An External-Memory Data Structure for Shortest Path Queries

Diplomarbeit

von

Norbert Zeh

zur Erlangung des akademischen Grades

Diplom-Informatiker

Friedrich-Schiller-Universität Jena
Fakultät für Mathematik und Informatik
Jena, Germany

November 1998

© Copyright
1998, Norbert Zeh

Selbständigkeitserklärung

Hiermit erkläre ich, daß ich die vorliegende Arbeit "An External-Memory Data Structure for Shortest Path Queries" selbständig und nur unter Verwendung der im Literaturverzeichnis aufgelisteten Literatur verfaßt habe.

Ich möchte jedoch erwähnen, daß ich viele meiner Ideen mit Mitgliedern der PARADIGM Group diskutiert habe.

Das Ergebnis in Abschnitt 3.4.2 ist das Resultat ausgiebiger Diskussionen mit Prof. Anil Maheshwari und David Hutchinson. In diesen Diskussionen ist es aus einer Reihe vorheriger Ansätze, die von mir entwickelt wurden, hervorgegangen.

Die Ergebnisse in den Kapiteln 6 und 7 konnte ich durch wertvolle Hinweise von Prof. Jörg-Rüdiger Sack, Prof. Anil Maheshwari und Dr. Lyudmil Aleksandrov stark vereinfachen.

Ottawa, den 19. November 1998, ______________________________

Norbert Zeh

An External-Memory Data Structure for Shortest Path Queries

Diplomarbeit

Norbert Zeh

Betreuer: Prof. Dr. Hans-Dietrich Hecker,
Friedrich-Schiller-Universität Jena, Germany

Prof. Dr. Jörg-Rüdiger Sack,
Carleton University, Ottawa, Canada

Friedrich-Schiller-Universität Jena
November 1998

To My Parents

Edeltraut and Winfried Zeh

Abstract

In this work, we deal with the problem of preprocessing graphs with $\sqrt{N}$-separators for fast shortest path queries. We look at this problem in an external-memory environment. We use an efficient internal memory approach to this problem due to Djidjev and replace the basic ingredients of this algorithm by their external-memory equivalents.

These ingredients are (1) a separator algorithm for planar graphs and (2) a single source shortest path algorithm. For (2) we use an algorithm due to Crauser, Mehlhorn, and Ulrich. For (1) we present an external-memory version of the algorithm by Lipton and Tarjan, provided that an external-memory breadth-first search algorithm is given. Part of this algorithm is a new algorithm for triangulating embedded planar graphs in external memory.

Another important ingredient is preprocessing of the shortest path trees which are part of our data structure for reporting paths I/O-efficiently. More generally, we present an approach to store rooted trees in external memory so that bottom-up paths can be traversed I/O-efficiently.

Acknowledgements

Although there are many people who deserve to be thanked for their support in the course of my studies, there are some whom I want to thank especially and explicitly here.

First of all, I want to thank my two supervisors: Prof. Hans-Dietrich Hecker and Prof. Jörg-Rüdiger Sack for their devotion and for making my visit to Carleton University and thereby this work possible at all.

I want to thank Prof. Anil Maheshwari and David Hutchinson for their wonderful cooperation on external-memory matters and for all the time they spent on our joint research.

Prof. Hristo Djidjev deserves to be thanked for interesting discussions on problems concerning graph separators and shortest path preprocessing during his visit to Carleton University.

Jana Dietel deserves my most sincere thanks for being a wonderfully patient friend who coped with my never ending list of questions about all these small details in many papers that I really wanted to understand.

Still talking about school, I finally want to thank all the graduate students of the School of Computer Science at Carleton University who made my visit to Carleton University a wonderful experience. Especially, Pat Morin and Jason Morrison made thinking about algorithms and data structures real fun instead of hard work.

My parents deserve very, very special thanks for helping me to cope with all these every-day life problems, small and big ones, in a way only parents can do that. Thus, I could always focus on my studies. Thank you for always believing in me and my abilities.

Finally, I must not forget to thank all my friends for never taking me too seriously. This helped me a lot not to forget that there are other important things in life beside computer science. Thank you for filling my breaks from work, shorter and longer ones, with pleasure. I gained new energy for more work to come from that.

Contents

Abstract **v**

Acknowledgements **vi**

1 Introduction **1**

1.1 Overview 1

1.2 Model of Computation 2

1.2.1 Single Disk Model 3

1.2.2 Parallel Disk Model 4

1.2.3 Storing Data on Multiple Disks 5

1.2.4 Some Notation and Basic Results 5

1.2.5 An External-Memory Stack 6

1.3 Basic Definitions 6

1.3.1 Graphs 6

1.3.2 Paths in Graphs 8

1.3.3 Rooted Trees 9

1.3.4 Graph Separators 9

2 An External-Memory Shortest Path Data Structure **11**

2.1 Shortest Paths and Hierarchical Decomposition Trees 12

2.2 Distance Queries 16

2.3 Shortest Path Queries 18

2.4 Analysis 21

2.5 Constructing the Data Structure 22

3 Blocking Rooted Trees: The Storage Scheme **28**
3.1 Path Traversals 28
3.2 Speed-Up and Storage Blow-Up 29
3.3 Previous Results for Online Traversals 30
3.4 Offline Traversals 30
3.4.1 Tight Bounds without Storage Blow-Up 31
3.4.2 An Optimal Approach with Storage Blow-Up 37

4 Labeling Rooted Trees in External Memory **43**
4.1 Time-Forward Processing 44
4.2 3-Coloring a Tree in External Memory 47
4.3 Computing Vertex-to-Root Distances 50
4.4 Computing Subtree Sizes or Weights 57
4.5 Computing Preorder, Postorder, and Levelorder Numbers 58
4.5.1 Preorder and Postorder Numbers 58
4.5.2 Levelorder Numbers 61
4.6 Lower Bounds 63

5 Blocking Rooted Trees: The Algorithm **64**
5.1 Framework of the Algorithm 64
5.2 MakeEdgeList 67
5.3 MakeParentPointers 68
5.4 ComputeSubtreeLabels 69
5.5 ConstructSubtrees 71

6 Separating Embedded Planar Graphs in External Memory **76**
6.1 Data Representation 77
6.2 Framework of the Algorithm 77
6.2.1 Computing Connected Components 80
6.2.2 Finding the Heaviest Component 83
6.2.3 Constructing A and B 85
6.3 Separating Connected Planar Graphs 85

6.4 Finding a Simple Cycle Separator in an Embedded Triangulation . . 96
6.5 Graph Separators and Shortest Paths 104

7 Triangulating Embedded Planar Graphs in External Memory 105
7.1 Framework of the Algorithm 105
7.2 Identifying Faces 106
7.3 Triangulating Faces 111

8 Conclusions 124

A Deutsche Zusammenfassung 127

Bibliography 130

Chapter 1

Introduction

1.1 Overview

In this work, we develop an external-memory data structure for fast shortest path queries on graphs with $\sqrt{N}$-separator. We also present an I/O-efficient algorithm to construct this data structure for a given graph. Finding shortest paths in graphs is a very important and intensively studied problem. It has applications in communication systems, transportation problems, scheduling, computation of network flows, and geographic information systems.

The problem of preprocessing graphs for fast shortest path queries has been considered by Feuerstein and Marchetti [19], Djidjev, Pantziou, and Zariolagis [18], and Djidjev [16]. Our external-memory approach is based on the approach in [16]. We describe our solution in Chapter 2.

This solution relies on some other ingredients. The data structure contains shortest path trees providing us with the shortest path information between appropriately chosen pairs of vertices. To report the shortest path between two query vertices, we have to traverse two paths in one of these shortest path trees towards the root. In Chapter 3, we show how to store shortest path trees so that these path traversals take an optimal number of I/Os.

For the construction of the shortest path data structure, we need a separator algorithm for embedded planar graphs, a single source shortest paths algorithm, and

an algorithm that converts the shortest path trees constructed by the shortest path algorithm into the blocked representation presented in Chapter 3.

The algorithm for converting the shortest path trees into a blocked representation is described in Chapter 5. It uses some basic labeling techniques for rooted trees which are developed in Chapter 4. For the single source shortest paths problem, we use the currently best known algorithm by Crauser, Mehlhorn, and Meyer [13]. Our external-memory separator algorithm is based on Lipton and Tarjan's planar separator algorithm [26]. We describe our separator algorithm in Chapter 6. We show that computing a separator of an embedded planar graph is asymptotically at most as hard as computing a breadth-first spanning tree of this graph. Computing such a spanning tree I/O-efficiently is still an open problem. We use a new external-memory algorithm for triangulating embedded planar graphs as part of our algorithm. We develop this algorithm in Chapter 7.

In Chapter 8, we highlight on some open problems related to the results in this thesis and, more importantly, on some ideas how to avoid the apparently hard problem of computing a BFS-tree in our separator algorithm.

The remaining two sections of this chapter introduce the model of computation that we use, present results for some basic operations and data structures in this model, and give the definitions of all terms that we use throughout this work. Additional definitions, if necessary, are presented in the respective chapters.

1.2 Model of Computation

In large-scale applications such as GIS, VLSI design, computational biology and chemistry, the whole data set usually does not fit into internal memory. Then the bottleneck of the computation is the transfer of data (input/output or short I/O) between fast internal memory and slow external memory. This bottleneck becomes more significant as parallel computing gains popularity and CPU speeds increase at an annual rate of 40–60%, while disk transfer rates increase by only 7–10% annually [30]. The internal memory sizes of computers also increase, but not nearly as fast as the amount of data to be handled grows.

That is why we chose a model of computation which is designed to analyze the I/O-complexity of algorithms. This model has been introduced by Aggarwal and Vitter [2] and has been extended to multiple disks by Vitter and Shriver [35].

1.2.1 Single Disk Model

The most time-consuming part of an I/O-operation is the time to position the read-write head of the disk. Thus, it takes almost the same amount of time to read or write a block of B consecutive data items as it takes to read or write only one item from or to disk. Therefore, modern computers try to "hide" (amortize) disk latency by performing I/O-operations blockwise, where a block is a sequence of B contiguous items on disk. Usually, as for instance in the UNIX file system, a block is also required to begin at a position which is a multiple of B.

These features of real computers are captured in the I/O model of Aggarwal and Vitter [2]. In this model we assume that the computer can hold M data items in its fast internal memory. The problem size N is considerably larger than M. Hence, most of the information is stored in slow external memory. Let the memory cells in external memory be numbered beginning with 0. We group external memory locations iB through $(i+1)B-1$ into block $\mathcal{B}_i, i \in \mathbb{N}$. In practice, B is some parameter given by the characteristics of the computer that we use. The only permitted operation in external memory is the transfer of at most B items from internal memory to some block $\mathcal{B}_i$ or vice versa. One such transfer to or from a single block is counted as one I/O-operation. Computation has to take place in internal memory and is for free. This is reasonable because the time to transfer the data between internal and external memory is the dominating part of the running time of an algorithm involving I/O-operations. The complexity of an algorithm is the number of I/Os that it performs.

In order to be able to transfer at least one block at a time to or from external memory, we assume that $M \geq B$ and often $M \geq cB$, for some constant $c > 1$, so that we can hold a constant number of blocks in internal memory.

1.2.2 Parallel Disk Model

A popular approach to increase the bandwidth between internal and external memory is to use a number of disks in parallel. This approach is captured in the *parallel disk model* (PDM) due to Vitter and Shriver [35], which is an extension of the model of Aggarwal and Vitter. In this model, we assume that we have $D > 1$ parallel disks and that it is possible to transfer up to D blocks between internal and external memory in a single I/O-operation, as long as each of these blocks is stored on another disk. The greater bandwith would not be much use, if we did not have enough memory to hold at least D blocks of data at a time in internal memory. Thus, we assume that $M \geq DB$ or, as above, $M \geq cDB$, for some constant $c > 1$.

Hence, the four parameters in terms of which we have to describe the I/O-complexity of our algorithms are:

$$\begin{aligned} N &= \text{problem size}, \\ M &= \text{internal memory size}, \\ B &= \text{block size}, \\ D &= \text{number of parallel disks}. \end{aligned}$$

An important issue in the PDM is balancing the load between the disks, i.e., distributing the data evenly over the D disks. We say that the load is evenly distributed iff the number of used blocks differs by at most one between any two of the D disks. Most of our algorithms balance the load; some do not.

Since we only distinguish between internal and external memory, the model that we use is a two-level memory model. Other models describing more than two levels of memory (e.g., registers, on-chip cache, off-chip cache, main memory, external memory) have been proposed (e.g., [4] and [36]). We have chosen two-level models because algorithm design is easier in these models than in one of the more sophisticated multi-level models and because the most significant difference in speed and size is between external and main memory.

1.2.3 Storing Data on Multiple Disks

For many results in the parallel disk model, the layout of the data on the parallel disks is crucial for the efficiency of the algorithm. The following layout, called *disk striping* [35], is assumed for almost all PDM-algorithms.

