $O(\textsc{sort}(N))$. Step 1 takes $O(N \log N)$ time in internal memory, while Steps 3 and 4 take $O(1)$ time per vertex and edge and operate on graphs of total size $O(N)$. Thus, it remains to bound the internal-memory computation time of Step 2. In particular, we argue once again that the construction of G_i^{R} from G_i takes $O(B^2 \log B)$ time.

The addition of edges sv to G_i^{R}, for all boundary vertices of G_i reachable from s in G_i^s, and the computation of the weights of these edges is yet another level computation on G_i^s and hence takes $O(B^2)$ time. To add edges between the boundary vertices of each connected subregion $G_{i,j}$, we observe that the length of the longest path from u to v in $G_{i,j}$ is the negation of the length of the shortest path from u to v in $G_{i,j}$ after assigning length -1 to each edge. As noted by Klein et al. [83, Section 2.5], the algorithm of Klein [82] can be used to compute shortest paths from all vertices on the same boundary cycle C even in the presence of negative-length edges, provided the distances from some vertex $v \in C$ to all other vertices in $G_{i,j}$ are known. Computing these distances requires a longest-path computation (with the original edge lengths) similar to the addition of edges sv to G_i^{R}. Only this time we operate on a graph $G_{i,j}^s$, which is obtained from $G_{i,j}$ by adding a single length-0 edge sv. We repeat this distance computation once per boundary cycle of $G_{i,j}$. Since $G_{i,j}$ has $O(1)$ boundary cycles, this takes $O(n_{i,j})$ time. Computing the distances between all pairs of boundary vertices of $G_{i,j}$ using Klein's algorithm takes $O((n_{i,j} + b_{i,j}^2) \log n_{i,j})$ time again, which dominates the cost of processing the subregion $G_{i,j}$. Thus, by summing these costs for all subregion of G_i as in Section 5.4.1, we obtain a bound of $O(B^2 \log B)$ for the internal-memory cost of constructing G_i^{R} from G_i and a bound of $O(N \log B)$ for the total internal-memory cost of Step 2.

One limitation of the algorithm just described is that it applies only to

planar graphs of bounded degree. The more complicated topological sorting algorithm of Arge, Toma and Zeh [23] does not make this assumption but uses an I/O-efficient shortest-path algorithm to compute two rooted spanning trees of the dual of the given DAG. Using the algorithm discussed in Section 5.4.1, these spanning trees can be computed using $O(\text{SORT}(N))$ I/Os and $O(N \log N)$ time. It is easy but tedious to verify that the remainder of the algorithm of [23] performs only $O(\text{SORT}(N))$ I/Os and $O(N \log N)$ time. Thus, we obtain the complexity of topologically sorting planar DAGs claimed in Theorem 5.3.

5.4.3 Strongly connected components

To compute the strongly connected components of a planar graph, Arge and Zeh [25] use the following algorithm, once again based on the 4-step framework from Section 5.4.1. Step 1 computes a graph partition with the same properties as in Section 5.4.1. Step 2 computes a graph G^{R} from G by replacing each region G_i of the partition with a graph G_i^{R} over its boundary vertices. G_i^{R} contains an edge uv if and only if there exists a connected subregion $G_{i,j}$ of G_i that has u and v on its boundary and there exists a path from u to v in $G_{i,j}$. It is easy to verify that two vertices of G^{R} belong to the same strongly connected component of G^{R} if and only if they belong to the same strongly connected component of G. Step 3 assigns a *component label* to each vertex u of G^{R}, which uniquely identifies the strongly connected component of G^{R} (and hence of G) that contains u. Step 4 extends this labelling by assigning component labels to the non-boundary vertices of G. To this end, it processes each region G_i in turn. Region G_i is augmented with cycles connecting the boundary vertices of G_i that belong to the same strongly connected component of G according to the component labels computed in Step 3. Let G_i' be the resulting graph. Next the algorithm assigns component labels to the vertices of G_i' such that every boundary vertex of G_i receives the same component label as in Step 3 and two vertices in the same strongly connected component of G_i' receive the same label. It is easy to verify that this ensures that the final labelling assigns the same component label to two vertices if and only if they belong to the same strongly connected component of G.

Once again, we use our separator algorithm from Section 5.3 to compute the graph partition in Step 1 using $O(\text{SORT}(N))$ I/Os and $O(N \log N)$ time. Arge and Zeh showed that the remainder of the algorithm takes $O(\text{SORT}(N))$ I/Os, and it is easy to verify that Steps 3 and 4 take $O(N)$ time. To implement Step 2, it suffices to compute the distances between the boundary vertices of each connected subregion $G_{i,j}$ as discussed in Section 5.4.1. If this distance is finite, a path between the two vertices exists, and we add the corresponding edge to G_i^{R}; otherwise we do not. Thus, Step 2 takes $O(N \log B)$ time as in Section 5.4.1, and we obtain a strong connectivity algorithm with the complexity claimed in Theorem 5.3.

Part II

Range Searching Data Structures

Chapter 6

Range Searching Background

Orthogonal range reporting is not only the most important example of range searching, but also one of the classic and most fundamental data structure problems in the fields of computational geometry and (spatial) databases. Given a set of N points in d-dimensional space, the goal is to represent the points in a data structure, such that given an axis-aligned query rectangle, one can efficiently list all points contained therein. A standard example motivating such queries in the field of databases is: "Give me a list of all employees hired between 2001 and 2004 who earn between 60 000 and 80 000 per year". By representing each employee as the point (year hired, salary), this query translates into the orthogonal range query specified by the axis-aligned rectangle $[2001 : 2004] \times [60\,000 : 80\,000]$. Given its key role, this problem has been extensively studied in almost all models of computation, including the word-RAM and the I/O-model. The two-dimensional variant of the problem is particularly well understood, with optimal I/O-model solutions dating back to the 1990s [19]. However, in many natural applications, standard orthogonal range reporting is too generic as specialized problems may be solved much more efficiently using specialized data structures. In this part of the dissertation we investigate two such specialized types of range searching: range diameter and categorical range reporting.

The first problem is to report only the two furthest points in a query range as opposed to all points. It would be great if we could save the time necessary for reporting all points in case we only need the two furthest points without increasing the search time, but known (internal-memory) solutions to the problem do not even come close to this goal. We therefore look into the hardness of this problem: Are there known hard problems that are related to it? Is the problem also hard in simplified scenarios?

For the second problem, the situation looks much brighter. We are interested in the scenario where points are assigned a category (or color), and a query asks to report only the set of colors represented by the points inside the query range. For this problem data structures have been developed that solve the problem fairly efficiently in different models of computation, and, in the pointer-machine, even optimally. However, especially the existing I/O-model solutions are not as efficient as one would hope for, and also in the word-RAM model further improvements are possible.

In the remainder of this introductory chapter, we first revisit the topic of

computational models to highlight some assumptions that are typically made for data structure design and lower bounds. Then, we give an overview of existing work on the two problems, and where relevant, related problems. Finally, we give an overview of the specific contributions in the chapters ahead.

6.1 Models of computation

We briefly summarize the models of computation relevant for the following chapters.

Word-RAM. In the unit-cost word-RAM model, a data structure consists of a memory divided into words of w bits each. For range reporting problems, we make the standard assumption that the number of bits in a word is proportional to the number of bits needed to represent an input point and to address any input point, i.e. if the input points have integer coordinates from a universe $[U] = \{0, \ldots, U-1\}$, then we assume $w = \Theta(\log N + \log U)$. The memory words may represent arbitrary information about the input. When answering a query, a data structure may use indirect addressing and any standard operation on words in constant time, including for instance addition, subtraction, multiplication, division and bitwise operations.

Real-RAM. For applications in computational geometry the word-RAM often does not suffice since it would only support integer coordinates. Therefore, a variant of the word-RAM called the real-RAM model is used extensively (and often implicitly) in computational geometry [107, Section 1.4]. The difference with the word-RAM is that in the real-RAM a word is allowed to store any arbitrary real number, while the operations on these numbers are restricted to a set of reasonable primitives such as addition, subtraction, multiplication, division, comparison, as well as exponentiation and taking a logarithm. This restriction is necessary since the number of bits in a word is not restricted, and thus certain operations (such as the floor function) could be "abused" to solve hard problems using fewer operations than would be possible on any physical machine.

I/O-model. For designing data structures in the I/O-model, we assume that the B words making up a disk block consist of $\Theta(\log N + \log U)$ bits each, as in the word-RAM model. The data structure is required to fully reside on disk between operations, so relevant parts need to be moved to and from main memory when needed. Another assumption that is typically made for range reporting data structures in the I/O-model is the *indivisibility assumption*, that is, an input point has to be stored "uncompressed" in a word in order to be reported. Other than that, words can store arbitrary information and a data structure is allowed to perform random accesses to disk blocks. Making the indivisibility assumption typically allows for very high and near-tight lower bounds using the indexability framework of Hellerstein et al. [73]. The I/O-model data structures we present all satisfy the indivisibility assumption.

Pointer machine. The pointer machine model is a constrained variant of the word-RAM where the memory words of the data structure can only be accessed through pointers. Furthermore, the input points to a range searching problem are assumed *indivisible* and we assume coordinates can only be *compared*. This means that any memory word may store one input point plus a constant number of pointers to other memory words. The main motivating factor for studying the pointer machine is that we can prove polynomially high and often tight lower bounds for range reporting problems using the framework of Chazelle [45]. This stands in sharp contrast to the highest query time lower bound proved for *any* static data structure problem in the word-RAM, which is only $\Omega(\log N)$, even for linear space data structures [88]. Note that since any memory cell stores a constant number of pointers, one cannot generally hope for a query time below $\Theta(\log N)$.

6.2 Range-aggregate extent queries

In this section, we review previous work on range-aggregate queries for sets of points in the plane, which is a class of queries similar to our problem of interest: finding (the distance between) the furthest pair of points in a query range. A brief description of problems in this class follows.

- Diameter: the distance between the pair of furthest points in a point set.
- Closest pair distance: the distance between the pair of closest points in a point set.
- Width: minimum distance between any two parallel lines that enclose the point set.
- Radius of minimum enclosing disk: the radius of the smallest disk covering the point set.

These functions are often used to measure the "extent" or "spread" of points in applications like clustering, collision detection, shape-fitting, data mining, etc. [7, 28, 58, 71, 72]. Computing aggregate functions on a subset of points contained in a query range is interesting from the perspective of both computational geometry and database applications, and has attracted the attention of researchers from both research communities [5, 68, 69, 75, 101, 108].

Unlike many other range searching problems (such as range counting, range reporting, and range maximum), the range-aggregate functions described above are not decomposable in the sense that they cannot be computed by dividing the point set into subsets, computing the function for each subset, and then aggregating these partial results [68]. In fact, the main difficulty in such a divide and conquer strategy is to combine the partial results, that is, for example finding the diameter of two point sets. Storing the answers to all such subproblems would yield constant query time per subproblem, but uses at least quadratic space in total. We can reduce the amount of space required for answering diameter queries as follows. Let S_1 and S_2 be two disjoint subsets of the input

point set, where $|S_1| \leq |S_2|$. We can find the largest distance between each point in S_1 and all points in S_2 in $O(|S_1| \log |S_2|)$ time using a furthest-point Voronoi diagram of S_2. The currently best known solution to the range diameter problem stores the diameter for pairs of large subsets (with more than k points), while using Voronoi diagrams for finding the diameter between other pairs of subsets [69]. This results in a tradeoff with $O((N + (N/k)^2) \log^2 N)$ space and $O(k \log^5 N)$ query time, for a parameter k where $1 \leq k \leq N$.

Even though, strictly, the closest pair problem is not decomposable, more efficient solutions have been obtained for this problem [1, 68, 69, 111, 112]. Specifically, Gupta et al. [69] describe a data structure using $O(n \log^5 n)$ space that answers queries in $O(\log^2 n)$ time. For the width and minimum enclosing disk problems no exact solutions are known, but a $(1 + \delta)$-approximation of the width can be obtained with an $O(1/\sqrt{\delta} \cdot n \log^2 n)$ space data structure in $O(1/\sqrt{\delta} \cdot \log^3 n)$ time [69], while the radius of a minimum enclosing disk can be approximated using coresets [101].

6.3 Categorical range searching

Consider the following generalization of the query we considered earlier: "Give me the different job positions of those employees hired between 2001 and 2004 who earn between 60 000 and 80 000 per year". This problem can obviously be solved using standard orthogonal range reporting by augmenting the above points with the job position of the corresponding employee and then scanning through the output list. However, this solution may be very inefficient as the number of employees with the same job position could be huge. In range searching terminology, the above problem is an instance of *categorical range reporting* (a.k.a. *colored range reporting* or *generalized range reporting*). For categorical range reporting, we assume the input consists of a set of N points in d-dimensional space, each assigned a color. The goal is to represent the points in a data structure, such that given an axis-aligned query rectangle, one can efficiently report the set of distinct colors assigned to points contained in the query rectangle. While the categorical range reporting problem has numerous applications (for example in document retrieval, IP package routing, geographic information systems and VLSI layout, see [6, 70, 80]), determining the exact complexity of the problem remains open, even in the two-dimensional case.