Assume that, in the single disk model, we would store the data in blocks $\mathcal{B}_0, \ldots, \mathcal{B}_k$, in this order. Using disk striping, we store block $\mathcal{B}_i$ in the $\lfloor i/D \rfloor$-th block on the $(i \bmod D)$-th disk. This ensures that we can always read D consecutive blocks with a single I/O. We say that the data is *striped across D disks.*

If not stated otherwise, the inputs and outputs of our algorithms are striped. In some of our results, we stripe the data and tie blocks $\mathcal{B}_{iD}, \ldots, \mathcal{B}_{(i+1)D-1}$ together to form a superblock $\mathcal{B}'_i$. Each such superblock can be accessed with a single I/O. That is, we can use D disks as a single disk with block size DB. However, it must be guaranteed that we have enough internal memory to store DB items. We capture this in the following lemma.

Lemma 1.1 *A single disk algorithm with I/O-complexity $\mathcal{I}(N, M, B) = f(N, M, B)$ for $M \geq cB$ ($c \geq 1$) can be generalized to a multiple disk algorithm with I/O-complexity $\mathcal{I}(N, M, B, D) = f(N, M, DB)$ for $M \geq cDB$.*

1.2.4 Some Notation and Basic Results

Assuming that B, D, and M are understood, we denote by $sort(N)$ the number of I/Os required to sort N items on a machine with M internal memory and D parallel disks with block size B. Analogously, $perm(N)$ (resp. $scan(N)$) denotes the number of I/Os required to create a given permutation of N items (resp. to scan an array of N items from head to tail once). It is easy to see that $scan(N) = \Theta(\frac{N}{DB})$. Aggarwal and Vitter [2] showed that, for $D = 1$, $sort(N) = \Theta(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ and $perm(N) = \Theta(\min\{N, \frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B}\})$. Arge [6] presented an alternative proof for the upper and lower bounds for sorting. To prove the upper bound, he used a new data structure called *buffer tree* [5]. The lower bound follows from the proof of Arge, Knudsen, and Larsen that any problem with an $\Omega(N \log N)$ time bound in the comparison model

has an $\Omega(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B})$ I/O-bound in this model [7]. Nodine and Vitter [29] and Vitter and Shriver [35] proved that, for $D > 1$, $\mathit{sort}(N) = \Theta(\frac{N}{DB}\log_{\frac{M}{B}}\frac{N}{B})$. Vitter and Shriver [35] showed that, for $D > 1$, $\mathit{perm}(N) = \Theta(\min\{\frac{N}{D}, \frac{N}{DB}\log_{\frac{M}{B}}\frac{N}{B}\})$. Note that these two bounds are better than the ones that we would achieve by applying Lemma 1.1 to the single-disk results.

1.2.5 An External-Memory Stack

A stack can be implemented with an optimal number of I/Os as follows: Assume that $M \geq 2B$. Then we keep the stack in internal memory as long as its size is less than $2B$. When its size becomes $2B$, we write the bottommost B items to external memory. This takes one I/O and makes room for another B items in internal memory. The next I/O becomes necessary when either we do not have enough internal memory to push more items on the stack or we try to pop an item that is not in internal memory from the stack. For the first situation to arise, we have to push at least B additional items on the stack. In order to access the topmost item that is not in internal memory, we have to pop the B items which are in internal memory first. Thus, we perform at least B stack operation before the next I/O. Hence, a sequence of N push and pop operations on the stack takes optimal $O(N/B)$ I/Os. Using Lemma 1.1, this generalizes to $O(N/DB)$ I/Os in the case $D > 1$.

1.3 Basic Definitions

The major part of this work is about graphs and rooted trees. We give some basic definitions here. Additional definitions, if necessary, are given in the respective chapters.

1.3.1 Graphs

DEFINITION 1.1 A *directed graph* is a pair $G = (V(G), E(G))$ of sets. $V(G)$ is called the *vertex set* of G, and $E(G) \subseteq V(G) \times V(G)$ is called the *edge set* of G.

DEFINITION 1.2 An *undirected graph* is a pair $G = (V(G), E(G))$ of sets. $V(G)$ is called the *vertex set* of G, and $E(G) \subseteq \{\{v,w\} : v,w \in V(G)\}$ is called the *edge set* of G.

For every directed graph $G = (V(G), E(G))$, there is a corresponding undirected graph $G' = (V(G'), E(G'))$ with $V(G') = V(G)$ and $E(G') = \{\{v,w\} : (v,w) \in E(G)\}$. In this work, all graphs are undirected unless stated otherwise.

DEFINITION 1.3 A graph $G' = (V(G'), E(G'))$ is a *subgraph* of another graph $G = (V(G), E(G))$, if $V(G') \subseteq V(G)$, $E(G') \subseteq E(G)$, and for each edge $\{v,w\}$ (resp. (v,w)), $v,w \in V(G')$.

DEFINITION 1.4 Two vertices v and w of an undirected graph $G = (V(G), E(G))$ are *adjacent*, if $\{v,w\} \in E(G)$. Adjacency in a directed graph is defined as adjacency in the corresponding undirected graph.

DEFINITION 1.5 A graph G is *planar*, if it can be drawn in the plane such that vertices are represented by points and edges are represented by pairwise non-intersecting continuous curves connecting their endpoints.

DEFINITION 1.6 The removal of the points and curves representing vertices and edges of a planar graph G divides the plane into connected regions. We call these regions the *faces* of G.

Note that the faces of a planar graph G are not well-defined. They depend on the embedding, i.e., on the way we draw the graph G in the plane. We call a graph *embedded*, if it is drawn in the plane.

DEFINITION 1.7 The *in-degree* (resp. *out-degree*) of a vertex v in a directed graph $G = (V(G), E(G))$ is defined as $|\{w : (w,v) \in E(G)\}|$ (resp. $|\{w : (v,w) \in E(G)\}|$).

DEFINITION 1.8 The *degree* of a vertex v in an undirected graph $G = (V(G), E(G))$ is defined as $|\{w : \{v,w\} \in E(G)\}|$.

That is, the degree of a vertex v is the number of vertices in G that are adjacent to v. Note that we use a slightly different definition of this term for rooted trees (see Section 1.3.3).

1.3.2 Paths in Graphs

DEFINITION 1.9 A *path* p from a vertex v to a vertex w in a directed graph $G = (V(G), E(G))$ is a sequence $(v = v_0, \ldots, v_l = w)$ of vertices in $V(G)$ with $(v_i, v_{i+1}) \in E(G)$, for $1 \leq i < l$. We call l the *(unweighted) length* of p, which we denote by $|p|$. If $l = 0$, we call p *empty*.

DEFINITION 1.10 Let v and w be two vertices in a graph G. An *(unweighted) shortest path* $\pi(v, w)$ from v to w is a path from v to w with minimal (unweighted) length $|\pi(v, w)|$ among all paths from v to w. We call $d(v, w) = |\pi(v, w)|$ the *(unweighted) distance* from v to w.

DEFINITION 1.11 Let a directed graph $G = (V(G), E(G))$, a weight function $w : V(G) \to \mathbb{R}$ (resp. $w : E(G) \to \mathbb{R}$), and a path $p = (v_1, \ldots, v_l)$ in G be given. The *weighted length* of p is defined as $||p|| = \sum_{i=1}^{l} w(v_i)$ (resp. $||p|| = \sum_{i=1}^{l-1} w((v_i, v_{i+1}))$).

Paths, path lengths, shortest paths, and distances are defined analogously for undirected graphs.

DEFINITION 1.12 A *cycle* in a graph $G = (V(G), E(G))$ is a non-empty path $c = (v = v_0, \ldots, v_l = v)$. We call c a *negative cycle*, if $||c|| < 0$.

DEFINITION 1.13 An undirected graph G is *connected*, if, for any two vertices v and w in G, there is a path from v to w.

DEFINITION 1.14 A *connected component* C of a graph G is a maximal connected subgraph of G. That is, there is no connected graph C' with $C \subset C' \subseteq G$.

DEFINITION 1.15 A *(free) tree* is a connected undirected graph that does not contain cycles.

DEFINITION 1.16 Let $G = (V(G), E(G))$ be a (directed) graph that does not contain negative cycles. Then for any two vertices v and w in G, there is a path p from v to w that has minimal weighted length $||p||$ among all paths from v to w. We call this path the *weighted shortest path* from v to w and denote it by $\Pi(v, w)$. We call $\delta(v, w) = ||\Pi(v, w)||$ the *weighted distance* from v to w.

1.3.3 Rooted Trees

DEFINITION 1.17 A *rooted tree* is a tree $T = (V(T), E(T))$ with a distinguished vertex $r \in V(T)$. We call r the *root* of T. For an edge $\{v, w\} \in E(T)$, we call v the *parent* of w and w the *child* of v, if $d(r, v) < d(r, w)$. Two vertices are *siblings*, if they have the same parent. Vertex v is an *ancestor* of w, and w is a *descendant* of v, if the shortest path $\pi(v, w)$ does not contain v's parent. If a vertex v has no children, we call v a *leaf* of T. If v is no leaf, we call it an *internal node.*

For every rooted tree, there is a corresponding free tree.

DEFINITION 1.18 The *degree* of a vertex v in a rooted tree T is the number of children of v in T.

Note that according to this definition all internal nodes of a rooted tree, except for the root of the tree, have degree one less than in the corresponding free tree. This seems to be a bit inconsistent. However, considering rooted trees as directed graphs with edges directed from parents to children, the degree of each vertex is just its out-degree.

1.3.4 Graph Separators

Let $G = (V(G), E(G))$ be a graph and $w : V(G) \to \mathbb{R}^+$ be a weight function with $\sum_{v \in V(G)} w(v) \leq 1$. We define the weight of a subgraph $G' \subseteq G$ as the sum of the weights of the vertices in $V(G')$. A *(vertex)-separator* of G is a vertex set $C \subseteq V(G)$ that divides G into two components A and B, each of weight at most $\frac{2}{3}$. That is, G contains no edge $\{a, b\}$ with $a \in A$ and $b \in B$. We say that an $f(N)$-separator theorem holds for a certain class of graphs, if each N-vertex graph in this class has a

separator of size $O(f(N))$. For trees and outerplanar graphs, a 1-separator theorem holds. In particular, every tree has a separator consisting of only a single vertex. Every outerplanar graph has a separator of size at most 2. For grid graphs, a $\sqrt{N}$-separator theorem holds. Lipton and Tarjan were the first to present a proof that, for planar graphs, a $\sqrt{N}$-separator theorem holds.

If there are no weights assigned to the vertices of G, the goal is to find a small set of vertices that divides G into two components, each of which contains at most $\frac{2}{3}|V(G)|$ vertices. This case can be reduced to the weighted case by assigning vertex weights $1/|V(G)|$ to all vertices of G.

Chapter 2

An External-Memory Shortest Path Data Structure

In this chapter, we present an external-memory shortest path data structure that is based on Djidjev's internal-memory shortest path data structure for connected graphs with $\sqrt{N}$-separator [16]. Djidjev's data structure uses $O(N^{3/2})$ space and allows for answering distance queries in $O(\sqrt{N})$ time. Reporting the corresponding shortest path takes $O(K)$ additional time, where K is the length (number of vertices) of the path.

Our data structure is an optimal external-memory version of the data structure in [16]. That is, answering queries takes the same amount of computation, $O(\sqrt{N}/DB)$ I/Os for answering distance queries, and $O(K/DB)$ additional I/Os for reporting the shortest path. The data structure uses $O(N^{3/2}/B)$ blocks of external memory.

We present this approach as a framework where we assume that certain parts can be implemented to take a certain number of I/Os. The details of these parts are presented in later chapters.

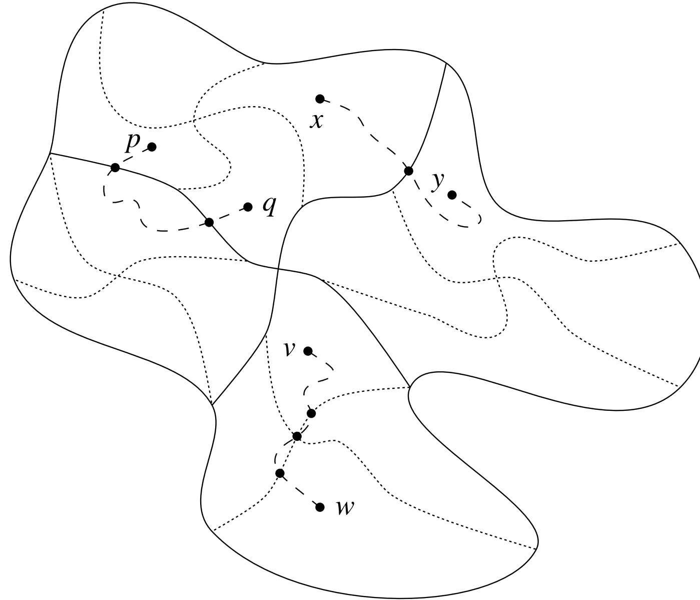

Figure 2.1: Illustrating the different cases of shortest paths in a hierarchically decomposed graph G. Solid lines represent the "graph boundary" and the top-level separator S. Dotted lines represent the separators of each of the four components produced by the top-level separator. Dashed lines represent the shortest paths between the respective pairs of vertices.

2.1 Shortest Paths and Hierarchical Decomposition Trees

As already mentioned, our method is based on an internal-memory shortest path preprocessing algorithm due to Djidjev. It can be applied to all graphs with a $\sqrt{N}$-separator theorem. We assume that the graph is connected. If not, we build a separate shortest path data structure for every connected component of the graph. The basic idea is as follows:

Let G be a graph with non-negative weights assigned to its vertices or edges, S be a vertex separator of G, and s and t be two query vertices. Let $G_1, \ldots, G_r$ be the connected components of $G - S$. We want to compute the shortest path $\Pi(s,t)$ from s to t in G. We have to distinguish two cases:

(1) s and t are in two different components G_s and G_t of $G \setminus S$. Then the shortest

path from s to t must contain some vertex $b \in S$. Thus, we can compute $\Pi(s,t)$ as the concatenation of $\Pi(s,b)$ and $\Pi(b,t)$ ($\Pi(x,y)$ in Figure 2.1).

(2) s and t are in the same component. Then the shortest path might contain some vertex in S ($\Pi(p,q)$ in Figure 2.1) or it might not ($\Pi(v,w)$ in Figure 2.1). We use a separator S_i of the component G_i containing s and t and repeat the whole process recursively. Note, however, that, as in the case of $\Pi(p,q)$ in Figure 2.1, $\Pi(s,t)$ does not necessarily contain a vertex in S_i. We show that, in this case, it either contains a vertex in S or stays inside a connected component of $G_i \setminus S_i$. Hence, we can apply this argument recursively.

Based on these observations, we construct a shortest path data structure as follows: First, we build a hierarchical decomposition tree $T(G)$ of G. With each node v of $T(G)$, we associate a subgraph $G(v)$ of G and a vertex separator $S(v)$ of $G(v)$. We store $S(v)$ explicitly, while $G(v)$ is given only implicitly.

If $|V(G)| = 1$, then $T(G)$ consists of a single node r with $G(r) = G$ and $S(r) = V(G)$.

Otherwise, we apply a separator algorithm to find a $\sqrt{|V(G)|}$-separator S of G. Let $G_1, \ldots, G_r$ be the connected components of $G \setminus S$. We construct hierarchical decomposition trees $T(G_1), \ldots, T(G_r)$ for these components, create a root node r of $T(G)$ with $G(r) = G$ and $S(r) = S$, and make the root nodes of $T(G_1), \ldots, T(G_r)$ children of r.

Observation 2.1 *Each vertex $v \in G$ is stored exactly once in $T(G)$.*

Proof. This is shown by induction on the size of G.

If $|V(G)| = 1$, $T(G)$ consists of a single node r. The only vertex of G is stored in $S(r)$.

If $|V(G)| = n > 1$, assume that the claim is true for all graphs G' with $n' < n$ vertices. Each component G_i of $G \setminus S$ has less than n vertices. Each vertex of G is either in S or in some component G_i. By induction hypothesis, the vertices of each G_i are stored exactly once in the subtree $T(G_i)$. The vertices in S are stored in $S(r)$. □

Let s and t be two query vertices. We define $v(s)$ and $v(t)$ to be the nodes in $T(G)$ such that $s \in S(v(s))$ and $t \in S(v(t))$. $v(s,t)$ is the lowest common ancestor of

$v(s)$ and $v(t)$ in $T(G)$. $V(s,t)$ is the set of all ancestors of $v(s,t)$, inclusive. $B(s,t)$ is the set

$$B(s,t) = \bigcup_{v \in V(s,t)} S(v).$$

The following observation will come in handy later.

OBSERVATION 2.2 $V(s,t) = \{v \in T(G) : \{s,t\} \subseteq G(v)\}$.

PROOF. Let $x \in G$ and $V(x)$ be the set containing all ancestors of $v(x)$, inclusive, in $T(G)$. Then $V(s,t) = V(s) \cap V(t)$. Hence, it suffices to show that, for any vertex $x \in G$ and any node $v \in T(G)$, $x \in G(v)$ iff $v \in V(x)$.

Let v be a node of $T(G)$ that is not in $V(x)$. If v is a descendant of $v(x)$, then $G(v) \subseteq G(v(x)) \setminus S(v(x))$. But $x \in S(v(x))$, so that $x \notin G(v)$. Since $v \notin V(x)$, v cannot be an ancestor of $v(x)$. Therefore, if v is not a descendant of $v(x)$, the lowest common ancestor w of v and $v(x)$ in $T(G)$ must be different from both v and $v(x)$. Let v' be the child of w such that v is a descendant of v'; v'' be the child of w such that $v(x)$ is a descendant of v''. Then $G(v) \subseteq G(v')$ and $G(v(x)) \subseteq G(v'')$. But $G(v') \cap G(v'') = \emptyset$ and $x \in G(v(x)) \subseteq G(v'')$. Thus, $x \notin G(v)$. Hence, $x \in G(v)$ implies that $v \in V(x)$.

Let $v_1, \ldots, v_k$ be the vertices in $V(x)$ in top-down order, i.e., v_1 is the root of $T(G)$, $v_k = v(x)$, and v_i is the parent of v_{i+1} in $T(G)$, for all $1 \leq i < k$. Then $G(v_1) \supseteq G(v_2) \supseteq \cdots \supseteq G(v_k)$, and $x \in S(v_k) \subseteq G(v_k)$. Thus, $x \in G(v)$ for each vertex $v \in V(x)$. □

Next we prove the most important lemma of this chapter.

LEMMA 2.1 $\Pi(s,t) \cap B(s,t) \neq \emptyset$.

More intuitively, this lemma says that, for any pair of vertices $s, t \in G$, there is at least one vertex in $B(s,t)$ that is on the shortest path from s to t.

PROOF. Let $v_1, \ldots, v_k$ be the nodes in $V(s,t)$ in top-down order, i.e., v_1 is the root of $T(G)$ and $v_k = v(s,t)$. We prove that, if $\Pi(s,t) \cap \bigcup_{i=1}^{j-1} S(v_i) = \emptyset$, $\Pi(s,t)$ stays completely inside $G(v_j)$, for each $1 \leq j \leq k$.

The proof is by induction, the base case, $j = 1$, being trivial. If $j > 1$ and $\Pi(s,t) \cap \bigcup_{i=1}^{j-1} S(v_i) = \emptyset$, then $\Pi(s,t) \cap \bigcup_{i=1}^{j-2} S(v_i) = \emptyset$. Therefore, $\Pi(s,t)$ stays inside $G(v_{j-1})$ by induction hypothesis. The only way to leave a connected component of $G(v_{j-1}) \setminus S(v_{j-1})$ is to cross $S(v_{j-1})$ or to leave $G(v_{j-1})$. Thus, as $\Pi(s,t) \cap S(v_{j-1}) = \emptyset$ and $\Pi(s,t)$ does not leave $G(v_{j-1})$, $\Pi(s,t)$ stays inside $G(v_j)$. If $\Pi(s,t) \cap \bigcup_{i=1}^{k-1} S(v_i) = \emptyset$, $\Pi(s,t)$ stays completely inside $G(v_k) = G(v(s,t))$. However, s and t are in different connected components of $G(v(s,t)) \setminus S(v(s,t))$. Therefore, as $\Pi(s,t)$ stays inside $G(v(s,t))$, it must cross $S(v(s,t))$ to get from s to t. Hence, $\Pi(s,t) \cap B(s,t) \neq \emptyset$. $\square$

Assume that, for each node $v \in T(G)$, we have stored distances $dist(b,w)$ between all vertices $b \in S(v)$ and $w \in G(v)$; $dist(b,w)$ is the length of the shortest path between b and w in $G(v)$. We show the following lemma.

Lemma 2.2 *The distance $\delta(s,t)$ between two query vertices s and t in G is*

$$\delta(s,t) = \min\{dist(s,b) + dist(b,t) : b \in B(s,t)\}.$$

Proof. Lemma 2.1 implies that

$$\delta(s,t) = \min\{\delta(s,b) + \delta(b,t) : b \in B(s,t)\}.$$

Thus, it suffices to show that

$$\min\{dist(s,b) + dist(b,t) : b \in B(s,t)\} = \min\{\delta(s,b) + \delta(b,t) : b \in B(s,t)\}.$$

Let v be the highest node in $T(G)$ such that $\Pi(s,t)$ stays inside $G(v)$. Then $v \in V(s,t)$ because $s,t \in G(v)$. Hence, $S(v) \subseteq B(s,t)$. Moreover, if $\Pi(s,t) \cap S(v) = \emptyset$, $\Pi(s,t)$ would stay inside a connected component of $G(v) \setminus S(v)$, contradicting the choice of v. Thus, there is a vertex $b \in S(v) \cap \Pi(s,t)$, and the distances $dist(s,b)$ and $dist(b,t)$ have been computed with respect to $G(v)$. As $\Pi(s,t)$ stays inside $G(v)$, $\Pi(s,b)$ and $\Pi(b,t)$ do as well. Hence, $\delta(s,b) = dist(s,b)$ and $\delta(b,t) = dist(b,t)$, and $\delta(s,t) = \delta(s,b) + \delta(b,t) = dist(s,b) + dist(b,t)$. That is,

$$\min\{dist(s,b) + dist(b,t) : b \in B(s,t)\} \leq \min\{\delta(s,b) + \delta(b,t) : b \in B(s,t)\}.$$

On the other hand, if

$$\min\{dist(s,b) + dist(b,t) : b \in B(s,t)\} < \min\{\delta(s,b) + \delta(b,t) : b \in B(s,t)\},$$

there would be a vertex $b \in B(s,t)$ with $dist(s,b)+dist(b,t) < \delta(s,b)+\delta(b,t)$. That is, either $dist(s,b) < \delta(s,b)$ or $dist(b,t) < \delta(b,t)$, or both. W.l.o.g. let $dist(s,b) < \delta(s,b)$. Then, however, there would be a shorter path from s to b in $G(v(b))$ than in G. This is impossible because $G(v(b))$ is a subgraph of G. This concludes the proof. □

At this point, we only sketch the issue of reporting $\Pi(s,t)$ itself. We observe that, while computing $d(s,t)$, we can memorize the vertex b that produced the minimal sum $dist(s,b) + dist(b,t)$. With the shortest path tree $SPT(b, G(v))$ of b with respect to $G(v)$ stored somewhere, we can retrieve the corresponding shortest paths $\Pi(s,b)$ and $\Pi(b,t)$ from $SPT(b, G(v))$ and concatenate them to produce $\Pi(s,t)$. We will see later in this chapter how to do this efficiently.

2.2 Distance Queries

If we store $T(G)$ and distances between the vertices in $S(v)$ and those in $G(v)$ for each node v of $T(G)$, the data structure quickly becomes complex to handle, especially as we do not store $G(v)$ anywhere. We show how a simple data structure consisting of $|V(G)|$ arrays (one per vertex) achieves the same performance as $T(G)$. However, we use $T(G)$ to construct this data structure.

First, we redefine $S(v)$ to be an array of separator vertices rather than a set, for each node v of $T(G)$. This is important because it defines some order on the vertices in $S(v)$. In addition we redefine $V(w)$, for each vertex $w \in G$, to be a list of nodes in $T(G)$ rather than a set. This list will be sorted top-down with respect to $T(G)$.

Let $V(w) = \langle v_1, \ldots, v_k \rangle$. We define a list $B(w) = S(v_1) \mid S(v_2) \mid \cdots \mid S(v_k)$, where "|" is the concatenation operator. For each vertex $b \in B(w)$, we compute $dist(b,w)$ as the distance between b and w in $G(v(b))$ and store it with b in $B(w)$.

We claim that these arrays $B(w), w \in G$ provide all necessary information to compute the distance $\delta(s,t)$ between two query vertices s and t in G. Algorithm 2.1 shows how.

DISTANCEQUERY(s, t, B):

Input: The query vertices s and t of G and the arrays $B(v), v \in G$ as defined in the text.

Output: The distance $\delta(s, t)$ between s and t in G.

1: $\delta \leftarrow \infty$
2: Scan $B(s)$ and $B(t)$ to compute the distance $\delta(s, t)$:
{v is the current vertex in $B(s)$. w is the current vertex in $B(t)$.}
  if $v = w$ **then**
    if $dist(v, s) + dist(w, t) < \delta$ **then**
      $\delta \leftarrow dist(v, s) + dist(w, t)$
    end if
  else
    Stop scanning.
  end if
3: Return δ

Algorithm 2.1: Querying the distance $\delta(s, t)$ between two vertices $s, t \in G$.