The categorical range reporting problem was first introduced by Janardan and Lopez in 1993 [80]. Their solutions are for the pointer machine model and they presented an optimal $O(N)$ space and $O(\log N + K)$ query time data structure for one-dimensional categorical range reporting. Here, and throughout the chapter, K denotes the number of distinct colors in the output of a query, and space is measured in number of words. For the (four-sided) two-dimensional problem, they presented two different tradeoffs: One data structure using $O(N \log N)$ space and answering queries in $O(\log^2 N + K)$ time and one using $O(N \log^2 N)$ space and answering queries in optimal $O(\log N + K)$ time.

The next results on the two-dimensional problem were due to Agarwal et

al. [6]. They studied the problem in the word-RAM model and assumed the coordinates of the input points lie on a $U \times U$ integer grid. For such inputs, they presented a data structure using $O(N \log^2 U)$ space and answering queries in $O(\log \log U + K)$ time. This query time is optimal by reduction from predecessor search [106]. Agarwal et al. also consider the case where the query is three-sided. For this problem, they present a data structure using $O(N \log N)$ space and answering queries in $O(\log \log U + K)$ time. Again the query time is optimal.

Subsequently, Shi and JaJa [113] presented a linear space pointer machine data structure that answers three-sided queries in optimal $O(\log N + K)$ time. By a standard reduction, this also gave an $O(N \log N)$ space data structure for four-sided queries, answering queries in optimal $O(\log N + K)$ time. Other variants of categorical range reporting were investigated by Gupta et al. [70] and Bozanis et al. [35, 36], with a main focus on dynamic data structures and other types of query ranges.

Very recently, several results were presented for categorical range reporting in the I/O-model [89, 100]. In [100], Nekrich presents a number of tradeoffs for three-sided queries: His first data structure uses linear space and answers queries in $O((N/B)^\varepsilon + K/B)$ time, where $\varepsilon > 0$ is an arbitrarily small constant. Note that the desired output term in the I/O-model is $O(K/B)$ and not $O(K)$, since writing K output points to disk costs $O(K/B)$ I/Os. For input points with coordinates on an $N \times N$ grid (rank space), he presented two additional data structures for three-sided queries, one answering queries in optimal $O(1+K/B)$ I/Os using $O(N \log \log N \log \log \log N)$ space, and one answering queries in $O(\log^\varepsilon \log N + K/B)$ I/Os with $O(N \log \log N)$ space, where $\varepsilon > 0$ is an arbitrarily small constant. Finally, he presented a data structure for three-sided queries when the coordinates are on a $U \times U$ grid. This data structure answers queries in $O(\log \log_B U + K/B)$ I/Os and uses $O(N \log \log N \log \log \log N)$ space. The query time is optimal by reduction from predecessor search [106]. Using these data structures as building blocks, he also obtained an improved data structure for the general problem on a $U \times U$ grid, using $O(N \log N \log \log N \log \log \log N)$ space and answering queries in optimal $O(\log \log_B U + K/B)$ I/Os. We note that all the data structures of Nekrich can be implemented using only comparisons, at the cost of increasing the search cost to $O(\log_B N)$. They can also be implemented in the word-RAM at the cost of removing the B's from the above bounds. His results for input points with coordinates on a $U \times U$ grid thus improve over the previous best word-RAM results of Agarwal et al. [6]

For the related problem of one-dimensional categorical range counting (that is, counting the number of different colors in a one-dimensional range), Gupta et al. [70] showed a reduction to two-dimensional range counting *without* colors. Thus using the two-dimensional range counting word-RAM data structure of JaJa et al. [79], one can solve the problem in linear space and $O(\log N / \log \log N)$ time. This bound is optimal for two-dimensional range counting [104], but is not known to be optimal for the one-dimensional categorical range counting problem.

Unconditional lower bounds. Since standard range reporting is a special case of categorical range reporting, we can conclude that any I/O-model data structure with $O(f(N,B) + K/B)$ query time for two-dimensional categorical range reporting (four-sided queries) must use $\Omega(N \log N / \log f(N,B))$ space [19]. When coordinates can only by compared, the optimal bound on $f(N,B)$ is $O(\log_B N)$, so the space lower bound is $\Omega(N \log N / \log\log_B N)$. We note that $O(\log_B N)$ is also the best achievable query time for two- and three-sided queries when coordinates can only be compared. When coordinates are integers on a $U \times U$ grid, where $U = N^{1+\Omega(1)}$, the optimal bound on $f(N,B)$ is $O(\log\log_B U)$ for all three variants [106]. Finally, if points are on an $N \times N$ grid, only the four-sided problem has an $\Omega(\log\log_B N)$ lower bound [106], whereas two- and three-sided queries can be solved with $f(N,B) = O(1)$.

In the word-RAM, the only lower bounds known are obtained by reduction from predecessor search. This allows us to conclude that for any $N \log^{O(1)} N$ space, the query time must be $\Omega(\log\log U + K)$ for the four-sided problem on points with coordinates on a $U \times U$ grid. For the three-sided problem on a $U \times U$ grid we can similarly conclude that the query time must be $\Omega(\log\log U + K)$ for any $N \log^{O(1)} N$ space, provided that $U = N^{1+\Omega(1)}$. We note that on an $N \times N$ grid, the standard three-sided problem can be solved in $O(1 + K)$ time [11], while the four-sided problem still has an $\Omega(\log\log N + K)$ lower bound [106].

Since these word-RAM lower bounds do not specify precisely what space we can achieve in optimal query time, we mention that the best known solution for standard range reporting (four-sided) in the word-RAM answers queries in optimal $O(\log\log U + K)$ time using $O(N \log^{\varepsilon} N)$ space, where $\varepsilon > 0$ is an arbitrarily small constant [44].

6.4 Contributions

Most of the contributions in this part of the dissertation are of a more theoretical nature, and include reductions showing that lower-bounds can be carried over between certain problems. Still, some of the presented data structures may very well be implemented. We now give an overview of the results obtained in the following chapters.

Range diameter queries

As mentioned earlier, we are interested in internal-memory data structures that store a set of points in the plane such that the furthest pair of points in a given orthogonal query range can be found efficiently. The currently best-known data structure for this problem uses $O((N + (N/k)^2)\log^2 N)$ space and has $O(k \log^5 N)$ query time, for $1 \leq k \leq N$ [69]. In other words, a data structure with polylogarithmic query time requires almost quadratic space. In Chapter 7, we show that this is also likely the best possible (up to polylogarithmic factors). We come to this conclusion by giving a reduction from the set intersection problem, for which it is widely believed that no significantly better solution exists. We also give two lower bounds for related problems using similar reductions.

Finally, we show that range diameter queries can be answered much more efficiently for the case of points in convex position by describing a data structure of size $O(N \log N)$ that supports queries in $O(\log N)$ time. The amount of space used by this data structure can be reduced at the expense of query time. The same techniques can also be used to construct a data structure for answering range width queries with the same space and time bounds.

Categorical range reporting

In Chapter 8, we obtain several improved results on orthogonal range reporting on two-dimensional categorical data, as well as a new, tight lower bound for one-dimensional categorical range counting (i.e. counting the number of distinct colors in a query interval). For the word-RAM, we obtain a linear space data structure with $O(\log \log U + K)$ query time for input points on a $U \times U$ grid and $O(1 + K)$ query time for input points on an $N \times N$ grid, where K is the number of distinct colors in the output, which is optimal in both cases. For the I/O-model, we present two alternative data structures for three-sided queries (rectangular ranges that are unbounded on one side). The first data structure answers queries in optimal $O(\log_B N + K/B)$ I/Os using $O(N \log^* N)$ space, where $\log^* N$ denotes the iterated logarithm of N, that is, the smallest value h such that $\log^{(h)} N$ is constant. Our second data structure uses linear space and answers queries in $O(\log_B N + \log^{(h)} N + K/B)$ I/Os for any constant integer $h \geq 1$. Here $\log^{(h)} N$ is the logarithm iterated h times, that is, $\log^{(1)} N = \max\{1, \log N\}$ and $\log^{(i)} N = \max\{1, \log(\log^{(i-1)} N)\}$. We show that the $\log_B N$ terms in the query costs can be reduced to optimal $\log \log_B U$ when the input points lie on a $U \times U$ grid, and even further to just $O(1)$ if the points are given on an $N \times N$ grid. Both solutions also lead to improved data structures for four-sided queries.

Chapter 7

Two-Dimensional Range Diameter Queries

Computing the diameter of points contained in a rectangular query range (see Figure 7.1) is an interesting type of range aggregate extent query for which no efficient data structure currently exists. More formally, the problem we study in this chapter is preprocessing a set of points from $\mathbb{R}^2$ into a data structure such that given an orthogonal range query, we can find the pair of furthest points within the query range efficiently. We are specifically interested in finding out why this and related (sub)problems turn out to be hard, and whether restrictions on the input (such as convexity) make the problem easy to solve.

Our contribution

In Section 7.1, we look into the hardness of range diameter queries. We show that determining the disjointness of two sets among a collection of sets, that is, a *set-intersection* query, can be solved using a range diameter query over a suitable set of points (reducing set-intersection queries to other problems has been previously considered to study the hardness of approximate distance oracles [49, 105]). Obtaining space-efficient data structures that support set-intersection queries is known to be hard. A folklore conjecture states that, for m sets of cardinality polylogarithmic in m, answering a set-intersection query in $O(1)$ time requires an $\tilde{\Omega}(m^2)$ space data structure[1], and answering a set-intersection query in polylogarithmic time (but asymptotically smaller than the maximum cardinality of the sets) requires $\tilde{\Omega}(m^{2-\varepsilon})$ space [105]. Our reduction in Section 7.1.1 implies that range diameter queries are as hard as set-intersection queries in the real-RAM without the floor function. We conjecture that $\tilde{\Omega}((N/k)^2)$ space is required to answer range diameter queries in $\tilde{O}(k)$ time.

As previously mentioned, in answering range diameter queries using a divide and conquer strategy, computing the diameter of two disjoint point sets arises as a subproblem. In 1985, Edelsbrunner [55, Section 4] considered the related problem of computing the diameter of two convex polygons that are separated by a vertical line and are preprocessed independently. He showed that if each polygon is represented as a list of vertices, then linear query time in the number of vertices of the larger polygon is required. In Section 7.1.2, we show a cell probe lower bound that is almost linear in the number of vertices

[1] We use tilde notation to hide polylogarithmic factors.

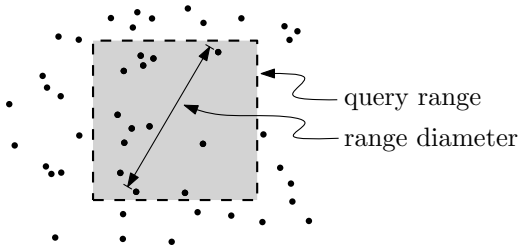

Figure 7.1: The answer to a range diameter query is a pair of points that are furthest away in a given orthogonal range.

of the smaller polygon, no matter how much space and preprocessing time is spent. This lower bound not only addresses an open problem mentioned by Edelsbrunner [55], but also may be a step forward in proving our lower bound conjecture in Section 7.1.1 for range diameter queries.

Our reduction from set-intersection queries to range diameter queries uses a hard instance of the problem consisting of a set of points placed on a number of concentric circles, thus the set is not in convex position (see Figure 7.2). In Section 7.2, we show that range diameter queries can be answered much more efficiently in case points are in convex position. In particular, we describe a data structure that stores a set of N points in convex position using $O(N \log N)$ space such that range diameter queries can be answered in $O(\log N)$ time. Interestingly, this data structure is not as trivial as one might expect.

In Section 7.2, we also show how to adapt and extend the range diameter data structure to create a data structure with the same space and query bounds for finding the width within any given query range of a set of points in convex position.

7.1 Reduction from set-intersection queries

In this section, we provide support for the hardness of range diameter queries by presenting a reduction from the set-intersection problem. This reduction implies a lower bound for range diameter queries based on a generalization of a folklore conjecture for the set-intersection problem. We finish this section by proving a lower bound for determining the diameter of two convex polygons that are separated by a vertical line. This problem usually arises as a subproblem in answering range diameter queries. We conclude that this lower bound may be a step forward in proving our conjecture for range diameter queries.

The set-intersection problem is to preprocess m sets $S_1, S_2, \ldots, S_m$ of positive real numbers into a data structure that supports set-intersection queries asking whether the sets S_i and S_j are disjoint, for given query indices i and j. Let T be the total number of elements in the sets, that is $T = \sum_{i=1}^{m} |S_i|$.