LEMMA 2.3 *We can store the arrays* $B(w)$ *in* $O\left(\frac{\sum_{v \in G} |B(v)|}{B}\right)$ *blocks such that Algorithm 2.1 reports the distance* $\delta(s, t)$ *between* s *and* t *in the graph* G*. This takes* $O(\min\{|B(s)|, |B(t)|\}/DB)$ *I/Os.*

PROOF. We assume that the vertices of G are numbered contiguously from 1 through N, where $N = |V(G)|$. The query vertices $s = v_i$ and $t = v_j$ are given by their numbers i and j. Then we can store an array A and the arrays $B(v), v \in G$ at contiguous locations in external memory. A has size N, and $A[i]$ contains the address of $B(v_i)$. Storing A and all $B(v), v \in G$ this way uses $O\left(\frac{|V(G)| + \sum_{v \in G} |B(v)|}{B}\right)$ blocks. Since $|B(v)| \geq 1$, for each vertex $v \in G$, $|V(G)|$ is dominated by $\sum_{v \in G} |B(v)|$. Hence, the used storage is $O\left(\frac{\sum_{v \in G} |B(v)|}{B}\right)$ blocks. We retrieve the start addresses of $B(s)$ and $B(t)$ from A with at most 2 I/Os. Afterwards, we scan $B(s)$ and $B(t)$ simultaneously until we find the first position in both lists where the stored vertices do not match. Thus, we scan sublists of $B(s)$ and $B(t)$ of length at most $\min\{|B(s)|, |B(t)|\}$. This takes $O(scan(\min\{|B(s)|, |B(t)|\})) = O(\min\{|B(s)|, |B(t)|\}/DB)$ I/Os. That is, our query algorithm achieves the claimed I/O-bound. It remains to show its correctness.

Obviously, Algorithm 2.1 computes the minimum of the sums $dist(v, s) + dist(v, t)$ over all visited vertices v. Thus, it suffices to show that it visits all vertices in $B(s, t)$ and only those.

First, we show that the algorithm visits all vertices in $B(s, t)$: The nodes in $V(s, t)$ are stored at the beginning of $V(s)$ and $V(t)$. They appear in the same order in both lists because both lists are sorted top-down. Therefore, the first $|B(s, t)|$ vertices in $B(s)$ and $B(t)$ are the vertices in $B(s, t)$. They appear in the same order in both lists because of the strict orders of the nodes in $V(s)$ and $V(t)$ and of the vertices in each $S(v), v \in T(G)$. It remains to show that the $(|B(s, t)| + 1)$-st pair of vertices in $B(s)$ and $B(t)$ is no match anymore.

If $v(s)$ is an ancestor of $v(t)$ in $T(G)$, then $v(s, t) = v(s)$. Thus, $|B(s)| = |B(s, t)|$, i.e., there is no $(|B(s, t)| + 1)$-st vertex pair. The case that $v(t)$ is an ancestor of $v(s)$ is analogous.

Otherwise (if neither s nor t is an ancestor of the other), there is a $(|B(s, t)| + 1)$-st vertex pair because $V(s)$ and $V(t)$ each contain at least one node v that is not in $V(s, t)$. For this vertex, $S(v) \neq \emptyset$. Let this $(|B(s, t)| + 1)$-st pair of vertices be (x, y). Then x is the first vertex in $S(v)$ and y is the first vertex in $S(w)$, where v (resp. w) is the child of $v(s, t)$ on the path from $v(s)$ (resp. $v(t)$) to $v(s, t)$. Since $v(s, t)$ is the lowest common ancestor of $v(s)$ and $v(t)$, $v \neq w$. Therefore, as $G(v)$ and $G(w)$ are two different connected components of $G(v(s, t)) \setminus S(v(s, t))$, $G(v) \cap G(w) = \emptyset$. $S(v)$, however, is a separator of $G(v)$. Therefore, it contains only vertices in $G(v)$. Analogously, $S(w)$ contains only vertices in $G(w)$. Hence, $S(v) \cap S(w) = \emptyset$ and $x \neq y$. This concludes the proof. □

2.3 Shortest Path Queries

In this section, we show how to augment the data structure described in the previous section so that we can answer shortest path queries I/O-efficiently.

Let s and t be two vertices at distance $\delta(s, t)$ from each other. By Lemma 2.2, there is a vertex $b \in B(s, t)$ such that $\delta(s, t) = dist(s, b) + dist(b, t)$. That is, the shortest path $\Pi(s, t)$ is the concatenation of the shortest paths form s to b and from

b to t in $G(v(b))$. Consequently, if the shortest path tree $SPT(b, G(v))$ is stored somewhere, we can augment the distances $dist(s, b)$ and $dist(b, t)$ with pointers to s and t in $SPT(b, G(v))$. To compute $\Pi(s, t)$, we traverse the paths from s to the root b and from t to b in $SPT(b, G(v))$ and concatenate them.

That is, we extend our data structure as follows. For each node $v \in T(G)$ and each vertex $b \in S(v)$, we compute $SPT(b, G(v))$ and store it somewhere. For each vertex $w \in G$ and each vertex $b \in B(w)$, w must be a vertex in $SPT(b, G(v(b)))$ because $b \in B(w)$ implies that $w \in G(v(b))$. We store a pointer $p(b)$ to the copy of w in $SPT(b, G(v(b)))$ together with b in $B(w)$.

Now, given two query vertices s and t, we first compute $\delta(s, t)$ and keep pointers p_s and p_t to the copies of s and t, respectively, in the shortest path tree $SPT(G(b))$, where b is the currently found vertex that minimizes $dist(s, b) + dist(b, t)$. When we are finished, we use these pointers and traverse the paths from s to the root of the shortest path tree and from t to the root of the shortest path tree. The code is shown in Algorithm 2.2. We prove the following lemma.

LEMMA 2.4 *We can store the lists $B(w)$ and all shortest path trees $SPT(b, G(v(b)))$ in $O\left(\frac{\sum_{v \in G} |B(v)|}{B}\right)$ blocks such that Algorithm 2.2 reports the shortest path $\Pi(s, t)$ between two query vertices s and t in G with $O\left(\frac{\min\{|B(s)|, |B(t)|\} + K}{DB}\right)$ I/Os, where $K = |\Pi(s, t)|$.*

PROOF. The correctness of Algorithm 2.2 follows immediately from the discussion at the beginning of this section.

By Lemma 2.3, we can store the lists $S(w)$ in $O\left(\frac{\sum_{v \in G} |B(v)|}{B}\right)$ blocks such that Step 2 takes $O(\min\{|B(s)|, |B(t)|\}/DB)$ I/Os.

As we show in Corollary 3.2, the shortest path trees $SPT(b, G(v(b)))$ can be stored in $O\left(\frac{\sum_{b \in G} |SPT(b, G(v(b)))|}{B}\right)$ blocks so that traversing $\Pi(s, b)$ and $\Pi(t, b)$ takes $O(|\Pi(s, b)|/DB)$ and $O(|\Pi(t, b)|/DB)$ I/Os, respectively. The backward scan of $\Pi(t, b)$ in Step 3 takes $O(|\Pi(t, b)|/DB)$ I/Os. Hence, Step 3 takes $O((|\Pi(s, b)| + |\Pi(t, b)|)/DB) = O(|\Pi(s, t)|/DB)$ I/Os.

We have shown that we can store the lists $B(w)$ and the shortest path trees $SPT(b, G(v(b)))$ in $O\left(\frac{\sum_{v \in G} |B(v)| + \sum_{b \in G} |SPT(b, G(v(b)))|}{B}\right)$ blocks such that reporting

SHORTESTPATHQUERY(s, t, B):

Input: The query vertices s and t of G, the arrays $B(v), v \in G$ as defined above, and all shortest path trees $SPT(b, G(v(b)))$ referenced by pointers in B.

Output: The shortest path $\Pi(s, t)$ from s to t in G.

1: $\delta \leftarrow \infty$

2: Scan $B(s)$ and $B(t)$ to compute the distance $\delta(s, t)$:
{v is the current vertex in $B(s)$. w is the current vertex in $B(t)$.}

```
if v = w then
  if dist(v, s) + dist(w, t) < δ then
    δ ← dist(v, s) + dist(w, t)
    p_s ← p(v)
    p_t ← p(w)
  end if
else
  Stop scanning.
end if
```

3: Construct $\Pi(s, t)$:

Traverse $\Pi(s, b)$ in $SPT(b, G(v(b)))$ and write it it to disk. This path starts at position p_s.
Traverse $\Pi(t, b)$ in $SPT(b, G(v(b)))$ and write it it to disk. This path starts at position p_t.
Scan $\Pi(t, b)$ backward, starting one vertex after b. Append the traversed vertices to $\Pi(s, b)$.
{We start this scan one vertex after b so as not to include two copies of b in $\Pi(s, t)$.}

Algorithm 2.2: Computing the shortest path $\Pi(s, t)$ between two vertices $s, t \in G$.

$\Pi(s,t)$ takes $O\left(\frac{\min\{|B(s)|,|B(t)|\}+K}{DB}\right)$ I/Os. Next, we show now that $\sum_{v\in G}|B(v)| = \sum_{b\in G}|SPT(b, G(v(b)))|$.

Let w be a vertex in a shortest path tree $SPT(b, G(v(b)))$. Then $w \in G(v(b))$ and $b \in S(v(b))$. Thus, $v(b) \in V(w)$ and $b \in B(w)$. That is, each vertex w in a shortest path tree $SPT(b, G(v(b)))$ has a corresponding entry in the list $B(w)$. This entry cannot correspond to another vertex w' in a shortest path tree $SPT(b', G(v(b')))$. □

2.4 Analysis

In this section, we show that each list $B(w)$ has size $O(\sqrt{N})$, provided that we apply a $\sqrt{N}$-separator theorem to construct the hierarchical decomposition of G.

LEMMA 2.5 *Assume that a $\sqrt{N}$-separator theorem holds for the given graph G and all its subgraphs and that we apply this theorem to construct the hierarchical decompositon of G. Then $|B(w)| = O(\sqrt{N})$, for each vertex $w \in G$.*

PROOF. Let w be a vertex of G and $v_1, \ldots, v_k$ be the nodes of $T(G)$ in $V(w)$ in top-down order. Then $G(v_1) = G$. The separator theorem implies that $|V(G(v_i))| \leq \frac{2}{3}|V(G(v_{i-1}))|$, for $1 < i \leq k$, and $|S(v_i)| \leq c\sqrt{|V(G(v_i))|}$, for some positive constant c and $1 \leq i \leq k$. This implies that

$$
\begin{aligned}
|B(w)| &= |\bigcup_{i=1}^{k} S(v_i)| \\
&= \sum_{i=1}^{k} |S(v_i)| \\
&\leq \sum_{i=1}^{k} c\sqrt{|V(G(v_i))|} \\
&\leq c\sum_{i=1}^{k} \sqrt{\left(\frac{2}{3}\right)^{i-1} |V(G)|} \\
&= c\sum_{i=1}^{k} \left(\frac{2}{3}\right)^{\frac{i-1}{2}} \sqrt{|V(G)|} \\
&\leq c\sum_{i=0}^{\infty} \left(\frac{2}{3}\right)^{\frac{i}{2}} \sqrt{|V(G)|}
\end{aligned}
$$

$$= c\frac{1}{1-\sqrt{\frac{2}{3}}}\sqrt{|V(G)|}$$
$$= O(\sqrt{N}).$$

□

COROLLARY 2.1 *Given a graph G of size N such that a $\sqrt{N}$-separator theorem holds for G and all its subgraphs, we can construct a data structure stored in $O(N^{3/2}/B)$ blocks that allows for answering distance queries between two query vertices s and t in G with $O(\sqrt{N}/DB)$ I/Os. Reporting the corresponding shortest path takes $O(K/DB)$ additional I/Os, where K is the length of the reported path.*

PROOF. This follows immediately from Lemmas 2.3, 2.4, and 2.5. □

2.5 Constructing the Data Structure

In this section, we show how to construct the shortest path data structure described in this chapter efficiently, using algorithms presented in subsequent chapters.

Algorithm 2.3 shows the code of the construction algorithm. To understand it, we have to explain the input and output representations of procedures SEPARATE and EM_SSSP first.

Initially, G is given as a vertex list $V(G)$ and an edge list $E(G)$, represented as arrays. Procedure SEPARATE computes a separator S of G and identifies the connected components of $G \setminus S$. It labels all vertices $v \in S$ with $C(v) = 0$ and all edges e incident to a vertex $v \in S$ with $C(e) = \infty$. All other vertices and edges are labeled with values $C(v) > 0$ (resp. $C(e) < \infty$) that identify the connected components containing in this vertex or edge. Eventually, it sorts $V(G)$ (resp. $E(G)$) by increasing labels $C(v)$ (resp. $C(e)$). That is, let $G_1, \dots, G_r$ be the connected components of $G \setminus S$. Then SEPARATE sorts the vertices and edges of G such that $V(G)$ is stored as $S, V(G_1), \dots, V(G_r)$, and $E(G)$ is stored as $E(G_1), \dots, E(G_r), (E(G) \setminus E(G \setminus S))$. The details of this procedure are provided in Chapter 6.

ConstructSP(G):

Input: A graph G as a vertex array $V(G)$ and an edge array $E(G)$.

Output: Distance arrays and shortest path trees as described in this chapter.

```
1: Push the pair (G, 0) on the stack.
   {We do not actually push G on the stack. We represent G by pointers to V(G)
   and E(G) and sizes |V(G)| and |E(G)|.}
2: while the stack is not empty do
3:    Pop the pair (G', s) from the stack.
4:    if |V(G')| > 1 then
5:       Separate(G')
6:    end if
7:    Scan V(G') to construct shortest path trees for all separator vertices:
      {v is the current vertex.}
         if C(v) = 0 then
           {v is a separator vertex.}
           EM_SSSP(v, G', s)
         else
           Stop scanning.
         end if
8:    if |V(G')| > 1 then
9:       Continue scanning V(G') to push the connected components of G' − C on
         the stack:
         {v is the current vertex.}
            while we have not reached the end of V(G') do
              V(G'') ← {all vertices with the same component label as v}
              E(G'') ← {all edges with the same component label as v}
              Push the pair (G'', s + 1) on the stack.
            end while
10:   end if
11: end while
12: Construct the lists B:
      Sort the tuples (s, b, w, d, p) by increasing w, s, and b.
      Scan B and keep only one of multiple tuples with the same values s, b, and w.
13: Construct the address array A:
      Scan B. For each w, add the address of the first entry in B with label w to A. This is the
      address of B(w).
```

Algorithm 2.3: Constructing the shortest path data structure.

Procedure EM_SSSP is an augmented version of the external-memory single source shortest paths algorithm due to Crauser, Mehlhorn, and Meyer [13]. It computes the shortest path tree $SPT(b, G')$ for its input vertex b in the graph G' and blocks it using Algorithm 5.1. For each vertex w' in the blocked (!) tree, it writes a tuple (s, b, w, d, p) to disk. Here, s is the depth of G' in $T(G)$, b is the given vertex b, w is the vertex of G of which w' is a copy, $d = dist(b, w)$, and p is a pointer to w'. Note that, when blocking $SPT(b, G')$, we can easily postpone writing the last non-full block of $SPT(b, G')$ together with writing the corresponding tuples until we block the next shortest path tree. We just keep this information in a buffer in internal memory. Thus, we end up with a blocking of the forest of all shortest path trees instead of blocking each shortest path tree separately. Hence, we can apply Corollary 3.2 to estimate the used storage. Doing this, however, we have to use a version of Algorithm 5.1 without Step 7. The transformation of parent names back to parent pointers has to be done for the whole forest by an explicit invokation of Algorithm 5.3.

Now we can state the following lemma.

LEMMA 2.6 *Algorithm 2.3 constructs the shortest path data structure described in this chapter with* $O\left(\frac{N^{3/2}}{D} + \frac{N^{3/2}}{DB} \log_{\frac{M}{B}} \frac{N}{B}\right)$ *I/Os with high probability, for* $\sqrt{N} \frac{\log \frac{N}{B}}{\log \frac{M}{B} \log \frac{N}{M}} \geq DB$.

PROOF. Steps 1–11 of Algorithm 2.3 implement a depth-first search in $T(G)$. We start by pushing G on the stack. In the WHILE-loop, we pop the topmost component G' labeled with its depth in $T(G)$ from the stack. In Step 5, we separate it into smaller components. These components are pushed on the stack in Step 9.

Step 9 is correct because, in Step 7, we have already scanned the separator of G'. What remains of $V(G')$ are the vertices in $G' - S$. They are sorted by increasing component labels. The edges in $E(G')$ are sorted by increasing component labels as well. Thus, we can easily identify $V(G'')$ and $E(G'')$, for each component G'' of $G' - S$. We push the addresses and sizes of these two sets on the stack.

Thus, our algorithm constructs the hierarchical decomposition of G correctly. In Step 7, we construct the shortest path tree $SPT(b, G')$, for each separator vertex b of G'.

Procedure EM_SPPP writes tuples (s, b, w, d, p) to disk, one per vertex in the blocked shortest path trees. As there is a one-to-one correspondence between entries in B and vertices in the shortest path trees, these tuples are all we need to construct B. We only have to sort them by increasing w, s, and b. The primary key ensures that $B(w)$ is stored contiguously, for each w. The secondary key ensures that each $B(w)$ is sorted top down according to the position of each vertex in $T(G)$. The tertiary key ensures a fixed order in the lists $S(v)$, where $v \in T(G)$. This matches exactly the definition of the list B. However, as the shortest path trees are blocked, there are multiple tuples with the same values w, s, and b. As they are stored consecutively now, we scan B and keep only one of such multiple tuples.

Eventually, the start position of $B(w)$ is the address of the first tuple in B with label w. Thus, we just scan B. Whenever, we find a tuple with a new label w, we write its address to the array A.

We have shown that we construct the shortest path trees and arrays B and A correctly. Now let us analyze the I/O-complexity of Algorithm 2.3. We divide the I/Os performed into five parts: (1) stack operations, (2) output generation, (3) graph separation, (4) shortest path computation, and (5) rearranging of the lists $S(w), w \in G$.

The number of stack operations is bounded by N because each node of $T(G)$ stores at least one node of G. Each node of $T(G)$ is pushed on the stack and popped form the stack exactly once. Thus, the stack operations take $O(N/DB)$ I/Os.

The output consists of two parts: the shortest path trees, and the lists $B(w), w \in G$. Both parts of the output are written to disk, block by block, either by the blocking algorithm, or by sequentially writing the tuples (s, b, w, d, p) to disk. Thus, if N' is the output size, this takes $O(scan(N')) = O(N'/DB) = O(N^{3/2}/DB)$ I/Os.

As for the graph separation and the shortest path computations, we assume that, if $|G'| \leq M$, we perform all computation in internal memory. Stack operations are still performed in external memory.

Using the result from Chapter 6, we obtain the following recurrence for the number

of I/Os that it takes to compute all separators of the hierarchical decomposition:

$$\mathcal{I}(N) = \begin{cases} O(N/DB) & \text{if } N \leq M \\ O(N) + \sum_{i=1}^{k} \mathcal{I}(N_i) & \text{if } N > M, \end{cases}$$

where $N_i \leq \frac{2}{3}N$ for $1 \leq i \leq k$ and $N - a\sqrt{N} \leq \sum_{i=1}^{k} N_i \leq N$. That is, for $N \geq (2a)^{2/3}$, $\frac{N}{2} \leq \sum_{i=1}^{k} N_i \leq N$. The solution to this recurrence is $\mathcal{I}(N) = O\left(N \log \frac{N}{M}\right)$[1]. For $N \leq M$, $N/DB < N \log \frac{N}{M}$. For $N \geq \max\{M, (2a)^{2/3}\}$, we obtain

$$\begin{aligned}
\mathcal{I}(N) &\leq cN + \sum_{i=1}^{k} c' N_i \log \frac{N_i}{M} \\
&= cN + \sum_{i=1}^{k} c' \log \frac{3}{2} N_i \log_{3/2} \frac{N_i}{M} \\
&\leq cN + \sum_{i=1}^{k} c' \log \frac{3}{2} N_i (\log_{3/2} \frac{N}{M} - 1) \\
&\leq cN - \frac{c'}{2} \log \frac{3}{2} N + \sum_{i=1}^{k} c' \log \frac{3}{2} N_i \log_{3/2} \frac{N}{M} \\
&\leq c' \log \frac{3}{2} N \log_{3/2} \frac{N}{M} \qquad \text{(We choose } c' = 2c/\log \tfrac{3}{2}.\text{)} \\
&= c' N \log \frac{N}{M}.
\end{aligned}$$

For the number of I/Os that it takes to compute shortest paths, we obtain the following recurrence using Crauser, Mehlorn, and Meyer's result [13]:

$$\mathcal{I}(N) = \begin{cases} O(N/DB) & \text{if } N \leq M \\ \sqrt{N} O(\frac{N}{D} + \frac{N}{DB} \log_{\frac{M}{B}} \frac{N}{B}) + \sum_{i=1}^{k} \mathcal{I}(N_i) & \text{if } N > M, \end{cases}$$

where $N_i \leq \frac{2}{3}N$ for $1 \leq i \leq k$ and $N - a\sqrt{N} \leq \sum_{i=1}^{k} N_i \leq N$. The solution to this recurrence is $\mathcal{I}(N) = O\left(\frac{N^{3/2}}{D} + \frac{N^{3/2}}{DB} \log_{\frac{M}{B}} \frac{N}{B}\right)$. For $N \leq M$, $N/DB \leq \frac{N^{3/2}}{D} +$

[1] We define $\log \frac{N}{M} = \max\left\{1, \log \frac{N}{M}\right\}$, where the logarithm on the right hand side is defined as usual.

$\frac{N^{3/2}}{DB} \log_{\frac{M}{B}} \frac{N}{B}$. For $N > M$, we get

$$\begin{aligned}
\mathcal{I}(N) &\leq c\left(\frac{N^{3/2}}{D} + \frac{N^{3/2}}{DB}\log_{\frac{M}{B}}\frac{N}{B}\right) + \sum_{i=1}^{k}\mathcal{I}(N_i)\\
&\leq c\left(\frac{N^{3/2}}{D} + \frac{N^{3/2}}{DB}\log_{\frac{M}{B}}\frac{N}{B}\right) + \sum_{i=1}^{k}c'\left(\frac{N_i^{3/2}}{D} + \frac{N_i^{3/2}}{DB}\log_{\frac{M}{B}}\frac{N_i}{B}\right)\\
&\leq c\left(\frac{N^{3/2}}{D} + \frac{N^{3/2}}{DB}\log_{\frac{M}{B}}\frac{N}{B}\right) + \sum_{i=1}^{k}c'\left(\frac{N_i^{3/2}}{D} + \frac{N_i^{3/2}}{DB}\log_{\frac{M}{B}}\frac{N}{B}\right)\\
&\leq c\left(\frac{N^{3/2}}{D} + \frac{N^{3/2}}{DB}\log_{\frac{M}{B}}\frac{N}{B}\right) + \sum_{i=1}^{k}c'\left(\frac{N_i\sqrt{\frac{2}{3}N}}{D} + \frac{N_i\sqrt{\frac{2}{3}N}}{DB}\log_{\frac{M}{B}}\frac{N}{B}\right)\\
&\leq c\left(\frac{N^{3/2}}{D} + \frac{N^{3/2}}{DB}\log_{\frac{M}{B}}\frac{N}{B}\right) + c'\left(\frac{N\sqrt{\frac{2}{3}N}}{D} + \frac{N\sqrt{\frac{2}{3}N}}{DB}\log_{\frac{M}{B}}\frac{N}{B}\right)\\
&= \left(c + \sqrt{\frac{2}{3}}c'\right)\left(\frac{N^{3/2}}{D} + \frac{N^{3/2}}{DB}\log_{\frac{M}{B}}\frac{N}{B}\right)\\
&= c'\left(\frac{N^{3/2}}{D} + \frac{N^{3/2}}{DB}\log_{\frac{M}{B}}\frac{N}{B}\right) \qquad \text{(for } c' = c/(1-\sqrt{\tfrac{2}{3}})\text{)}.
\end{aligned}$$

Finally, it costs $O(sort(N^{3/2}))$ I/Os to create the lists $B(w)$.

For $\sqrt{N}\frac{\log\frac{N}{B}}{\log\frac{M}{B}\log\frac{N}{M}} \geq DB$, the I/O-complexities of all steps are dominated by the I/O-complexity for computing all shortest path trees. Thus, Algorithm 2.3 takes $O\left(\frac{N^{3/2}}{D} + \frac{N^{3/2}}{DB}\log_{\frac{M}{B}}\frac{N}{B}\right)$ I/Os with high probability. □

THEOREM 2.1 *Let a graph G of size N be given. Assume that a $\sqrt{N}$-separator theorem holds for G and all its subgraphs. Then we can construct a data structure stored in $O(N^{3/2}/B)$ blocks that allows for answering distance queries between two query vertices s and t in G with $O(\sqrt{N}/DB)$ I/Os. Reporting the corresponding shortest path takes $O(K/DB)$ I/Os, where K is the length of the reported path. The construction of this data structure takes $O\left(\frac{N^{3/2}}{D} + \frac{N^{3/2}}{DB}\log_{\frac{M}{B}}\frac{N}{B}\right)$ I/Os with high probability, for $\sqrt{N}\frac{\log\frac{N}{B}}{\log\frac{M}{B}\log\frac{N}{M}} \geq DB$.*

PROOF. This follows immediately from Corollary 2.1 and Lemma 2.6. □

Chapter 3

Blocking Rooted Trees: The Storage Scheme

In this chapter, we consider the problem of blocking rooted trees for fast path traversals. This is motivated by the desire to retrieve shortest paths quickly from the shortest path trees that are part of the data structure described in Chapter 2.