Theorem 7.1 *Given a data structure of size $s(N)$ that supports range diameter queries in $t(N)$ time on any point set of N in the plane, we can build a data structure of size $s(2T)$ supporting set-intersection queries in $t(2T)$ time.*

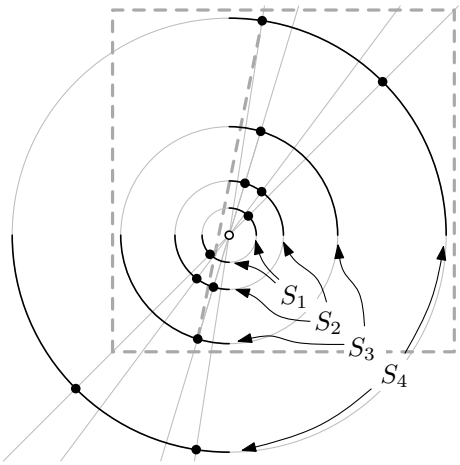

Figure 7.2: Example of the reduction in Theorem 7.1. Each element e is shown as a line $y = ex$, with points at the intersections with circle c_i if and only if $e \in S_i$. The dashed query rectangle is used to report whether S_3 and S_4 intersect. As the diameter (indicated by the dashed line) is less than $r_3 + r_4$, $S_3 \cap S_4 = \emptyset$. Note that $S_2 \cap S_3 \neq \emptyset$.

Proof. We transform sets $S_1, S_2, \ldots, S_m$ into a set P of $2T$ points in the plane, and show that any set-intersection query can be answered using an appropriate range diameter query on P.

Let $r_i = 2^{i-1}$ for $i = 1, \ldots, m$. We map each $e \in S_i$ to two points in P, positioned on the first and third quadrant of the circle c_i with radius r_i centered on $(0, 0)$. The positions are determined by the two intersection points of the line $y = ex$ with c_i (see Figure 7.2). Note that for $e \in S_i$ and $e \in S_j$, the distance between the corresponding points on the first quadrant of c_i and the third quadrant of c_j is $r_i + r_j$. By the triangle inequality, for $e \in S_i$ and $e' \in S_j$, where $e \neq e'$, the distance between the point corresponding to e on the first quadrant of c_i and the point corresponding to e' on the third quadrant of c_j is less than $r_i + r_j$. Therefore, to find out whether S_i and S_j are disjoint, we ask a range diameter query over the rectangle with bottom-left point $(-r_i, -r_i)$ and top-right point (r_j, r_j). If the diameter of the points within this rectangle is $r_i + r_j$, then $S_i \cap S_j \neq \emptyset$, and if the diameter is smaller than $r_i + r_j$, then $S_i \cap S_j = \emptyset$ (they are disjoint). □

7.1.1 Conditional lower bound

We can naively solve the set-intersection problem with $O(1)$ query time using $O(m^2)$ space by tabulating the answers to all queries. Cohen and Porat [49] note that it is also possible to construct a data structure of size $O((T/k)^2)$ that supports set-intersection queries in $O(k \log T)$ time, for a parameter k where $1 \leq k \leq T$. The solution simply consists in tabulating the answers of queries for which both sets have at least k elements. Otherwise, let S_i and S_j be the queried sets and assume w.l.o.g. that S_i has less than k elements. We then simply search for each element of S_i in S_j in logarithmic time. (The query time can

be improved to $O(k)$ using linear perfect hashing in the word-RAM [48, 49].) Note that the same approach was used in [69] to obtain the currently best known data structure for range diameter queries. Pătraşcu and Roditty [105] describe a folklore conjecture stating that $\tilde{\Omega}(m^2)$ space is required to support set-intersection queries in $O(1)$ time, for a universe of size polylogarithmic in m. They also strengthen the conjecture to polylogarithmic query time (but asymptotically smaller than the maximum cardinality of the sets) and a space lower bound of $\Omega(m^{2-\varepsilon})$ in the cell probe model. The following is a generalization of their conjecture, which would imply that the best known upper bound of Cohen and Porat [49] is optimal up to polylogarithmic factors.

Conjecture 7.1 *Given a collection of m sets of T real numbers in total, where the maximum cardinality of the sets is polylogarithmic in m, any real-RAM data structure that supports set-intersection queries in $\tilde{O}(k)$ time without using the floor function, requires $\tilde{\Omega}((T/k)^2)$ space, for $1 \leq k \leq T$.*

From Theorem 7.1 and Conjecture 7.1, we conclude the following.

Theorem 7.2 *Assuming Conjecture 7.1, any real-RAM data structure that supports range diameter queries on a set of N points from $\mathbb{R}^2$ in $\tilde{O}(k)$ time without using the floor function, requires $\tilde{\Omega}((N/k)^2)$ space, for $1 \leq k \leq N$.*

Remark. In our reduction in Section 7.1, we transform a collection of sets into a set of points with exponentially large coordinates. As a result, the lower bounds for the set-intersection problem only imply lower bounds for range diameter queries in a computational model where working with unbounded numbers is allowed (such as the real-RAM). An interesting open problem is to give an equivalent transformation algorithm in the word-RAM. Such a transformation would imply that cell probe lower bounds for the set-intersection problem also apply to range diameter queries.

7.1.2 Diameter of two convex polygons

We prove a lower bound for the problem of representing two convex polygons P and Q in the plane, that are separated by a vertical line (the preprocessing of each polygon into its representation is oblivious to the other polygon), such that we can determine the furthest pair of points in $P \cup Q$ using the two representations. This problem often arises as a subproblem when answering range diameter queries, in case we divide a query into disjoint subqueries and then combine the answers of the subqueries. Our lower bound essentially shows that it is hard to combine the answers of two subqueries if we do not store any information about both subqueries together. This may be a step forward in proving Theorem 7.2 unconditionally.

In 1985, Edelsbrunner [55, Theorem 4.1] showed that if we represent a polygon as a list of vertices, then $\Omega(|P|+|Q|)$ is a lower bound on the worst-case time complexity of determining the diameter of $P \cup Q$. He raised the question of determining the complexity of the problem for other representations of polygons.

We address this open problem by proving a lower bound of $\tilde{\Omega}(\min\{|P|, |Q|\})$ for any representation of the polygons, derived by a reduction from the asymmetric (lopsided) version of the *set-disjointness* problem in communication complexity. Our reduction is similar to the reduction from set disjointness to computing the diameter of a planar point set [107, Section 4.2.3]. This reduction implies a lower bound of $\Omega(N \log N)$ time to compute the diameter of a planar point set in the algebraic computation tree model.

The asymmetric set-disjointness problem in communication complexity can be described as follows. Alice has a set A and Bob has a set B, and they wish to determine whether sets A and B are disjoint, given that $A, B \subseteq [N]$, and $|A| < |B| < N/2$. It is known that Alice and Bob need to communicate at least $\Omega(|A|)$ bits in this case [97]. This lower bound implies that for any representation of two given sets A and B, $\tilde{\Omega}(|A|)$ time is required to establish the disjointness of A and B in the cell probe model. We use this lower bound to prove the following.

Theorem 7.3 *For any independent representation of two convex polygons P and Q that are vertically separated, finding the furthest pair of points in $P \cup Q$ requires $\tilde{\Omega}(\min\{|P|, |Q|\})$ time, in the cell probe model.*

Proof. As previously mentioned, $\tilde{\Omega}(|A|)$ time is required to determine whether sets A and B are disjoint, for any representation of given sets A and B, where $A, B \subseteq [N]$, and $|A| < |B| < N/2$. We construct two vertically separated point sets P and Q corresponding to A and B respectively, and then we show that the disjointness of A and B can be verified by finding the diameter of $P \cup Q$.

We map each element $e \in A$ to a point positioned on the intersection point of the line $y = ex$ with the first quadrant of the unit circle. Similarly, we map each element $e \in B$ to a point positioned on the intersection point of the line $y = ex$ with the third quadrant of the unit circle. Hence, P has $|A|$ points and Q has $|B|$ points. It is clear that there exists an element e belonging to both A and B if and only if there exist a point $p \in P$ and a point $q \in Q$ such that the distance between p and q is 2. We compute the diameter of $P \cup Q$. If the diameter is 2 then there is a common element in A and B, and otherwise (the diameter is less than 2) A and B are disjoint. □

7.2 Points in convex position

As it appears unlikely that we can get polylogarithmic query time when using $O(N^{2-\varepsilon})$ space for range diameter queries on sets of N points, we consider in this section the case of sets of points in convex position. For this case we describe data structures with polylogarithmic query time using near-linear space. The precise bounds on space and query time depend on the choice of underlying data structures. We also describe a data structure for the range width problem with the same bounds.

Let a *section* of a convex polygon be a sequence of consecutive vertices of that polygon. We first describe how to find the (at most four) disjoint sections containing all the vertices covered by a query range (see Figure 7.3). Second, we

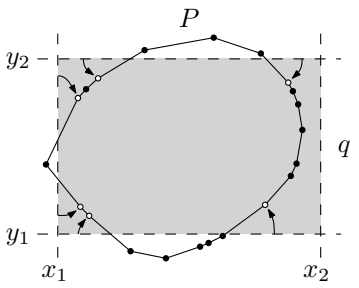

Figure 7.3: Range diameter query on the vertices of a convex polygon. In this example, the query range covers three disjoint sections. Predecessor (and successor) queries are indicated by arrows. The white vertices are within the query range and determine the sections of $q \cap P$.

show how to solve the problem of finding the furthest pair of points between two given sections. For the description and analysis of our approach to answering such *section–section queries* we review a characteristic of convex polygons called *modality* and a derivative thereof that we use in our analysis. We show that section–section queries can be answered efficiently using two data structures: one storing the distances between a set of $O(N)$ selected point pairs explicitly, and one for answering *point–section queries*, a special case of section–section queries in which one section contains only one point.

7.2.1 Reduction to section–section queries

The following lemma is easy to prove using predecessor data structures (Figure 7.3).

Lemma 7.1 *A convex polygon $P = (p_1, p_2, \ldots, p_N)$ can be preprocessed to obtain a linear space data structure for finding the at most four sections of P intersecting a given query range $q = [x_1 : x_2] \times [y_1 : y_2]$ in $O(\log N)$ time.*

Let S_a be the sequence of points in the ath section of $q \cap P$. The diameter of the points in $q \cap P$ can be found by taking the maximum of all point pair distances in all pairs of sections: $\max_{a,b}\{\max_{p \in S_a, q \in S_b}\{d(p,q)\}\}$, where $d(p,q)$ is the Euclidean distance between points p and q. We can therefore focus on determining the maximum point pair distance between two (possibly equal) sections S_a and S_b.

7.2.2 Section–section queries

The main complicating factor in designing algorithms for convex polygons appears to be the fact that for a given vertex of a convex polygon the sequence of distances to the other vertices in order around the polygon may contain more than one local maximum. The maximum number of local maxima in the distance sequence of any vertex of a polygon $P = (p_1, p_2, \ldots, p_N)$ is called the *modality* of P. More formally, take $p_0 := p_n$, $p_{N+1} := p_1$, and let $M_i := \{1 \leq j \leq N : d(p_i, p_{j-1}) < d(p_i, p_j)$ and $d(p_i, p_{j+1}) < d(p_i, p_j)\}$ be the set of local

maxima for vertex p_i, then the modality of P is $\max_{i=1}^{N} |M_i|$. Avis et al. [27] show there exist polygons for which the *total modality* $\sum_{i=1}^{N} |M_i| = \Theta(N^2)$, so we cannot hope for a space-efficient data structure that stores the local maxima for all vertices.

Reciprocal modality. The main observation on which our solution to the section–section problem is based is that, given two sections S_a and S_b, in case we know that a point $p \in S_a$ is *not* a local maximum in the distance sequence of point $q \in S_b$, the distance $d(p, q)$ cannot be the maximum distance between S_a and S_b unless p is either the first or the last point in S_a, because otherwise both neighbours of p in P are in the query range and the distance from q to one of those neighbours is larger than $d(p, q)$. This observation implies that for any pair of sections S_a and S_b, the furthest point pair is either a *pair of reciprocal local maxima* for which both points are local maxima in each other's distance sequence, or one of the points is the first or the last in its section. The largest distance between S_a and S_b is therefore equal to the maximum of

1. the distance of the furthest pair of reciprocal local maxima of which one point is in S_a and the other point is in S_b, and
2. the distance from the first/last point in S_b (S_a) to the furthest point in S_a (S_b).

The furthest pair of reciprocal local maxima can be found in the following way. Let Q be a set containing a point (i, j) for each pair of reciprocal local maxima (p_i, p_j), that is, $Q := \{(i, j) \mid j \in M_i \text{ and } i \in M_j\}$. To each point (i, j) we assign a weight of $d(p_i, p_j)$. We create a two-dimensional range-maximum data structure over Q to be able to efficiently find the point with maximum weight within a given orthogonal query range, if it exists. Let f_a be the index of the first point in S_a and l_a the index of the last point in S_a. Note that the points in Q inside a range $[f_a : l_a] \times [f_b : l_b]$ (for $f_a \leq l_a$ and $f_b \leq l_b$) represent the pairs of reciprocal local maxima between S_a and S_b, so the point in this range with the highest weight corresponds to the furthest pair of reciprocal local maxima between S_a and S_b. In case $f_a > l_a$ or $f_b > l_b$, we take the maximum of two or four queries covering the whole query range.