More precisely, storing a rooted tree in external memory as we do in internal memory — that is, with pointers between vertices and the vertices at fairly arbitrary memory positions — we must expect to perform one I/O operation per vertex of the traversed path. Previous results by Nodine, Goodrich, and Vitter [28] show that we can do better. We show that, for traversals toward the root, we can beat the bounds proved in [28]. Since we can store B vertices per block and we have D parallel disks, the optimum would be one I/O per DB vertices. We show that, for offline traversals toward the root, we can achieve $O(1)$ I/Os per DB vertices at the expense of increasing the space used to store the tree by a constant factor.

3.1 Path Traversals

When talking about traversing paths in rooted trees, there are different scenarios to think about. First, we distinguish between *online* and *offline* traversals. Traversing a path online, we are given only the start vertex. The data read at each vertex

determines which of its neighbours has to be visited next. We repeat this process until some final condition is reached. Traversing a path offline, we are given the whole path (at least implicity) in advance, once we know the start vertex.

The second distinction comes from restrictions on the edges of the tree. They might be undirected or directed. We assume that undirected edges can be traversed in both directions, while directed edges can only be traversed from their source vertices to their target vertices. We do not see any useful applications of the mixed case, where some edges are directed from parents to children and others from children to parents. Therefore, we only distinguish between the case where all edges are directed from children to parents and the case where all edges are directed from parents to children. In the case of undirected edges, we call the traversal *unrestricted*; if edges are directed from parents to children we call it *top-down*; if edges are directed from children to parents we call it *bottom-up*.

Note that there are no online bottom-up traversals because every vertex has exactly one parent except for the root of the tree, which has no parent. Therefore, given a start vertex v at distance d from the root of the tree and a path length $l \leq d$, there is exactly one bottom-up path of length l starting at v. If no length l is given, the query is to report the path from v to the root, which is equivalent to querying the bottom-up path of length $l = d$ starting at v.

Also, note that, as long as every vertex is stored exactly once or we do not modify the information in the tree, there is no difference between offline bottom-up and offline top-down traversals: For a top-down traversal, we can traverse the tree bottom-up, thereby collecting the information in reverse order, process the collected information in top-down order and, if necessary, write it back in another bottom-up traversal.

3.2 Speed-Up and Storage Blow-Up

In certain applications such as our shortest path data structure, we only want to read the data but not modify it. Then we may try to facilitate blockwise I/O by storing data items more than once. Of course, this requires more space than storing each item once. Hence, the two main parameters here are the *speed-up* σ and the *storage*

blow-up ρ. The speed-up σ of a given algorithm $\mathcal{A}$ is defined as the fraction of the number of data items accessed by $\mathcal{A}$ and the number of I/Os performed by $\mathcal{A}$. The storage blow-up ρ is the fraction of the number of blocks that $\mathcal{A}$ uses to store the data and the minimal number of blocks, $\lceil N/B \rceil$, for storing the N data items. Obviously, $1 \leq \sigma \leq DB$. Usually, there is a trade-off between the speed-up and the storage blow-up that we are willing to accept.

We state some results in terms of I/Os and others in terms of the achievable speed-up. Note that an upper (resp. lower) bound on the number of I/Os is equivalent to a lower (resp. upper) bound on the speed-up and vice versa.

3.3 Previous Results for Online Traversals

For unrestricted online traversals in perfect d-ary trees, Nodine, Goodrich, and Vitter [28] have used an adversary argument to prove an upper bound of $O(\log_d B)$[1] on the achievable worst-case speed-up in the case $D = 1$. This bound is independent of the storage blow-up. They also proved a matching lower bound of $\Omega(\log_d B)$ with only constant storage blow-up. The adversary argument used in [28] generalizes easily to multiple disks, giving an upper bound of $O(\log_d DB)$ on the speed-up achievable for unrestricted online traversals in the PDM. Using Lemma 1.1, the lower bound generalizes to $\Omega(\log_d(DB))$ in the PDM.

A slight modification of their arguments easily gives a $\Theta(\log_d(DB))$ bound for the achievable speed-up in the case of online top-down traversals.

Since there are no online bottom-up traversals, this exhausts the class of online traversals of rooted trees.

3.4 Offline Traversals

In this section, we consider bottom-up traversals, which are always offline. We prove speed-up bounds of $\Theta(D \log_d B)$ for perfect d-ary trees and $\Omega(\log_d DB)$ for arbitrary

[1] For convenience, we define $\log_b x = \max\{1, \lfloor \log_b x \rfloor\}$, where the logarithm on the right hand side is defined as usual.

d-ary trees in the case $\rho = 1$, i.e., if we do not accept any storage blow-up at all. These blockings are important for applications where the data stored at the nodes of the tree are modified during the traversals. Our main result of Section 3.4.2 does not allow for efficient updating of all copies of a modified vertex. It is an interesting open problem to find a better way to speed up modifying offline traversals using another blocking scheme than the one proposed in this chapter.

In Section 3.4.2, we prove the main result of this chapter. We show that one can achieve a speed-up $\sigma = \Theta(DB)$ for offline bottom-up traversals in arbitrary rooted trees with only constant storage blow-up. Note that this main result is independent of the maximal degree d of the given tree.

Using the observation at the end of Section 3.1, our bounds hold for offline top-down traversals in the respective trees under the respective conditions as well.

3.4.1 Tight Bounds without Storage Blow-Up

If we do not accept any storage blow-up, i.e., $\rho = 1$, we have the following result.

LEMMA 3.1 *Given an arbitrary d-ary tree of size N stored in $\lceil N/B \rceil$ blocks on D parallel disks, the best achievable worst-case speed-up for the traversal of a path from a vertex v to an ancestor w of v in T is $\Omega(\log_d DB)$*

PROOF. By Lemma 1.1, we can assume that we are given a single disk with block size DB.

Let h be the height of T and $h' = \log_d DB$. We cut T into layers $L_0, \dots, L_{\lceil h/h' \rceil - 1}$, where layer L_i is the subgraph of T induced by all vertices of T at distances ih' through $(i+1)h' - 1$ from the root of T. Each layer is a forest of rooted trees of height at most h'. Denote all these subtrees of T by $T_1, \dots, T_r$. Each such subtree T_i has size at most $\frac{d^{h'}-1}{d-1} \le \frac{\max\{d,DB\}-1}{d-1} \le DB$. (See Figure 3.1.)

We store the vertices of T so that the vertices of each T_i are stored consecutively. This means that we can access each T_i with at most 2 I/Os because the at most DB vertices of T_i are distributed over at most 2 blocks.

A bottom-up path p of length k crosses at most $\lceil k/h' \rceil + 1$ different layers of T. Moreover, the subpath of p within a layer L_i is contained completely in some subtree

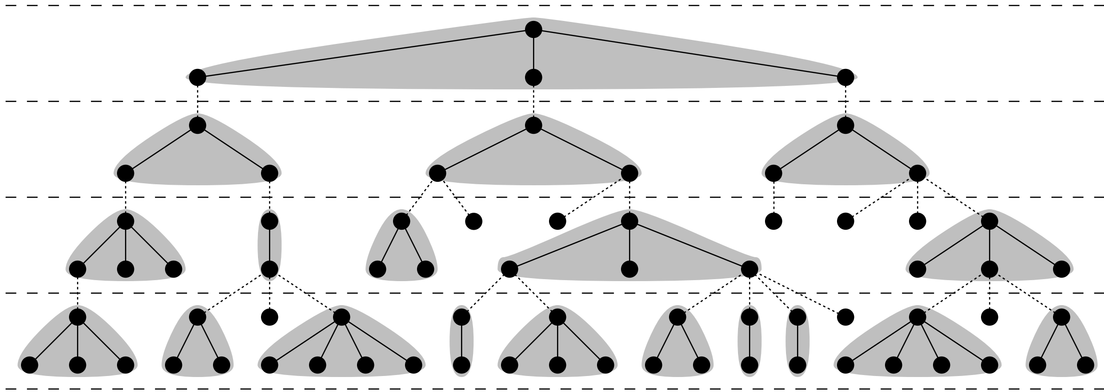

Figure 3.1: A given tree T of maximal degree $d = 4$ that we cut into layers of height 2. We cut the tree along the dashed lines thereby removing all dotted edges. The resulting trees are shaded.

T_j of L_i. That is, p crosses at most $\lceil k/h' \rceil + 1$ subtrees T_j of T, each of which can be accessed with at most 2 I/Os.

Thus, we need at most $2(\lceil k/h' \rceil + 1) = 2(\lceil k/log_d DB \rceil + 1) = O(k/\log_d DB)$ I/Os to traverse p. This means that the speed-up is $\Omega(\log_d DB)$. □

If the given tree is perfect, we can show a better result. We also show a matching upper bound, which is also an upper bound for arbitrary d-ary trees. In order to achieve a better lower bound for perfect d-ary trees, we rely on the regular structure of these trees. This allows us to take advantage of the independence of the D disks that we use. Using Lemma 1.1, we would tie groups of D blocks, one per disk, rigidly together. Here, we keep the blocks indepent from each other and group them according to our needs only when we are about to read them with the next I/O.

Lemma 3.2 *Given a perfect d-ary tree T of size N stored in $\lceil N/B \rceil$ blocks on D parallel disks, the best achievable worst-case speed-up for the traversal of a path from a vertex v to an ancestor w of v in T is $\Omega(D \log_d B)$.*

Proof. We use the same proof idea as for Lemma 3.1. That is, we cut T into $\lceil h/h' \rceil$ layers of height h', where h is the height of T; but this time, $h' = \log_d B$.

The perfectness of T implies that all subtrees except those in the bottom layer have height exactly h' and size $S' = \frac{d^{h'}-1}{d-1} \leq B$. All trees in the bottom layer have

height $h'' = (h \bmod h') \leq h'$ and size $S'' = \frac{d^{h''}-1}{d-1} \leq S' \leq B$.

Assume that we store all subtrees of layer L_i on disk $\mathcal{D}_{i \bmod D}$ such that the vertices of each subtree are stored consecutively.

Again, a path p of length k crosses at most $\lceil k/h' \rceil + 1$ subtrees T_i, one per crossed layer. Denote these trees by $T_{i_0}, \dots, T_{i_{\lceil k/h' \rceil}}$ so that p crosses them in this order.

Assume that we have a method to compute, for each tree crossed by p, its disk number and address on this disk without any extra I/Os. D consecutive trees $T_{i_{jD}}$ through $T_{i_{(j+1)D-1}}$, $j \in \mathbb{N}$ belong to D consecutive layers and are therefore stored each on another disk. That is, we can access these D trees with at most 2 I/Os because the vertices of each tree are distributed over at most 2 blocks on the same disk. Hence, traversing p takes at most $2\lceil(\lceil k/h' \rceil + 1)/D\rceil = 2\lceil(\lceil k/\log_d B\rceil + 1)/D\rceil = O(k/D \log_d B)$ I/Os. The speed-up is $\Omega(D \log_d B)$. It remains to describe such a method for computing the addresses of the trees crossed by p. This method relies crucially on the regular shape of T and would not work for arbitrary d-ary trees.

We define the size $|L_i|$ of layer L_i as the number of vertices in it. If L_i is the bottom layer, then L_i contains all vertices at levels ih' through h in T. Otherwise, it contains all vertices at levels ih' through $(i+1)h' - 1$. That is, for the bottom layer we have

$$\begin{aligned} |L_i| &= \frac{d^h - 1}{d-1} - \frac{d^{ih'} - 1}{d-1} \\ &= \frac{(d^{h-ih'} - 1)d^{ih'}}{d-1} \\ &= \frac{d^{h''} - 1}{d-1} d^{ih'} = S'' d^{ih'}. \end{aligned}$$

For all other layers we obtain

$$\begin{aligned} |L_i| &= \frac{d^{(i+1)h'} - 1}{d-1} - \frac{d^{ih'} - 1}{d-1} \\ &= \frac{(d^{h'} - 1)d^{ih'}}{d-1} \\ &= \frac{d^{h'} - 1}{d-1} d^{ih'} = S' d^{ih'}. \end{aligned}$$

If we store all layers stored on disk $\mathcal{D}_i$ in top-down order, the first vertex of layer L_j, where $j = i + kD$, is stored at position

$$\begin{aligned}
\sum_{l=0}^{k-1} |L_{i+lD}| &= \sum_{l=0}^{k-1} S' d^{(i+lD)h'} \\
&= S' d^{ih'} \sum_{l=0}^{k-1} \left(d^{Dh'}\right)^l \\
&= S' d^{ih'} \frac{\left(d^{Dh'}\right)^k - 1}{d - 1} \\
&= S' d^{ih'} \frac{d^{kDh'} - 1}{d - 1} \\
&= S' \frac{d^{(i+kD)h'} - d^{ih'}}{d - 1} \\
&= S' \frac{d^{jh'} - d^{(j \bmod D)h'}}{d - 1}.
\end{aligned}$$

Note that the size of each layer L_{i+lD} appearing in the sum is indeed $S'd^{(i+lD)h'}$ because the bottom layer is the last layer on its disk. That is, it is not stored before any other layer.

Let the trees of each layer L_i be numbered as $T_{i,0}, \ldots, T_{i,n_i}$ from left to right, where $n_i = d^{ih'}$. Let them also be stored in this order. Then, the address of tree $T_{i,j}$ is

$$S' \frac{d^{ih' - d^{i \bmod D}}}{d - 1} + S'j,$$

if L_i is not the bottom layer, and

$$S' \frac{d^{ih' - d^{i \bmod D}}}{d - 1} + S''j,$$

otherwise.

We number and store the vertices of each tree in preorder (starting with number 0). Then we can define the following labels for each vertex v: the layer number, $L(v)$, of the layer containing v, the tree number, $T(v) = j$, of the tree $T_{i,j}$ containing v, the preorder number, $n_p(v)$, of v in $T_{i,j}$, and a flag, $B(v)$, telling whether $L_{L(v)}$ is the bottom layer.

The address of v can now be computed as

$$A(v) = S'\frac{d^{L(v)h' - d^{L(v) \bmod D}}}{d-1} + S(v)T(v) + n_p(v), \text{ where}$$
$$S(v) = \begin{cases} S'' & \text{if } L_{L(v)} \text{ is the bottom layer} \\ S' & \text{otherwise} \end{cases}.$$

Originally, we would assume that we were given a pointer to v in T when querying a bottom-up path p in T starting at v. However, this pointer must have been created by some preprocessing algorithm and must be stored somewhere. Thus, instead of computing and storing this pointer, we can as well make the preprocessing algorithm compute and store labels $L(v), T(v), n_p(v)$, and $B(v)$ for v. Then a path p from v to the root of T crosses layers $L_{L(v)}, \ldots, L_0$. The tree crossed in layer L_i is $T_{i,\lfloor j/d^{(L(v)-i)h'} \rfloor}$. Using the formula above, we can easily compute the address of each such subtree $T_{i,j}$. This proves that we can indeed achieve a speed-up of $\Omega(D \log_d B)$.

There is a small technical detail that we have to deal with. As we have D disks, this method might leave us with D non-full blocks, one per disk. Then the total number of blocks would be more than $\lceil N/B \rceil$. We can collect the at most DB vertices stored in these blocks and store them in an optimal number of blocks, at most one per disk. Before starting the actual path traversal, we load these blocks into internal memory with a single I/O and keep them in internal memory. This makes these vertices accessible without any additional I/Os. Thus, the compression of the originally D blocks into a smaller number of blocks does not cause any problems. □

We want to note here that the method to achieve the lower bound in the proof of Lemma 3.2 might be disputable because the blocks containing the tree are not evenly distributed over the disks. In general, this is certainly an issue. It is of less importance for applications such as our shortest path data structure, where we have to store many shortest path trees. There we can try to achieve a good distribution of the data by storing layer L_j of the i-th shortest path tree on disk $\mathcal{D}_{(i+j) \bmod D}$ instead of disk $\mathcal{D}_{j \bmod D}$. Of course, this heuristic does not guarantee a perfectly even distribution of the data; but we expect this simple approach to work well in practice.

Next, we prove a matching upper bound.

LEMMA 3.3 *Given a perfect d-ary tree T of size N stored in $\lceil N/B \rceil$ blocks on D parallel disks, the best achievable worst-case speed-up for the traversal of a path from a vertex v to an ancestor w of v in T is $O(D \log_d B)$.*

PROOF. The number of I/Os required to traverse a given path p is no less than the number of blocks over which the vertices of p are distributed, divided by the number of disks. Note that, as each vertex is stored exactly once, this number of blocks is well-defined. We show that there is a path from a leaf of T to the root of T such that its vertices are distributed over at least $\Omega\left(\frac{\log_d N}{\log_d B}\right)$ blocks. This path has length $\log_d N$. Since we can read at most D blocks with a single I/O, this implies the claimed worst-case speed-up of $O(D \log_d B)$.

We can imagine each block as having a unique color. A vertex v has color c iff c is the color of the block containing v. Since every vertex is stored exactly once, the color of each vertex is unique. Now counting the number of blocks over which a path is distributed is equivalent to counting the number of different colors appearing on this path.

We divide T into layers of height $\lceil \log_d B \rceil + 1$. This divides T into subtrees of height $\lceil \log_d B \rceil + 1$.

Let $p' = (v, \ldots, w)$ be a subpath of the worst-case path p that we are going to construct, where v is the root of such a subtree T' and w is a leaf of T'. We say that p' *introduces* color c into p, if c appears on p' and no ancestor of v in T has color c. We say that p' is a *worst* path, if no path in T' would introduce more colors than p'.

We construct the worst-case path p in T in a greedy way. First, we look at the subtree T' rooted at the root of T and choose a worst path $p_1 = (v_1, \ldots, w_1)$ in T'. Then, we choose a child v_2 of w_1 in T and a worst path, p_2, in the subtree rooted at this child, and so on. Finally, p is the concatenation of $p_1, \ldots, p_k$, $k = \frac{\log_d N}{\lceil \log_d B \rceil + 1}$.

We maintain a potential function Φ. It represents the number of vertices that can be colored without introducing a new color into p. Initially, $\Phi = 0$, and clearly, $\Phi \geq 0$ at any time.

If a subpath p_i introduces $c > 0$ new colors, we increase Φ by cB. If p_i does not introduce a new color, then all vertices in the corresponding subtree must have been

colored with "old" colors because p_i is a worst path. That is why we decrease Φ by B in this case, as $|T'| = \frac{d^{\lceil \log_d B \rceil + 1} - 1}{d-1} \geq \frac{dB-1}{d-1} \geq \frac{dB-B}{d-1} = B$.

Assume that there are less than $\lceil k/2 \rceil$ subpaths that do not introduce new colors. Then, at least $\lceil k/2 \rceil$ subpaths introduce new colors, and p contains at least $\left\lceil \frac{\log_d N}{2(\lceil \log_d B \rceil + 1)} \right\rceil = \Omega\left(\frac{\log_d N}{\log_d B}\right)$ different colors.

Otherwise (if there are at least $\lceil k/2 \rceil$ such paths), these at least $\lceil k/2 \rceil$ subpaths have decreased Φ by at least $\lceil k/2 \rceil B$. However, at the end, $\Phi \geq 0$, initially $\Phi = 0$, and each newly introduced color increases Φ by B. Hence, there must be at least $\lceil k/2 \rceil$ different colors in p and, as above, p contains $\Omega\left(\frac{\log_d N}{\log_d B}\right)$ different colors. □

3.4.2 An Optimal Approach with Storage Blow-Up

In this section, we show how to achieve a speed-up $\sigma = \Theta(DB)$ with only constant storage blow-up not only for perfect d-ary trees, but for arbitrary rooted trees. Note that this means in particular that speed-up as well as storage blow-up are independent of the maximal degree of the vertices in the tree. It is also worth mentioning that the constants involved are small. More precisely, the speed-up is $\sigma \approx DB/3$, while $\rho \leq 5$.

We use the same proof idea as for Lemma 3.1. That is, we divide T into layers $L_0, \ldots, L_{\lceil h/h' \rceil - 1}$, this time of height $h' = DB/3$. We observed already that each such layer is a forest of rooted trees of height at most $h' = DB/3$. Let $T_1, \ldots, T_r$ be the set of these trees for all layers of T. We show how to divide each such subtree T_i into subtrees $T_{i,1}, \ldots, T_{i,s}$ so that the following properties hold:

1. $|T_{i,j}| \leq DB$, for all $1 \leq j \leq s$,
2. $\sum_{j=1}^{s} |T_{i,j}| \leq 2.5|T_i|$, and
3. For each leaf l of T_i, there is a tree $T_{i,j}$ such that the whole path from l to the root of T_i is contained in $T_{i,j}$.

We prove the following lemma.

LEMMA 3.4 *If we can divide a given rooted tree T into subtrees $T_{i,j}$ so that Properties 1–3 hold, this immediately translates into a blocking of T with storage blow-up at most*

5 such that traversing any bottom-up path of length k in T takes at most $\lceil 3k/DB \rceil + 1$ I/Os.

PROOF. By Lemma 1.1, we can again use the D parallel disks as one large disk with block size DB.

Property 1 ensures that we can store each $T_{i,j}$ in a single block. Therefore, Property 3 implies that, if we choose the right subtree $T_{i,j}$, we can read a whole leaf-to-root path in T_i with a single I/O. In fact, with our approach, each leaf l of T_i is stored exactly once. Thus, reading the block containing l, we automatically choose the right $T_{i,j}$ containing the whole path from l to the root of T_i.

Each bottom-up path p of length k in T spans at most $\lceil 3k/DB \rceil + 1$ layers. That is, we can divide p into maximal subpaths p_i each of which stays inside a single layer. Each p_i is a leaf-to-root path in a subtree T_i or a subpath thereof. Therfore, it can be accessed with a single I/O. Thus, reading p takes $\lceil 3k/DB \rceil + 1$ I/Os.

It remains to bound the storage blow-up caused by this subdivision of T. Storing each subtree $T_{i,j}$ in a single block, we might end up with many sparsely populated blocks because we do not have a lower bound on the sizes of these subtrees. However, we can use the following block-merging argument to achieve a storage blow-up of at most 5:

Initially, we store each $T_{i,j}$ in a single block. As long as we have at least two blocks holding at most $DB/2$ vertices each, we keep merging pairs of such blocks. This works because two such blocks together hold at most DB vertices. After merging blocks this way, we are left with at most one block that contains at most $DB/2$ vertices. If there is such a block, we check whether we can merge it with one of the more than half full blocks. If so, we merge these two blocks.

Let $\mathcal{B}_1, \ldots, \mathcal{B}_q$ be the set of blocks holding the subtrees $T_{i,j}$ of T after the merging. Let $\mathcal{B}_q$ be the possibly at most half full block. Since, by assumption, we cannot merge $\mathcal{B}_q$ and $\mathcal{B}_{q-1}$, they contain a total of more than DB vertices. Thus, on average, each of them contains more than $DB/2$ vertices. Each of the blocks $\mathcal{B}_1, \ldots, \mathcal{B}_{q-2}$ contains more than $DB/2$ vertices, by the above construction. Hence, all blocks together contain more than $q\frac{DB}{2}$ vertices. On the other hand, by Property 2, they hold at

most

$$\begin{aligned}\sum_{i=1}^{r} 2.5|T_i| &= 2.5 \sum_{i=1}^{r} |T_i| \\ &= 2.5|T|\end{aligned}$$

vertices, so that

$$\begin{aligned} q\frac{DB}{2} &< 2.5|T| \\ q &< 5\frac{|T|}{DB}. \end{aligned}$$

Since $\left\lceil \frac{|T|}{DB} \right\rceil$ is the minimum number of blocks to store T, we have shown that the storage blow-up for storing T this way is at most 5. □

Note that neither when dividing T into layers nor in the proof of Lemma 3.4, we used the fact that T is connected. Thus, Lemma 3.4 generalizes to forests of rooted trees, and we can state the following corollary.

Corollary 3.1 *If we can divide a given forest F of rooted trees into subtrees $T_{i,j}$ so that Properties 1–3 hold, this translates immediately into a blocking of F with storage blow-up 5 such that traversing any bottom-up path of length k in F takes at most $\lceil 3k/DB \rceil + 1$ I/Os.*

It remains to show that we can divide each tree T_i into subtrees $T_{i,j}$ so that Properties 1–3 hold. Lemma 3.5 states the existence of such a subdivision of T_i. We give a constructive proof. In this proof, we need the following two observations about preorder numbers, where we define that $u < v$, for two vertices u and v of the given tree, iff u has a smaller preorder number than v.

Observation 3.1 *Let a rooted tree T, a preorder numbering of the vertices of T, and two vertices $v \neq w$ of T be given, and let w be an ancestor of v. Then w has a smaller preorder number than v.*

Proof. Recall that preorder numbers are defined by choosing an Euler tour of T starting at the root of T and numbering the vertices according to the order of their first appearances on this tour.

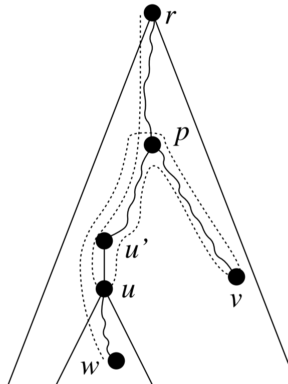

Figure 3.2: Illustrating the proof of Observation 3.2.