To bound the amount of space necessary for the 2D range-maximum data structure, we show that the *reciprocal modality* $|Q| = O(N)$.

Lemma 7.2 *The reciprocal modality of the vertex set of any convex polygon* $P = (p_1, p_2, \ldots, p_N)$ *is* $O(N)$.

Proof. We show that any pair p, q of reciprocal local maxima is also an *antipodal pair*, that is, there exist parallel lines through p and q such that all other points of P lie between these lines. As the number of antipodal pairs is linear [107, Section 4.2.3], the number of reciprocal local maxima, and hence the reciprocal modality, is $O(N)$.

Consider a pair p, q of reciprocal local maxima and draw parallel lines ℓ_p and ℓ_q through p and q, orthogonal to a line through both vertices (see Figure 7.4).

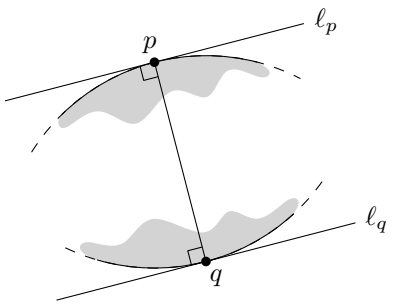

Figure 7.4: If p and q are reciprocal local maxima, they also form an antipodal pair.

As the vertices neighbouring p and q in the polygon must be closer to q and p, respectively, they must also be between ℓ_p and ℓ_q. By convexity, all points of P must be between ℓ_p and ℓ_q, so p and q form an antipodal pair. □

Point–section queries. Finally, we need a data structure for answering point–section queries. For this problem we can use a data structure of Aronov et al. [26] that uses $O(N \log^3 N)$ space to answer queries of the following type in $O(\log N)$ time: For a point q in the plane and a section of the polygon, find the point in this section furthest away from q. In our case q is always a vertex of the polygon, allowing us to design a data structure that uses less space.

Lemma 7.3 *The vertices of a convex polygon $P = (p_1, p_2, \ldots, p_N)$ can be preprocessed into a data structure of size $O(N \log N)$ such that queries of the following type can be answered in $O(\log N)$ time: given three indices i, j and k such that $j \leq k$, find the furthest point from p_i in the range $(p_j, p_{j+1}, \ldots, p_k)$.*

Proof. Our structure is a two-level structure, where the first level consists of a balanced binary search tree on the indices of the vertices of P with every vertex represented by a leaf. For every node v of the tree, let $P(v)$ denote the canonical set of v, that is, the set of vertices in the subtree rooted at v. Let $S(v, x)$ be the set of vertices $z \in P$ for which $x \in P(v)$ is the furthest vertex among all vertices in $P(v)$, that is, $S(v, x) := \{z \in P \mid x = \arg\max_{y \in P(v)} d(z, y)\}$ (see also Figure 7.5). Because each $S(v, x)$ forms a consecutive subsequence of P we can store a list $L(v)$ of the indices of the first vertex in $S(v, x)$ for each x as a second-level data structure for each node v. This requires $O(N \log N)$ space in total.

Queries for indices i, j and k can be answered as follows. Find the $O(\log N)$ nodes whose canonical sets together cover $(p_j, p_{j+1}, \ldots, p_k)$, but whose parents contain vertices outside the range. For each node v found in this way, do a binary search in $L(v)$ for i to obtain the furthest point from p_i among $P(v)$. By taking the maximum distance obtained from all these nodes we get the answer to the query in $O(\log^2 N)$ time. Since we search for the same value i in all $O(\log N)$ lists, we can apply fractional cascading to obtain $O(\log N)$ query time. □

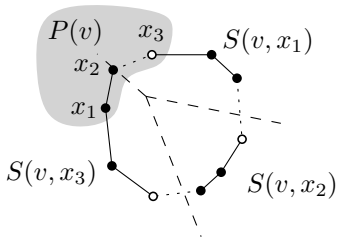

Figure 7.5: Example subdivision of P into sets of points that have the same vertex in $P(v)$ as their furthest point, where the dashed diagram shows the regions of the plane with the same furthest point among the vertices of $P(v)$. The indices of the white vertices are saved in $L(v)$.

Alternatively, we can use a linear space data structure with higher query time.

Lemma 7.4 *The vertices of a convex polygon $P = (p_1, p_2, \ldots, p_N)$ can be preprocessed into a data structure of size $O(N)$ such that queries of the following type can be answered in $O(\log^2 N)$ time: given three indices i, j and k such that $j \leq k$, find the furthest point from p_i in the range $(p_j, p_{j+1}, \ldots, p_k)$.*

Proof. Build a range tree as for Lemma 7.3, but without the second level. Instead, we add a secondary key to each node v to support searching for the furthest point from query point p_i inside $P(v)$. The secondary key represents the range of vertices of P whose furthest point is in the left subtree of v.

For answering a query, we again find the $O(\log N)$ nodes whose canonical sets together cover $(p_j, p_{j+1}, \ldots, p_k)$. For each such node v, find the furthest point from p_i among $P(v)$ using the secondary keys. By taking the maximum distance obtained from all these nodes we can answer the query in $O(\log^2 N)$ time. □

Now that we have described all necessary components, we can put them together to obtain the main result of this section.

Theorem 7.4 *Given a convex point set, we can construct (1) in $O(N \log N)$ time an $O(N \log N)$ space data structure that answers range diameter queries in $O(\log N)$ time, or (2) an $O(N \log^{\varepsilon} N)$ space data structure with $O(\log^2 N)$ query time in the word-RAM model.*

Proof. The predecessor structures of Lemma 7.1 use $O(N)$ space and take $O(\log N)$ time per query, and can be constructed in $O(N \log N)$ time. We construct set Q, containing pairs of indices of reciprocal local maxima, in $O(N)$ time by inspecting all antipodal pairs, which can be enumerated in linear time [107]. For result (1) we store Q in the 2D range maximum data structure of Gabow et al. [66], which answers queries in $O(\log N)$ time using $O(N \log N)$ space, and can be constructed in $O(N \log N)$ time. For result (2) we use a data structure of Chan et al. [44] that has $O(\log \log N)$ query time using $O(N \log^{\varepsilon} N)$ space in the word-RAM model.

Point–section queries are answered by constructing in $O(N \log N)$ time, (1) the data structure of Lemma 7.3, using $O(N \log N)$ space and $O(\log N)$ query

time, or (2) the data structure of Lemma 7.4, using $O(N)$ space and $O(\log^2 N)$ query time.

As described, range maximum queries can be answered using a constant number of queries on these data structures. □

7.2.3 Range width

Recall that the width of a set of points in the plane is the smallest distance between any two parallel lines enclosing all points. For sets of points in convex position, it is easy to show that these lines are always incident to antipodal pairs, suggesting that we can follow a similar approach to answering range width queries in convex polygons as we did for answering range diameter queries. We now sketch how to build on the techniques described above to develop such a data structure.

We follow the same structure and first split the query into sections. For each pair of sections, we find the closest antipodal pair using a 2D range minimum data structure on the indices of the vertices forming antipodal pairs. We then only need to show how to answer point–section queries. The main difference between the two problems is that for range width, we cannot answer point–section queries in isolation: a valid pair of points and incident parallel lines may exist for a given point and section, while no parallel lines through these points exist that enclose all other points. Therefore, we shrink the section to only include points that allow valid parallel lines. As a preprocessing step, we use the rotating calipers algorithm to find for each edge of the polygon the one or two vertices that are furthest away from the line through that edge [107]. For a point–section query on indices i, j, and k, we first find the vertices $p_{j'}$ and $p_{k'}$ opposite edges (p_i, p_{i+1}) and (p_{i-1}, p_i) (where $p_0 := p_N$ and $p_{N+1} := p_1$), taking the vertices that are furthest apart in case of parallel edges. Then, we search for the closest point to p_i within the range $[j : k] \cap [j' : k']$.

Chapter 8

Near-Optimal Range Reporting Structures for Categorical Data

In this chapter, we present significant improvements for the three-sided categorical range reporting problem. As mentioned earlier, we provide optimal word-RAM and near-optimal I/O-model data structures for categorical range reporting when the query rectangles are unbounded in one direction, i.e. they are of the form $[x_0 : x_1] \times (-\infty : y]$. In the literature, three-sided range queries have received perhaps even more attention than the general four-sided problem. This is owed mainly to two things: It is the "hardest" range searching problem that still admits linear space solutions with less than polynomial query time, and secondly, almost all solutions for the general problem use data structures for three-sided queries as building blocks. Indeed, using our three-sided data structures as black boxes, we also obtain improved word-RAM and I/O-model data structures for the general problem.

Our contribution

Our main results are near-optimal data structures for three-sided categorical range reporting in the I/O-model and optimal data structures in the word-RAM. More specifically, in Sections 8.1 and 8.2 we show that for any integer $h \geq 1$ (not necessarily constant), one can obtain a data structure for three-sided queries that uses $O(Nh)$ space and answers queries in $O(\log_B N + \log^{(h)} N + K/B)$ I/Os. At one extreme of the tradeoff, we get that for any constant integer $h \geq 1$, there exists a linear space data structure answering queries in $O(\log_B N + \log^{(h)} N + K/B)$ I/Os. At the other extreme, we get a data structure using $O(N \log^* N)$ space and answering queries in $O(\log_B N + K/B)$ I/Os. Note that the previous fastest I/O-model data structure with linear space usage answers queries in $O((N/B)^\varepsilon + K/B)$ I/Os! The best previous data structure with optimal query time uses $O(N \log\log N \log\log\log N)$ space.

In Section 8.3, we extend our I/O-model results to the case where points have coordinates on a $U \times U$ grid. In this case, we reduce the $\log_B N$ term in the query bounds to an optimal $\log\log_B U$ term. Finally, for points on an $N \times N$ grid, we reduce it all the way to $O(1)$. This again improves over all the previous best tradeoffs.

For the word-RAM, we present in Section 8.4 an optimal $O(N)$ space and $O(\log\log U + K)$ query time data structure when points have coordinates on a $U \times U$ grid. We improve the query time further to $O(1 + K)$ when the points are on an $N \times N$ grid. The best previous linear space data structure in the word-RAM is in fact the pointer machine result with $O(\log N + K)$ query time. The best previous result with optimal $O(\log\log U + K)$ query time is the $O(N \log\log N \log\log N)$ space data structure of Nekrich.

Using a standard reduction, all these results (except for the $N \times N$ grid) extend to four-sided queries at the cost of increasing the space by a $\log N$ factor. This improves over all the previous tradeoffs in both models.

Finally, in Section 8.5, we establish a tight lower bound for one-dimensional categorical range counting. This lower bound is established by giving an elegant reduction from (standard non-colored) two-dimensional range counting to one-dimensional categorical range counting. Not only does this establish a tight $\Omega(\log N/\log\log N)$ lower bound on the query time of any $N \log^{O(1)} N$ space word-RAM data structure [104], but combined with the previous reduction from one-dimensional categorical range counting to two-dimensional standard range counting [70], it leads to the very surprising conclusion that the two problems are identical!

Note on duplication of colors

In previous work, categorical range reporting data structures were allowed to report the same color from within a query range a constant number of times, and not necessarily just once (note that since it is a constant number of times, the $O(K)$ or $O(K/B)$ term in the reporting time remains unchanged). This assumption can easily be removed in the word-RAM by removing duplicates in the output either by using an array indexed by color or hashing. The assumption is a little more questionable in the I/O-model, but we still make it and note that this was also assumed in the previous I/O-model results.

8.1 Three-sided categorical range reporting

In this section we present a linear space data structure for answering three-sided categorical range queries in $O(\log(N/B) + K/B)$ I/Os based on ideas from the internal-memory data structures by Mortensen [98] and Shi and JaJa [113]. This is not yet the query time we are aiming for, but the data structure will be used as a building block for our final data structure in the next section.

In Section 8.1.1 we first describe a standard reduction of three-sided categorical range reporting to offline partially-persistent *one-dimensional* categorical range reporting. Then, we show how to solve this one-dimensional problem by dividing the points into buckets of size B and building a binary tree of height $O(\log(N/B))$ over the buckets. Following [98] and [113], we then assign a y-coordinate representing a level in the binary tree to each one-dimensional point. We show that a one-dimensional range query on the original points can be answered using two three-sided non-colored range queries on two transformed point sets that have only $O(\log(N/B))$ different y-coordinates.

In Section 8.1.2 we begin by constructing a simple partially-persistent one-dimensional data structure, which can be used for each of the $O(\log(N/B))$ y-coordinates in the above reduction. This data structure supports range queries in $O(1 + K/B)$ I/Os when given a pointer to the disk block storing the leftmost point inside a query range. To find these leftmost points for each of the $O(\log(N/B))$ one-dimensional data structures, we construct a rectangular subdivision of the plane for each of them, and show that a point location query can be used to find the desired leftmost points. We overlay all $O(\log(N/B))$ rectangular subdivisions and use a known rectangle-stabbing data structure with query complexity $O(\log(N/B) + K/B)$ I/Os to perform all $O(\log(N/B))$ point locations simultaneously. Note that the output size is only $O(\log(N/B))$, so the total cost of this rectangle-stabbing query is $O(\log(N/B))$ I/Os.

8.1.1 Reduction to partially-persistent three-sided non-colored range reporting

The three-sided categorical range reporting problem can be solved using a data structure for *offline insertion-only partially-persistent one-dimensional categorical range reporting* [70]. Such a data structure processes a sequence of N insertions of one-dimensional colored points with increasing timestamps, which we think of as the insertion times. Given a query interval $[x_0 : x_1]$ and a timestamp t, the data structure reports all colors contained in the interval $[x_0 : x_1]$ as if only updates with timestamp at most t were performed.

The three-sided categorical range reporting problem can be solved using such a data structure by applying the following reduction: Given the N colored input points in $\mathbb{R}^2$, sweep a horizontal line from $y = -\infty$ towards ∞ and insert points (x, y) into the offline one-dimensional categorical range reporting data structure when they are hit by the sweep line, i.e. we map each point (x, y) to the one-dimensional point with coordinate x and insertion timestamp y. A query $[x_0 : x_1] \times (-\infty : y_0]$ now translates into the one-dimensional query $[x_0 : x_1]$ with the query timestamp set to y.

Offline one-dimensional insertion-only partially-persistent categorical range reporting. We now show how to solve offline one-dimensional insertion-only partially-persistent categorical range reporting using a data structure for partially-persistent two-dimensional three-sided non-colored range reporting where the number of different y-coordinates is $O(\log(N/B))$.

We first divide the set of x-coordinates of all input points into N/B buckets of size B and the points in each bucket are stored together with their insertion timestamps in $O(1)$ disk blocks. Then we conceptually build a balanced binary tree of height $O(\log(N/B))$ with these buckets as leaves. Let $b(p)$ denote the bucket containing a point p, and $h(v)$ the height of node v in the tree, counting from the leaves up, so the leaves have height 0.

Now process the input points in order of insertion time, and let p with color χ be any inserted point for which there is no χ-colored point right of p in $b(p)$ (at the time of the insertion). We maintain a value $y_l(p)$, which is the height $h(a)$ of the lowest ancestor a of $b(p)$ for which there are no inserted points with color χ

right of p in the range covered by the subtree rooted at a (see Figure 8.1(a)). At any step during the insertions, we let P_l denote the two-dimensional point set consisting of the set of currently inserted one-dimensional points that are the rightmost in their bucket. A point p is represented in P_l as the two-dimensional point with the same x-coordinate as p, and with y-coordinate $y_l(p)$.

When inserting a point p, it might be that there was already at least one point with color χ in the subtree rooted at a. If this is the case, we find the rightmost point p' amongst all points in the subtree rooted at a with color χ. We then find the lowest common ancestor a' of p and p', delete p' from P_l, and in case $b(p') \neq b(p)$ we re-insert p' with y-coordinate $h(a') - 1$. This operation re-establishes the representation of each point in P_l, i.e. any point p which is the rightmost of its color inside $b(p)$, is represented by $(x, y_l(p))$ in P_l, where x is the x-coordinate of p.

The point set P_l can thus be maintained with at most one insertion and one deletion for each one-dimensional point that is inserted. We now store P_l in a partially-persistent data structure for answering three-sided range queries (no colors, but insertions and deletions), but where the number of different y-coordinates is only $O(\log(N/B))$. The partial persistence allows us to query P_l as it was after processing only the one-dimensional insertions with timestamp at most t, for any t. This partially-persistent structure is described in Section 8.1.2.

To answer a query $[x_0 : x_1]$ at timestamp t, we first find the lowest common ancestor a of x_0 and x_1. In case $b(x_0) = b(x_1)$ we spend O(1) I/Os to read this bucket from disk and simply solve the range reporting problem in internal memory at no I/O costs. Otherwise, we answer the query in two parts (so we may report colors twice). Let c_0 be the child of a containing $b(x_0)$ in its subtree and let c_1 be the child containing $b(x_1)$ in its subtree. A *left query* is used for finding the relevant colors in the subtree rooted c_0 and a *right query* is used for finding the relevant colors in the subtree rooted at child c_1 (see Figure 8.1(b)).

For answering left queries, we let x_{m} be the largest x-coordinate of a point in the subtree rooted at c_0, where we ignore insertion time (this value can be stored at c_0 as a preprocessing step). We then use a data structure for partially-persistent (insertion and deletion) three-sided range reporting to report all points $(x, y) \in P_l$ for which $x_0 \leq x \leq x_{\mathrm{m}}$ and $y \geq h(a) - 1$. Note that in case a point with color χ is present in the range $[x_0 : x_{\mathrm{m}}]$, either the point is the rightmost with color χ in the subtree of c_0, in which case its height is at least $h(c_0) = h(a) - 1$, or otherwise, there is a representative of the same color with a higher x-coordinate and height at least $h(a) - 1$. Moreover, no two points with the same color in the subtree of c_0 can have height at least $h(a) - 1$, so for all colors in the range $[x_0 : x_{\mathrm{m}}]$, there is exactly one point of that color which is reported by the query on P_l.

For answering right queries we define $y_r(p)$ analogously to $y_l(p)$, that is, $y_r(p)$ is the height of the lowest ancestor such that p is the leftmost point of color χ in the subtree. Similarly, P_r is defined as P_l, with rightmost replaced by leftmost and $y_l(p)$ replaced by $y_r(p)$. The query is also analogous, we find all points $(x, y) \in P_r$ for which $x_{\mathrm{m}} < x \leq x_1$ and $y \geq h(a) - 1$.

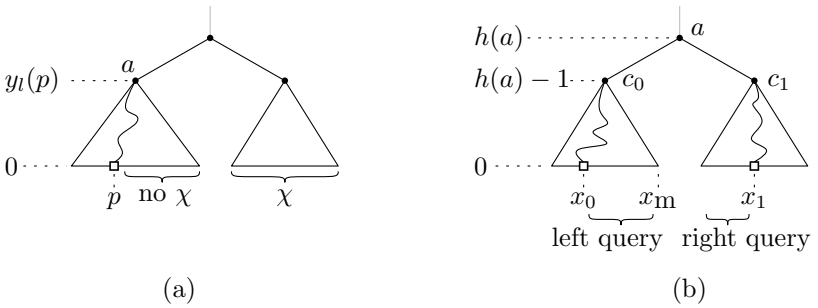

Figure 8.1: (a) Assignment of y coordinates. (b) Answering queries.

8.1.2 Offline partially-persistent three-sided non-colored range reporting

As the number of different y-coordinates is $O(\log(N/B))$, we can afford to do an I/O for each of them. Thus, if we can construct a one-dimensional partially-persistent data structure with query time $O(1 + K/B)$ for each y-coordinate, we have solved the problem (we just query the data structures stored for all y-coordinates inside the query range).

For the one-dimensional reporting problems, we simply construct a partially persistent B-tree [31] over the points. This partially-persistent B-tree stores points in sorted order in the leaves, and we simply augment the leaves with pointers between them. These pointers can easily be maintained partially-persistent with only a constant factor overhead. Thus if we are given a pointer to the leaf containing x_0 at time t, we can report the K points in a range $[x_0 : x_1]$ in $O(1 + K/B)$ I/Os simply by scanning leaves until a point with x-coordinate greater than x_1 is encountered. Our goal is therefore to find such a pointer for all $O(\log(N/B))$ different y-coordinates, using only $O(\log(N/B))$ I/Os.

Reduction to Partially-Persistent Rectangle Stabbing. Consider one of the partially persistent B-trees stored for a particular y-coordinate. Each insertion and deletion into such a tree changes $O(1)$ leaves amortized. We thus represent each version of each leaf as a rectangle $[x_0 : x_1] \times [t_0 : t_1]$, where $[x_0 : x_1]$ is the range of coordinates represented by the leaf after the update (starting from but excluding the last point in the previous leaf, or $-\infty$ for the first leaf), t_0 is the timestamp of the update and t_1 is the timestamp of the next update to the leaf. To find which leaf contains x at timestamp t, we need to find the rectangle containing the point (x, t), that is, we need to answer a *point location query* in a set of disjoint axis-aligned rectangles. The total number of rectangles is equal to the total number of leaf changes in the partially persistent B-tree, which is linear in the total number of updates, N.

We superimpose the rectangular subdivisions for all $O(\log(N/B))$ different partially persistent B-trees and store them in a linear space rectangle-stabbing data structure of Arge et al. [20], in which each rectangle is augmented with a pointer to the leaf it represents. This data structure answers rectangle-stabbing queries (report all K rectangles containing a query point) in $O(\log(N/B) +$

K/B) I/Os. Note that we report at most $O(\log(N/B))$ rectangles, so we have the following theorem. We note that the idea of solving a number of point location queries in parallel using a rectangle-stabbing query has appeared before in the work of Afshani et al. [2].

Theorem 8.1 *There exists a data structure for three-sided categorical range reporting that answers queries in $O(\log(N/B) + K/B)$ I/Os using linear space.*

8.2 Final data structure

In this section, we show that using the data structure from Theorem 8.1 as a building block, we can obtain a data structure that uses $O(Nh)$ space and answers queries in $O(\log_B N + \log^{(h)}(N/B) + K/B)$ I/Os for any integer $h > 1$ (not necessarily constant). Before presenting our solution, we need the following easy result for two-sided categorical range reporting:

Lemma 8.1 *There exists a data structure for two-sided categorical range reporting, using linear space and answering queries in optimal $O(\log_B N + K/B)$ I/Os.*

Proof. Assume the queries are unbounded towards $-\infty$ in both the x- and y-direction. To solve the problem, we maintain a partial persistent B-tree $\mathcal{T}$, which is initially empty. Now conceptually sweep a vertical line from $x = -\infty$ to ∞, and whenever a point (x, y) from the input intersects the sweep line, check if it has the lowest y-coordinate amongst all points of the same color which have been encountered so far. If so, we first delete the previous lowest point of the same color (if any) from $\mathcal{T}$ and set the timestamp of the deletion to $2x - 1$. We then insert the newly encountered point with key y and timestamp $2x$. To answer a query $(-\infty, x_0] \times (-\infty, y_0]$ we simply ask the (non-colored) range reporting query $(-\infty, y_0]$ on $\mathcal{T}$ with the timestamp being $2x_0$. The correctness follows immediately. □

With Theorem 8.1 and Lemma 8.1 established, we are ready to bootstrap with these two solutions to obtain an even faster solution for three-sided queries.

8.2.1 Bootstrapping

In the following, we use the data structure of Theorem 8.1 to reduce the query time even further. Our key idea is to only query the data structure from Theorem 8.1 in case we know we have $\Omega(B \log(N/B))$ points to report. If we have fewer points to report, then we can reduce the problem size we are working on essentially by an exponential factor. The details follow below.

Our final data structure takes as input an integer parameter $h \geq 1$ (not necessarily constant). If $h = 1$, we simply store the data structure of Theorem 8.1 which has space usage $O(Nh)$ and query time $O(\log^{(1)}(N/B) + K/B)$. In this case, we say that the data structure is a *leaf* data structure.

If $h > 1$, we do the following: First of all, we store all input points in the data structure of Theorem 8.1, which we will query in case we are sure

we have $\Omega(B \log(N/B))$ colors to report. Second, we sort the input points by x-coordinate and divide them into buckets of $B \log^2(N/B)$ consecutive points each. For each bucket, we implement the data structure from Lemma 8.1 such that we can answer two-sided queries on the points in a bucket. This uses linear space.

We then conceptually construct a complete binary search tree on the buckets and note that we can identify the lowest common ancestor of two buckets solely from the indices of the two. For a bucket b, we now store for each ancestor v in the binary search tree two lists of points, $S_\ell(b, v)$ and $S_r(b, v)$, as defined below. Intuitively, $S_\ell(b, v)$ contains a set of $O(B \log(N/B))$ points with unique colors in the subtrees hanging off to the left of the path from b to v (see Figure 8.2), and similarly $S_r(b, v)$ contains points from the subtrees hanging off to the right of the path from b to v.

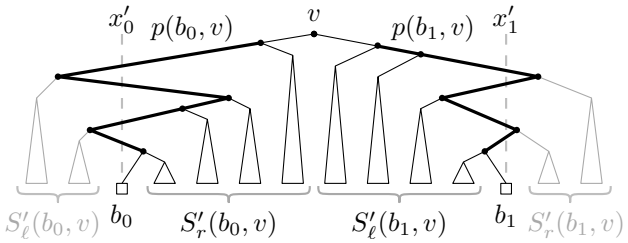

Figure 8.2: The answer to a query with x-range $[x_0 : x_1]$ is contained in $S'_r(b_0, v)$,$S'_\ell(b_1, v)$, b_0 and b_1.

More formally, let v be an ancestor of bucket b and $p(b, v)$ the path from b to v (excluding b and v). Now construct the set $S'_\ell(b, v)$ of all points in subtrees hanging off to the left of $p(b, v)$, and similarly $S'_r(b, v)$ the set of points in subtrees hanging off to the right of $p(b, v)$ (see Figure 8.2). For each b and v, let $S''_\ell(b, v)$ be the set points that each have the smallest y-coordinate amongst the points of the same color in $S'_\ell(b, v)$, i.e.

$$S''_\ell(b, v) := \{q \in S'_\ell(b, v) \mid \forall q' \in S'_\ell(b, v) : \chi(q') = \chi(q) \Rightarrow y(q') > y(q)\}.$$

Define $S''_r(b, v)$ symmetrically, with l replaced by r. Then, select the $B \log(N/B)$ points with lowest y-coordinate in $S''_\ell(b, v)$ to form the set $S_\ell(b, v)$ (do the same for $S_r(b, v)$). For a bucket b, we store $S_\ell(b, v)$ and $S_r(b, v)$ for each ancestor v as a list of points ordered by y-coordinate. As we have $O(N/B \log^2(N/B))$ buckets and $O(B \log(N/B))$ points for each of the $O(\log(N/B))$ ancestors of a bucket, we need $O(N/B \log^2(N/B) \cdot B \log^2(N/B)) = O(N)$ space for storing these lists.

Finally, we store the same data structure recursively on each bucket, but with h decreased by 1. Note that the ratio between the number of points and disk block size goes from N/B to $\log^2(N/B)$ in the recursively constructed data structures, and when the recursion bottoms out (h decreases to 1), the ratio has decreased to $O((\log^{(h-1)}(N/B))^2)$.

Answering Queries. To answer a query $[x_0 : x_1] \times (-\infty : y_0]$, first determine the successor of x_0, denoted $s(x_0)$, and the predecessor of x_1, denoted

$p(x_1)$, amongst the x-coordinates of the input points, as well as their respective ranks. This can be done by storing one global B-tree on the x-coordinates of all input points. We first determine whether $s(x_0)$ and $p(x_1)$ lie in the same leaf data structure. Since our recursive data structures are constructed as complete binary trees on the x-coordinates of the points, this can be determined solely from the ranks of $s(x_0)$ and $p(x_1)$, i.e. no I/Os are needed. If they are in the same leaf data structure, we simply answer the query using the data structure of Theorem 8.1 stored at the leaf.

If $s(x_0)$ and $p(x_1)$ are not in the same leaf data structure, we determine the first node, v, in all the recursively constructed binary trees, for which $s(x_0)$ and $p(x_1)$ lie in different subtrees. Let $\mathcal{T}_v$ be the complete binary tree containing v and let $1 < h' \leq h$ denote the value of h used when constructing $\mathcal{T}_v$. The number of points in $\mathcal{T}_v$ is $O((\log^{(h'-1)}(N/B))^2 B)$. We now find the bucket $b_0 \in \mathcal{T}_v$ containing $s(x_0)$ and $b_1 \in \mathcal{T}_v$ containing $p(x_1)$. Since $s(x_0)$ and $p(x_1)$ lie in different subtrees of v, we have $b_0 \neq b_1$. We now answer the query in at most four parts, so each color may be reported up to four times. We first report the colors of the points in b_0 and b_1 that intersect the query range using the two-sided data structures of Lemma 8.1 stored over all points in those buckets, where in b_0 we use the query range $[x_0, \infty) \times (-\infty, y_0]$ and in b_1 we use the query range $(-\infty, x_1] \times (-\infty, y_0]$. Then, we scan $S_r(b_0, v)$ and $S_\ell(b_1, v)$, starting from the point with smallest y-coordinate and reporting the colors of points until their y-coordinate is greater than y_0. In case the two lists both return less than all of their points, we are sure to have reported all colors in the range since any other colors either do not have points in the subtrees hanging off to the right (left) of the path from v to b_0 (v to b_1), or the points in there have y-coordinates greater than y_0 (otherwise they would have been represented in $S_r(b_0, v)$ or $S_\ell(b_1, v)$). In case all points are reported from one of the two lists, we discard the output so far and instead query the data structure of Theorem 8.1 containing all points in T_v. Since the output size is $K = \Omega(\min\{|S_r(b_0, v)|, |S_\ell(b_1, v)|\}) = \Omega(B \log((\log^{(h'-1)}(N/B))^2)) = \Omega(B \log^{(h')}(N/B))$ in this case, this costs $O(\log((\log^{(h'-1)}(N/B))^2) + K/B) = O(\log^{(h')}(N/B) + K/B) = O(K/B)$ I/Os.

Analysis. We first analyze the query time. Determining the successor, $s(x_0)$, of x_0 and predecessor, $p(x_1)$, of x_1, including their ranks, costs $O(\log_B N)$ I/Os. If $s(x_0)$ and $p(x_1)$ are in the same leaf data structure, we query the data structure of Theorem 8.1 on a set of $O((\log^{(h-1)}(N/B))^2 B)$ points, costing $O(\log^{(h)}(N/B) + K/B)$ I/Os. If $s(x_0)$ and $p(x_1)$ are not in the same leaf data structure, we query two two-sided data structures of Lemma 8.1, both stored on $O(N)$ points, costing $O(\log_B N + K/B)$ I/Os. Scanning the two lists $S_r(b_0, v)$ and $S_\ell(b_1, v)$ costs $O(K/B)$ I/Os since we only continue scanning while we report new colors. In case we report all points from one list, we spend another $O(K/B)$ I/Os querying the three-sided data structure stored on T_v. Thus the total query cost is $O(\log_B N + \log^{(h)}(N/B) + K/B)$ I/Os as claimed. For the space usage, observe that each recursive level uses linear space, thus the space is bounded by $O(Nh)$.

Theorem 8.2 *For any integer parameter $h \geq 1$ (not necessarily constant), there exists an $O(Nh)$ space data structure answering three-sided categorical range queries in $O(\log_B N + \log^{(h)}(N/B) + K/B)$ I/Os, where K is the number of colors reported.*

Four-sided queries. The three-sided data structure above also provides an improved data structure for four-sided queries: Simply use the binary range trees [32] to obtain a data structure for four-sided queries, at the cost of increasing the space by a $\log N$ factor.

8.3 Input points on a grid

In this section, we show how to reduce the query time of our I/O-model data structures when the input points are given on a grid. Our aim is to reduce the $\log_B N$ term and from the analysis in Section 8.2, we see that this term arises from two places:

- Finding the successor of x_0 and predecessor of x_1, including their ranks.
- Querying a two-sided data structure from Lemma 8.1 on a bucket.

The first part is easily dealt with since predecessor search in an integer universe of size U can be done in $O(\log\log_B U)$ I/Os and in $O(1)$ I/Os in a universe of size N (simply store an array). For the two-sided structure from Lemma 8.1, recall that we are solving this problem by asking one-sided queries on a partial persistent B-tree, i.e. queries of the form $(-\infty, y_0]$ at some timestamp x_0. It is easily seen that a partially-persistent B-tree can be implemented such that if we are given a pointer to the disk block containing the smallest element stored in a partially-persistent tree at a given timestamp x_0, then we can immediately jump to that disk block and scan points in the tree until the keys exceed y_0, i.e. we can answer the query in $O(1 + K/B)$ I/Os if given such a pointer. Now if we augment the partially-persistent B-tree structure with a sorted list of pointers, one for each operation performed on the structure, where each pointer points to the disk block containing the smallest element in the tree after the corresponding update operation, then we only need to determine where the update operation immediately preceding the query timestamp x_0 is located in this sorted array. This is a predecessor search problem. Now observe that the timestamps used in our solution is amongst $[2U]$ when the points are on a $U \times U$ grid, hence we conclude

Corollary 8.1 *For any integer parameter $h \geq 1$ (not necessarily constant), there exists an $O(Nh)$ space data structure answering three-sided categorical range queries in $O(\log\log_B U + \log^{(h)}(N/B) + K/B)$ I/Os when the N input points are given on a $U \times U$ grid, where K is the number of colors reported.*

Corollary 8.2 *For any integer parameter $h \geq 1$ (not necessarily constant), there exists an $O(Nh)$ space data structure answering three-sided categorical range queries in $O(\log^{(h)}(N/B) + K/B)$ I/Os when the N input points are given on an $N \times N$ grid, where K is the number of colors reported.*

8.4 Word-RAM data structures

In the following, we show how to extend our I/O-model solutions to obtain optimal solutions in the word-RAM. We first note that the I/O-model data structure of Theorem 8.1 translates directly to the following word-RAM result:

Corollary 8.3 *There exists a word-RAM data structure for three-sided categorical range reporting, using linear space and answering queries in $O(\log N + K)$ I/Os.*

$U \times U$ grid in the word-RAM. To obtain optimal query times in the word-RAM, we bootstrap with the data structure from Corollary 8.3 by using the same layout as Section 8.2, i.e. we partition the points into buckets of size $\log^2 N$ and construct a complete binary tree on the buckets. For the buckets, we do not recurse but instead store the word-RAM data structure of Corollary 8.3 directly. Since the number of points in a bucket is $\log^2 N$, the cost of querying such a data structure is $O(\log(\log^2 N) + K) = O(\log \log N + K)$. For the lists $S_r(b, v)$ and $S_\ell(b, v)$, we use the topmost $\log N$ points and hence the space usage is linear.

When coordinates lie on a $U \times U$ grid, we can find the successor (and rank) of x_0 and the predecessor (and rank) of x_1 in $O(\log \log U)$ time using a predecessor-search data structure [106]. The lowest common ancestor v and the lists $S_r(b_0, v)$ and $S_\ell(b_1, v)$ can be found in constant time by word-operations and indexing in the word-RAM, so the total query time is $O(\log \log U + K)$.

Theorem 8.3 *There exists a linear space data structure answering three-sided categorical range queries on a set of points in $[U] \times [U]$ in $O(\log \log U + K)$ time in the word-RAM, where K is the number of colors reported.*

$N \times N$ grid in the word-RAM. For input points on an $N \times N$ grid (i.e. rank space), we bootstrap with the data structure of Theorem 8.3, which we query in case we have $\Omega(\log \log N)$ colors to report. We again construct the layout of Section 8.2, i.e. we partition into buckets and construct a complete binary tree on them. This time the buckets contain $\log N \log \log N$ points each and the size of each $S_\ell(b, v)$ and $S_r(b, v)$ is $\log \log N$, so we use $O(N/(\log N \log \log N) \cdot \log N \cdot \log \log N) = O(N)$ space for storing the $S_\ell(b, v)$ and $S_r(b, v)$ lists. We do not recurse on the buckets, but instead we store a data structure that answers three-sided queries on $O(\log N \log \log N)$ points in $O(1+K)$ time and with linear space. This data structure is described in Section 8.4.1. This data structure also uses a lookup table of size $O(N^\varepsilon)$, where $\varepsilon > 0$ is an arbitrarily small constant, which can be shared among all buckets.

We query the data structure as in Section 8.2, except we use the $O(1 + K)$ query time data structure for handling the buckets. The elements $s(x_0)$ and $p(x_1)$ and their ranks can be found in constant time by storing an array over all x-coordinates. Hence, in this case the query time is $O(1 + K)$.

Theorem 8.4 *There exists a linear space data structure answering three-sided categorical range queries on a set of N points in $[N] \times [N]$ in $O(1+K)$ time in the word-RAM, where K is the number of colors reported.*

8.4.1 Word-RAM queries on few points

In this section, we show how to answer three-sided categorical range reporting queries on a set P of $m = O(\log N \log\log N)$ points with coordinates on an $N \times N$ grid. Our solution uses a lookup table of size $O(N^\varepsilon)$, where $\varepsilon > 0$ is an arbitrarily small constant. This lookup table is independent of the input set and hence can be shared amongst any number of data structures. In addition, our solution uses linear space (i.e. $O(m \log N)$ bits) and answers queries in $O(1+K)$ time. The techniques used are standard.

First we store a Fusion tree [61] on the m input points. This Fusion tree uses linear space and allows us to answer predecessor queries amongst the input points in constant time. Thus we can map a query into rank space coordinates w.r.t. P in constant time, i.e. we can consider all points in P as lying on an $m \times m$ grid and we can assume that the corner points of a query are also represented by points on the grid. Furthermore, we replace the colors of the input points by integers in $[m]$, such that two points with the same color are assigned the same new color in $[m]$ (i.e. a rank space reduction on the colors). Note that we can trivially recover the old coordinates and colors from our new representations of points using lookup arrays of total size $O(m \log N)$ bits. We let P^* denote our transformed set of input points.

We now partition the points in P^* into m/Δ buckets of $\Delta = \delta \log N / \log\log N$ points each, where $\delta > 0$ is a sufficiently small constant. The ith bucket consists of the Δ points in P^* with x-coordinates $\{i\Delta, \ldots, (i+1)\Delta - 1\}$. Now observe that a single bucket can be completely described in $\Delta 2 \log m < 3\Delta \log\log N = 3\delta \log N$ bits, simply by writing down the y-coordinate and color of each point, one after the other. Our shared lookup table has one entry for every possible combination of a bucket (i.e. bit representation of a bucket) and a query inside that bucket. Since a three-sided query inside a bucket can be described in $3 \log m$ bits, the total number of entries in the lookup table is bounded by $2^{3\delta \log N + 4 \log\log N} = O(N^\varepsilon)$ for an arbitrarily small constant $\varepsilon > 0$. Each entry of the lookup table simply stores the answer of the corresponding query on the corresponding bucket (note that things are in rank space, but we can recover the original points and colors using the arrays mentioned earlier). The total size of the lookup table can still be bounded by $O(N^\varepsilon)$ for any constant $\varepsilon > 0$.

Thus for any query range $[x_0 : x_1] \times (-\infty : y]$ (in rank space), we can partition the query into three disjoint queries: Let i_0 be the index of the bucket containing x_0 and let i_1 be the index of the bucket containing x_1. The query is partitioned into the following parts:

1. $[x_0 : (i_0+1)\Delta - 1] \times (-\infty : y]$. This part of the query is answered using the lookup table. This takes $O(1+K)$ time.

2. $[(i_0+1)\Delta : i_1\Delta - 1] \times (-\infty : y]$. We show how to handle this part below.

3. $[i_1\Delta : x_1] \times (-\infty : y]$. This part is also answered in $O(1+K)$ time using the lookup table.

Note that if a query lies completely within a bucket, we can immediately answer the query using the lookup table. Thus all that remains is to show how we answer the query $[(i_0+1)\Delta : i_1\Delta - 1] \times (-\infty : y]$. Note that the x-range of this query completely spans a number of consecutive buckets and stops at the border between two buckets on each side. The last part of our data structure therefore consists of a sorted list for every range of buckets. For a range of buckets, we first collect all points contained in those buckets. From this set of points, we select for each color the point with lowest y-coordinate. We store this subset of points in sorted order of y-coordinate, packed into words. Now given the above query range $[(i_0+1)\Delta : i_1\Delta - 1] \times (-\infty : y]$ we start by examining the first point in the sorted list stored for the range of buckets $i_0+1, \ldots, i_1-1$. If the y-coordinate of the point is at most y, we report it and examine the next point in the list. This continues until a point with y-coordinate greater than y is encountered, or until the end of the list is reached. Clearly this correctly reports the colors in the query range and the time needed is $O(1+K)$. Since any point is stored in at most $(m/\Delta)^2 = O(\log^4 \log N)$ such lists, the total space usage is bounded by $O(m \log^4 \log N \log m) = O(m \log^5 \log N) = o(m \log N)$ bits. We conclude that the space usage of our data structure is dominated by the Fusion tree and the arrays mapping points and colors from rank space to their original coordinates. Thus the space usage is linear.

Four-sided queries. The three-sided data structure above also provides an improved data structure for four-sided queries in the word-RAM. Again we simply use range trees [32] to get the following result:

Theorem 8.5 *There exists an $O(N \log N)$ space data structure answering four-sided categorical range queries in $O(\log \log U + K)$ time in the word-RAM when the input points lie on a $U \times U$ grid. Here K is the number of colors reported.*

8.5 One-dimensional categorical range counting

In this section, we present a reduction from two-dimensional range counting to one-dimensional categorical range counting. This establishes a tight $\Omega(\log N / \log \log N)$ query time lower bound for any categorical range counting data structure in the word-RAM that uses $N \log^{O(1)} N$ space. Combined with the previous reduction in the other direction [70], this also shows that the two problems are identical.

We reduce from two-dimensional *dominance* counting, i.e. counting where the query rectangles are of the form $(-\infty : x'] \times (-\infty : y']$. Note that dominance counting solves counting with four-sided ranges by adding and subtracting the answers to a constant number of queries.

Let P be the N input points to a two-dimensional dominance range counting problem and assume all coordinates are positive (this can easily be achieved

through a translation of the points). We map each input point to two one-dimensional colored points: If an input point p has coordinates (x, y), then it is mapped to the two points with coordinates $-x$ and y, respectively. The color of the two constructed points is set to p (or some value uniquely identifying p). Letting P' denote the constructed set of $2N$ one-dimensional points, we construct a one-dimensional categorical range counting data structure on P'. Finally, we construct the set P^* consisting of all points in P', but where the colors are changed such that every point has a unique color. We also construct a one-dimensional categorical range counting data structure on P^*. Note that this corresponds to a standard one-dimensional range counting data structure since all colors are distinct.

When presented with a dominance counting query $(-\infty : x'] \times (-\infty : y']$, we ask the one-dimensional categorical range counting query $[-x' : y']$ on the two data structures constructed for P' and P^*. Letting t' and t^* denote the answers to these queries, we return as our result $t^* - t'$.

To see that we have correctly answered the dominance counting query $(-\infty : x'] \times (-\infty : y']$, let p be an input point with coordinates (x, y). We have four cases:

1. $x \leq x'$ and $y \leq y'$: The point p is inside the dominance query. In this case, observe that the one-dimensional query range $[-x' : y']$ contains both one-dimensional points representing p. Therefore, p contributes 2 to the value t^* but only 1 to the value t' (the two representatives have the same color), i.e. it contributes 1 to the returned answer.

2. $x \leq x'$ and $y > y'$: The point p is outside the dominance query. In this case, the one-dimensional query range $[-x' : y']$ contains one of the two points representing p. Thus p contributes 1 to both t^* and t' and hence 0 to the returned answer.

3. $x > x'$ and $y \leq y'$: Symmetric to case 2.

4. $x > x'$ and $y > y'$: The point p is outside the dominance query. In this case, the one-dimensional query range $[-x' : y']$ contains none of the two points representing p. Thus p contributes 0 to both t^*, t' and to the returned answer.

The correctness of our reduction from two-dimensional dominance counting to one-dimensional categorical range counting follows immediately from the above case analysis.

Bibliography

[1] M. A. Abam, P. Carmi, M. Farshi, and M. Smid. On the power of the semi-separated pair decomposition. In *Proc. 11th Workshop on Algorithms and Data Structures*, volume 5664 of *LNCS*, pages 1–12, Aug. 2009.

[2] P. Afshani, L. Arge, and K. D. Larsen. Orthogonal range reporting: Query lower bounds, optimal structures in 3d, and higher dimensional improvements. In *Proc. 26th ACM Symposium on Computational Geometry*, pages 240–246, 2010.

[3] P. K. Agarwal, L. Arge, and K. Yi. I/O-efficient construction of constrained Delaunay triangulations. In *Proc. 13th Annual European Symposium on Algorithms*, pages 355–366, Sept. 2005.

[4] P. K. Agarwal, L. Arge, and K. Yi. I/O-efficient batched union-find and its applications to terrain analysis. In *Proc. 22nd Annual Symposium on Computational Geometry*, pages 167–176, June 2006.

[5] P. K. Agarwal and J. Erickson. Geometric range searching and its relatives. In *Advances in Discrete and Computational Geometry*, volume 223 of *Contemporary Mathematics*, pages 1–56. AMS, 1999.

[6] P. K. Agarwal, S. Govindarajan, and S. Muthukrishnan. Range searching in categorical data: Colored range searching on grid. In *Proc. 10th European Symposium on Algorithms*, pages 17–28, 2002.

[7] P. K. Agarwal, S. Har-Peled, and K. R. Varadarajan. Approximating extent measures of points. *Journal of the ACM*, 51(4):606–635, 2004.

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

[9] D. Ajwani, A. Beckmann, R. Jacob, U. Meyer, and G. Moruz. On computational models for flash memory devices. In *Proc. 8th International Symposium on Experimental Algorithms*, volume 5526 of *LNCS*, pages 16–27, June 2009.

[10] L. Aleksandrov and H. Dijdjev. Linear algorithms for partitioning embedded graphs of bounded genus. *SIAM Journal on Discrete Mathematics*, 9(1):129–150, 1996.

[11] S. Alstrup, G. S. Brodal, and T. Rauhe. New data structures for orthogonal range searching. In *Proc. 41st IEEE Symposium on Foundations of Computer Science*, pages 198–207, 2000.

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

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

[14] L. Arge, G. S. Brodal, and L. Toma. On external-memory MST, SSSP and multi-way planar graph separation. *Journal of Algorithms*, 53:186–206, 2004.

[15] L. Arge, M. T. Goodrich, and F. van Walderveen. Computing betweenness centrality in external memory. Manuscript submitted for publication, 2012.

[16] L. Arge, K. G. Larsen, T. Mølhave, and F. van Walderveen. Cleaning massive sonar point clouds. In *18th ACM SIGSPATIAL GIS*, pages 152–161, Nov. 2010.

[17] L. Arge, U. Meyer, and L. Toma. External memory algorithms for diameter and all-pairs shortest-paths on sparse graphs. In *Proc. 31st International Colloquium on Automata, Languages, and Programming*, volume 3142 of *LNCS*, pages 57–74, 2004.

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

[19] L. Arge, V. Samoladas, and J. S. Vitter. On two-dimensional indexability and optimal range search indexing. In *Proc. 18th ACM Symposium on Principles of Database Systems*, pages 346–357, 1999.

[20] L. Arge, V. Samoladas, and K. Yi. Optimal external memory planar point enclosure. *Algorithmica*, 54:337–352, 2009.

[21] L. Arge and M. Thorup. RAM efficient external memory algorithms. Manuscript submitted for publication, 2012.

[22] L. Arge and L. Toma. Simplified external memory algorithms for planar DAGs. In *Proc. 9th Scandinavian Workshop on Algorithm Theory*, volume 3111 of *LNCS*, pages 493–503, 2004.

[23] L. Arge, L. Toma, and N. Zeh. I/O-efficient topological sorting of planar DAGs. In *Proc. 15th Annual Symposium on Parallel Algorithms and Architectures*, pages 85–93, 2003.

[24] L. Arge, F. van Walderveen, and N. Zeh. Multiway cycle separators and I/O-efficient planar graph algorithms. Manuscript submitted for publication, 2012.

[25] L. Arge and N. Zeh. I/O-efficient strong connectivity and depth-first search for directed planar graphs. In *Proc. 44st Annual Symposium on Foundations of Computer Science*, pages 261–270, 2003.

[26] B. Aronov, P. Bose, E. Demaine, J. I. Joachim Gudmundsson, S. Langerman, and M. Smid. Data structures for halfplane proximity queries and incremental voronoi diagrams. In *Proc. 7th Latin American Theoretical Informatics Symposium*, pages 80–92, 2006.

[27] D. Avis, G. T. Toussaint, and B. K. Bhattacharya. On the multimodality of distances in convex polygons. *Computers & Mathematics with Applications*, 8(2):153–156, 1982.

[28] G. Barequet and S. Har-Peled. Efficiently approximating the minimum-volume bounding box of a point set in three dimensions. *Journal of Algorithms*, 38(1):91–109, 2001.

[29] M. Barthélemy. Betweenness centrality in large complex networks. *The European Physical Journal B - Condensed Matter and Complex Systems*, 38:163–168, 2004.

[30] M. Beauchamp. An improved index of centrality. *Behavioral Science*, 10(2):161–163, 1965.

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

[32] J. L. Bentley. Multidimensional divide-and-conquer. *Communications of the ACM*, 23(4):214–229, 1980.

[33] S. Boccaletti, V. Latora, Y. Moreno, M. Chavez, and D.-U. Hwang. Complex networks: Structure and dynamics. *Physics Reports*, 424(4–5):175–308, 2006.

[34] B. Bollobás and O. Riordan. The critical probability for random Voronoi percolation in the plane is 1/2. *Probability Theory and Related Fields*, 136(3):417–468, 2006.

[35] P. Bozanis, N. Kitsios, C. Makris, and A. Tsakalidis. New upper bounds for generalized intersection searching problems. In *Proc. 22nd International Colloquium on Automata, Languages, and Programming*, pages 464–474, 1995.

[36] P. Bozanis, N. Kitsios, C. Makris, and A. Tsakalidis. New results on intersection query problems. *The Computer Journal*, 40:22–29, 1997.

[37] U. Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25(2):163–177, 2001.

[38] U. Brandes. On variants of shortest-path betweenness centrality and their generic computation. *Social Networks*, 30(2):136–145, 2008.

[39] U. Brandes and D. Fleischer. Centrality measures based on current flow. In *Proc. 22nd Symposium on Theoretical Aspects of Computer Science*, volume 3404 of *LNCS*, pages 533–544. Springer, 2005.

[40] A. L. Buchsbaum, M. Goldwasser, S. Venkatasubramanian, and J. R. Westbrook. On external memory graph traversal. In *Proc. 11th ACM/SIAM Symposium on Discrete Algorithms*, pages 859–860, 2000.

[41] B. R. Calder and L. A. Mayer. Automatic processing of high-rate, high-density multibeam echosounder data. *Geochemistry, Geophysics, Geosystems*, 4(6):1048, June 2003.

[42] B. R. Calder and D. Wells. CUBE user's manual. Technical report, Center for Coastal and Ocean Mapping and NOAA/UNH Joint Hydrographic Center, University of New Hampshire, Jan. 2007. Version 1.13.

[43] G. Canepa, O. Bergem, and N. G. Pace. A new algorithm for automatic processing of bathymetric data. *IEEE Journal of Oceanic Engineering*, 28(1):62–77, Jan. 2003.

[44] T. M. Chan, K. G. Larsen, and M. Pătraşcu. Orthogonal range searching on the RAM, revisited. In *Proc. 27th ACM Symposium on Computational Geometry*, pages 1–10, 2011. See also arXiv:1011.5200.

[45] B. Chazelle. Lower bounds for orthogonal range searching: I. The reporting case. *Journal of the ACM*, 37(2):200–212, 1990.

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

[47] R. A. Chowdhury and V. Ramachandran. External-memory exact and approximate all-pairs shortest-paths in undirected graphs. In *Proc. 16th ACM/SIAM Symposium on Discrete Algorithms*, pages 735–744, 2005.

[48] H. Cohen and E. Porat. Fast set intersection and two-patterns matching. *Theoretical Computer Science*, 411(40-42):3795–3800, 2010.

[49] H. Cohen and E. Porat. On the hardness of distance oracle for sparse graph. *The Computing Research Repository (arXiv)*, 1006.1117, 2010.

[50] A. Danner. *I/O efficient algorithms and applications in geographic information systems.* PhD thesis, Duke University, 2006. Section 3.4.1.

[51] A. Danner, T. Mølhave, K. Yi, P. K. Agarwal, L. Arge, and H. Mitasova. TerraStream: from elevation data to watershed hierarchies. In *Proc. 15th ACM SIGSPATIAL GIS*, Nov. 2007.

[52] P. Davoodi, M. Smid, and F. van Walderveen. Two-dimensional range diameter queries. In *Proc. 10th Latin American Theoretical Informatics Symposium*, pages 219–230, Apr. 2012.

[53] T. K. Dey and S. Goswami. Provable surface reconstruction from noisy samples. *Computational Geometry: Theory and Applications*, 35:124–141, 2006.

[54] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[55] H. Edelsbrunner. Computing the extreme distances between two convex polygons. *Journal of Algorithms*, 6(2):213–224, 1985.

[56] EIVA. EIVA offline software: Post-processing and charting. Product brochure at `http://eiva.dk/Files/Billeder/Site_Images/Produktblad_Offline_web.pdf`.

[57] EIVA. AntiNoise plugin for NaviModel, May 2010. Press release at `http://www.hydro-international.com/news/id3899-Antinoise_Plugin.html`.

[58] C. Faloutsos and K.-I. Lin. FastMap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. In *Proc. SIGMOD International Conference on Management of Data*, pages 163–174, 1995.

[59] R. W. Floyd. Permuting information in idealized two-level storage. In R. Miller and J. Thatcher, editors, *Complexity of Computer Calculations*, pages 105–109. Plenum, 1972.

[60] G. N. Frederickson. Fast algorithms for shortest paths in planar graphs, with applications. *SIAM Journal on Computing*, 16(6):1004–1022, 1987.

[61] M. L. Fredman and D. E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47(3):424–436, 1993. See also STOC'90.

[62] L. C. Freeman. A set of measures of centrality based on betweenness. *Sociometry*, pages 35–41, 1977.

[63] L. C. Freeman. Centrality in social networks conceptual clarification. *Social Networks*, 1(3):215–239, 1978–1979.

[64] R. Freitas, J. Slember, W. Sawdon, and L. Chiu. GPFS scans 10 billion files in 43 minutes. Technical report, IBM Advanced Storage Laboratory, IBM Almaden Research Center, San Jose, 2011. Section 2.

[65] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. *ACM Transactions on Algorithms*, 8(1):4:1–4:22, 2012.

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

[67] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *Proc. 34th Annual Symposium on Foundations of Computer Science*, pages 714–723, Nov. 1993.

[68] P. Gupta. Algorithms for range-aggregate query problems involving geometric aggregation operations. In *Proc. 16th International Symposium on Algorithms and Computation*, pages 892–901, 2005.

[69] P. Gupta, R. Janardan, Y. Kumar, and M. H. M. Smid. Data structures for range-aggregate extent queries. In *Proc. 20th Canadian Conference on Computational Geometry*, pages 7–10, 2008.

[70] P. Gupta, R. Janardan, and M. Smid. Further results on generalized intersection searching problems: counting, reporting, and dynamization. *Journal of Algorithms*, 19:282–317, Sept. 1995.

[71] S. Har-Peled. A practical approach for computing the diameter of a point set. In *Proc. 17th ACM Symposium on Computational Geometry*, pages 177–186. ACM, 2001.

[72] S. Har-Peled and Y. Wang. Shape fitting with outliers. *SIAM Journal on Computing*, 33(2):269–285, 2004.

[73] J. M. Hellerstein, E. Koutsoupias, D. P. Miranker, C. H. Papadimitriou, and V. Samoladas. On a model of indexability and its bounds for range queries. *Journal of the ACM*, 49(1):35–55, 2002.

[74] M. R. Henzinger, P. Klein, S. Rao, and S. Subramanian. Faster shortest-path algorithms for planar graphs. *Journal of Computer and System Sciences*, 55(1):3–23, 1997.

[75] S. Hong, B. Song, and S. Lee. Efficient execution of range-aggregate queries in data warehouse environments. In *Proc. 20th International Conference on Conceptual Modeling*, pages 299–310, 2001.

[76] J. E. Hughes Clarke. Applications of multibeam water column imaging for hydrographic survey. *Hydrographic Journal*, 120:3–14, Apr. 2006.

[77] International Disk Drive Equipment and Materials Association (IDEMA). The advent of Advanced Format. Note at `http://www.idema.org/?page_id=2369`.

[78] M. Isenburg, Y. Liu, J. Shewchuk, and J. Snoeyink. Streaming computation of Delaunay triangulations. In *Proc. 33rd SIGGRAPH*, pages 1049–1056, Aug. 2006.

[79] J. JaJa, C. W. Mortensen, and Q. Shi. Space-efficient and fast algorithms for multidimensional dominance reporting and counting. In *Proc. 15th International Symposium on Algorithms and Computation*, pages 558–568, 2004.

[80] R. Janardan and M. Lopez. Generalized intersection searching problems. *International Journal of Computational Geometry & Applications*, 3:39–69, 1993.

[81] L. Katz. A new status index derived from sociometric analysis. *Psychometrika*, 18(1):39–43, 1953.

[82] P. N. Klein. Multiple-source shortest paths in planar graphs. In *Proc. 16th Annual Symposium on Discrete Algorithms*, pages 146–155, 2005.

[83] P. N. Klein, S. Mozes, and O. Weimann. Shortest paths in directed planar graphs with negative lengths: a linear-space $O(n \log^2 n)$-time algorithm. *ACM Transactions on Algorithms*, 6(2), 2010.

[84] D. Knoke and S. Yang. *Social Network Analysis*. Sage Publications, Inc, 2008.

[85] M. Komorowski. A history of storage cost. `http://www.mkomo.com/cost-per-gigabyte`.

[86] P. Kumar and E. A. Ramos. I/O-efficient construction of Voronoi diagrams. Technical report, Dec. 2002.

[87] V. Kumar and E. J. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proc. 8th IEEE Symposium on Parallel and Distributed Processing*, pages 169–176, 1996.

[88] K. G. Larsen. Higher cell probe lower bounds for evaluating polynomials. In *Proc. 53rd IEEE Symposium on Foundations of Computer Science*, 2012. To appear.

[89] K. G. Larsen and R. Pagh. I/O-efficient data structures for colored range and prefix reporting. In *Proc. 22nd ACM/SIAM Symposium on Discrete Algorithms*, pages 583–592, 2012.

[90] K. G. Larsen and F. van Walderveen. Near-optimal range reporting structures for categorical data. Manuscript submitted for publication, 2012.

[91] L. Leydesdorff. Betweenness centrality as an indicator of the interdisciplinarity of scientific journals. *Journal of the American Society for Information Science and Technology*, 58(9):1303–1319, 2007.

[92] R. J. Lipton and R. E. Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, 1979.

[93] A. Maheshwari and N. Zeh. I/O-optimal algorithms for planar graphs using separators. In *Proc. 13th Annual Symposium on Discrete Algorithms*, pages 372–381, 2002.

[94] A. Maheshwari and N. Zeh. I/O-efficient planar separators. *SIAM Journal on Computing*, 38(3):767–801, 2008.

[95] K. Mehlhorn and U. Meyer. External-memory breadth-first search with sublinear I/O. In *Proc. 10th Annual European Symposium on Algorithms*, volume 2461 of *LNCS*, pages 723–735, Sept. 2002.

[96] G. L. Miller. Finding small simple cycle separators for 2-connected planar graphs. *Journal of Computer and System Sciences*, 32:265–279, 1986.

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

[98] C. W. Mortensen. *Data structures for orthogonal intersection searching and other problems.* PhD thesis, IT University of Copenhagen, 2006. Chapter 6.

[99] K. Munagala and A. Ranade. I/O-complexity of graph algorithms. In *Proc. 10th Annual Symposium on Discrete Algorithms*, pages 687–694, Jan. 1999.

[100] Y. Nekrich. Space-efficient range reporting for categorical data. In *Proc. 31st ACM Symposium on Principles of Database Systems*, pages 113–120, 2012.

[101] Y. Nekrich and M. H. M. Smid. Approximating range-aggregate queries using coresets. In *Proc. 22nd Canadian Conference on Computational Geometry*, pages 253–256, 2010.

[102] M. Newman. Scientific collaboration networks. II. Shortest paths, weighted networks, and centrality. *Physical Review E*, 64(1):016132, 2001.

[103] M. Newman. A measure of betweenness centrality based on random walks. *Social networks*, 27(1):39–54, 2005.

[104] M. Pătraşcu. Lower bounds for 2-dimensional range counting. In *Proc. 39th ACM Symposium on Theory of Computation*, pages 40–46, 2007.

[105] M. Pătraşcu and L. Roditty. Distance oracles beyond the Thorup-Zwick bound. In *Proc. 51st IEEE Symposium on Foundations of Computer Science*, pages 815–823, 2010.

[106] M. Pătraşcu and M. Thorup. Time-space trade-offs for predecessor search. In *Proc. 38th ACM Symposium on Theory of Computation*, pages 232–240, 2006.

[107] F. Preparata and M. Shamos. *Computational geometry: an introduction.* Texts and monographs in computer science. Springer, 1991.

[108] S. Rahul, A. S. Das, K. S. Rajan, and K. Srinathan. Range-aggregate queries involving geometric aggregation operations. In *Proc. 5th WALCOM*, pages 122–133, 2011.

[109] RESON SeaBat 7125: ultra high resolution multibeam echosounder, 2011. Spec sheet at `http://www.reson.com/wp-content/uploads/2010/07/SeaBat-7125-product-leaflet-V12_small.pdf`.

[110] M. Hall (Seagate). Seagate reaches 1 terabit per square Inch milestone in hard drive storage with new technology demonstration, Mar. 2012. Press release at `http://www.reuters.com/article/2012/03/19/idUS157514+19-Mar-2012+BW20120319`.

[111] J. Shan, D. Zhang, and B. Salzberg. On spatial-range closest-pair query. In *Proc. 8th Advances in Spatial and Temporal Databases*, volume 2750 of *LNCS*, pages 252–269, 2003.

[112] R. Sharathkumar and P. Gupta. Range aggregate proximity queries. Technical Report IIIT/TR/2007/80, IIIT Hyderabad, 2007.

[113] Q. Shi and J. JaJa. Optimal and near-optimal algorithms for generalized intersection reporting on pointer machines. *Information Processing Letters*, 95(3):382–388, 2005.

[114] R. Tarjan. Depth-first search and linear graph algorithms. In *Proc. 12th Annual Symposium on Switching and Automata Theory*, pages 114–121, Oct. 1971.

[115] N. Zeh. An external-memory data structure for shortest path queries. Master's thesis, Friedrich-Schiller-Universität Jena, Germany, 1998.

[116] N. Zeh. I/O-efficient graph algorithms. Lecture notes from EEF Summer School on Massive Data Sets, Aarhus, Denmark, 2002.