The only path from the root r of T to v is via w because there is a path from r to v via w and there is only one path between two vertices in a tree. Thus, the Euler tour starting at r by which the preorder numbers are defined must visit w for the first time before visiting v for the first time. Hence, w must have a smaller preorder number than v. □

Observation 3.2 *Let a rooted tree T, a preorder numbering of the vertices of T, and three distinct vertices u, v, and w of T be given. Let u be an ancestor of w and $u < v < w$. Then u is an ancestor of v.*

Proof. Assume that $u < v < w$ and u is not an ancestor of v. Let u' be the parent of u and p be the lowest common ancestor of u and v. The Euler tour defining the preorder numbers would have to use edge $\{u', u\}$ for the first time to get from the root r of T to u, in order to visit u for the first time. The only way from u to v is via $\{u', u\}$ because v is no descendant of u. That is, we have to use this edge a second time in order to visit v for the first time. Since $v < w$, we visit w for the first time after we have visited v for the first time. As w is a descendant of u, we have to use edge $\{u', u\}$ for a third time to visit w for the first time. (See Figure 3.2 for an illustration.) That is, we use edge $\{u', u\}$ at least three times in contradiction to the fact that an Euler tour uses each tree edge exactly two times, once in each direction. □

LEMMA 3.5 *Given a tree T_i of height at most $DB/3$, we can divide T_i into subtrees $T_{i,1}, \ldots, T_{i,s}$ such that Properties 1–3 hold.*

PROOF. If $|T_i| \le DB$, we "divide" T_i into only one subtree $T_{i,1} = T_i$, and Properties 1–3 are trivially true. So assume that $|T_i| > DB$. In this case, we label the vertices of T_i with their preorder numbers in T_i starting with number 0. Let v_k denote the vertex of T_i with preorder number k. We divide the vertex set of T_i into subsets $V_0, \ldots, V_{\lceil |T_i| \frac{3}{2DB} \rceil - 1}$, where $V_j = \{v_{j\frac{2DB}{3}}, \ldots, v_{(j+1)\frac{2DB}{3}-1}\}$. For each V_j, we define a subtree $T_{i,j} = T_i(V_j)$ of T_i consisting of all vertices in V_j and all their ancestors in T_i. We claim that the set of these subtrees $T_{i,j}$ satisfies Properties 1–3.

Property 3 is explicitly ensured by including the ancestors of each vertex of $T_i(V_j)$ into V_j. If we assume that Property 1 is true, Property 2 follows because

$$\begin{aligned} \sum_{i=0}^{\lceil |T_i| \frac{3}{2DB} \rceil - 1} |T_i(V_j)| &\le \sum_{i=0}^{\lceil |T_i| \frac{3}{2DB} \rceil - 1} DB \\ &= \left\lceil |T_i| \frac{3}{2DB} \right\rceil DB \\ &< \left(|T_i| \frac{3}{2DB} + 1 \right) DB \\ &= 1.5|T_i| + DB \\ &< 2.5|T_i|. \end{aligned}$$

It remains to prove Property 1.

Let $A(V_j)$ be the set of ancestors of $v_{j\frac{2DB}{3}}$. Then Property 1 follows, if we can prove that $T_i(V_j)$ contains only vertices in $A(V_j) \cup V_j$ because this implies that $|T_i(V_j)| \le |A(V_j)| + |V_j| \le DB/3 + 2DB/3 = DB$. The bound on the size of $A(V_j)$ follows from the fact that T_i has height at most $DB/3$ and each vertex of T_i has at most one ancestor per level.

By Observation 3.1 all vertices in $T_i(V_j)$ have preorder numbers at most $(j + 1)\frac{2DB}{3} - 1$. All vertices with preorder numbers $j\frac{2DB}{3}$ through $(j+1)\frac{2DB}{3} - 1$ are in V_j. So we are only interested in vertices with preorder numbers smaller than $j\frac{2DB}{3}$. Assume that we have a vertex v_k in $T_i(V_j)$ which is an ancestor of some vertex v_l, $k < j\frac{2DB}{3} < l$. Then, by Observation 3.2, v_k is an ancestor of $v_{j\frac{2DB}{3}}$ and thus in $A(V_j)$. This shows that each vertex in $T_i(V_j)$ is either in V_j or in $A(V_j)$. □

Putting things together, we have proved the following theorem.

THEOREM 3.1 *Given a rooted tree T, we can store T in at most $5D\left\lceil\frac{|T|}{DB}\right\rceil$ blocks evenly distributed over D parallel disks so that traversing any bottom-up path of length k in T takes at most $\left\lceil\frac{3k}{DB}\right\rceil+1$ I/Os.*

PROOF. This follows immediately from Lemmas 3.4 and 3.5. □

COROLLARY 3.2 *Given a forest F of rooted trees, we can store F in at most $5D\left\lceil\frac{|F|}{DB}\right\rceil$ blocks evenly distributed over D parallel disks so that traversing any bottom-up path of length k in F takes at most $\left\lceil\frac{3k}{DB}\right\rceil+1$ I/Os.*

PROOF. This follows immediately from Corollary 3.1 and Lemma 3.5. □

Chapter 4

Labeling Rooted Trees in External Memory

In this chapter, we describe I/O-efficient algorithms to compute certain common labels of the vertices of rooted trees such as for instance preorder numbers. We will use these algorithms as components of the algorithms presented in Chapters 5, 6, and 7.

We assume here that the tree T is given in the representation that we use in Chapter 5. That is, each vertex $v \in T$ is represented as a pair $(v, p(v))$, where $p(v)$ is the parent of v. Here, $p(v)$ is not a pointer to the memory location containing v's parent, but the name of v's parent. This is important because we keep shuffling the vertices of T around. Thus, we would have to update the pointers all the time. Using names instead of pointers, we avoid these update operations. On the other hand, we cannot access v's parent right away because we do not know where it is stored. As we will see, this does not do us any harm. For the root r of T, we set $p(r) = 0$, assuming, w.l.o.g., that no vertex of T is named 0.

Basically this means that we are given T as a set of tree edges, which are directed from children to parents. For the root, we can assume that it has the imaginary vertex 0 as its parent.

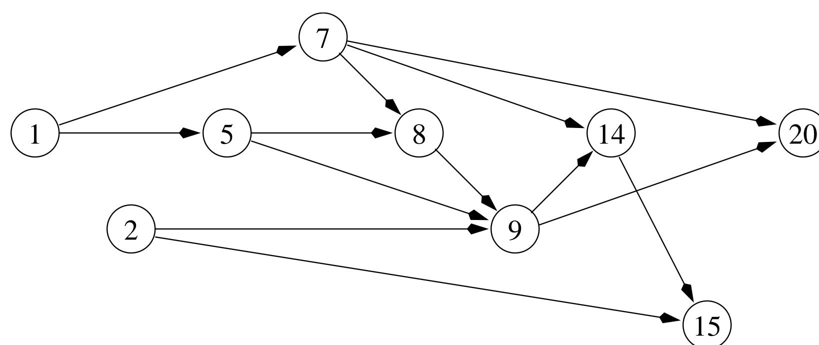

Figure 4.1: A circuit without cycles. Its vertices have been numbered such that $v < w$, for every edge $v \to w$. All edges are directed from left to right. Time-forward processing evaluates this circuit in a left-to-right sweep.

4.1 Time-Forward Processing

Time-forward processing is a technique to evaluate circuits that do not contain cycles. For such circuits, it is possible to number the nodes of the circuit so that, if node w needs the output of node v as input, $v < w$. Let the nodes be numbered this way and sorted by increasing numbers. We refer to such circuits as *topologically sorted* (see Figure 4.1).

We can evaluate a topologically sorted circuit by processing the nodes in their order of appearance. This guarantees that, whenever a node v is evaluated, the nodes the outputs of which are required as inputs for v have been evaluated before v.

We can imagine node v as being evaluated at time v. When evaluating v, we send v's output forward in time to every node w that needs v's output as input. This motivates the name of this technique.

The first algorithm for time-forward processing due to Chiang *et al.* [12] has two restrictions. Firstly, it works only for large values of m. Secondly, the fan-in of the evaluated circuit had to be bounded by some integer d. The latter is the more serious restriction for our application because we cannot say anything about the degrees of the nodes in our shortest path trees. Fortunately, Arge [5, 6] removes both restrictions by presenting an alternative and very elegant time-forward processing algorithm. This algorithm uses a priority queue that is based on the buffer tree [5, 6]. Arge still

assumes that, for each node v of the circuit, we are explicitly given v's indegree and the nodes that need v's output as input. This is reasonable when thinking exclusively about logical circuits. Using time-forward processing to implement operations on rooted trees, however, it will be handy to relax this assumption a little. First, we will review Arge's algorithm. Afterwards, we will see that, with minor modifications, this algorithm can also handle the following important cases:

- For each node v, we are explicity given the nodes that need v's output as input, but not v's fan-in.
- Each node has fan-in at most one. For each node v, we are explicitly given the node producing v's input. However, we do not know the vertices that need v's output as input.

Arge's time-forward processing algorithm evaluates the nodes of the circuit in their order of appearance. The first node v cannot need any input from another node because the vertices are topologically sorted. Thus, we can evaluate node v right away. We put its output into the priority queue, once for each node w that needs v's output as input. The copy of v's output meant for w is given a time stamp (priority) w. When we evaluate a subsequent node w of fan-in d, all nodes that produce an input for w have been evaluated. No other nodes have time-stamped their output with w. All entries with smaller priority in the priority queue have been used as inputs for the nodes before w. Thus, the d entries with smallest priorities in the priority queue are the inputs for w. We delete these d entries and obtain the necessary input values for w. We repeat this process until all nodes in the circuit have been evaluated.

For each edge of the circuit, this algorithm performs one *insert* and another *deletemin* on the priority queue. Thus, the total number of operations on the priority queue is $O(N)$, where N is the size of the circuit (nodes and edges). Arge proves the following theorem.

THEOREM 4.1 [5, 6] *A priority queue can be implemented in external memory so that a series of N insert, delete, and deletemin operations on the priority queue takes $O(sort(N))$ I/Os.*

COROLLARY 4.1 [5, 6] *A topologically sorted circuit C with a total of N nodes and edges can be evaluated with $O(sort(N))$ I/Os.*

Let us consider the case that, for each node v, we are given the nodes that need v's output as input, but not v's in-degree. Then we can produce outputs as usual. When reading inputs, we keep deleting minima from the priority queue until we have deleted an item with higher priority than that of the vertex that we are presently processing. This item is put back into the queue with the same priority. This way, we perform two additional operations per node. The total number of operations on the queue is still $O(N)$.

In the second special case, each node has fan-in at most one. For each node v, we are given the node that produces the input for v. However, we do not know the nodes that need v's output as input. We can sort vertices by their input nodes, where vertices with fan-in zero are assumed to have input nodes with numbers $-\infty$. Note that the circuit is still topologically sorted: If u produces an output for v and v produces an output for w, then $u < v < w$. Thus, v will be stored before w because $u < v$. Each node v puts its output into the priority queue with priority v. Let w be a node that takes v's output as input, u be the node evaluated immediately before w. If u did not take v's output as input, then w is the first node that needs the output of v as input. By the order that the nodes are stored in, all nodes that need an input with smaller priority than v have already been evaluated. Thus, we can keep deleting minima from the priority queue until we delete an entry with priority v. This is the input for w. In addition to using this entry as input for w, we store the number v and the input in internal memory. Thus, if u took v's output as input, then v and its output are stored in internal memory. Therefore, when evaluating w, we can use this information to retrieve w's input. This does not require any operation on the priority queue.

Using this strategy, each vertex puts one output into the queue and each entry is deleted exactly once from the queue. Hence, the number of operations on the queue is again $O(N)$.

We have shown that Corollary 4.1 holds for these two slightly relaxed cases as well.

4.2 3-Coloring a Tree in External Memory

In this section, we show how to compute a 3-coloring of a rooted tree T in external memory. To do this, we generalize the approach used by Chiang *et al.* [12] for 3-coloring a list. In general, 3-coloring a tree might not be the most useful operation; but it is a basic step in the tree labeling algorithms presented in this chapter.

According to our convention, the edges of the tree are directed from children to parents. However, it is easy to 'turn the edges around'. This does not even have to be done physically. Instead, we can handle edge (v, w) in our algorithm as if it was edge (w, v). Thus, assume that our tree is represented as a set of edges directed from parents to children.

We call an edge (v, w) a *forward* (resp. *backward*) *edge*, if $v < w$ (resp. $v > w$), where "$<$" is a total order on the vertex names. A *forward* (resp. *backward*) *tree* is a maximal subtree of T containing only forward (resp. backward) edges. Vertices with only backward (resp. forward) edges incident on them are considered forward (resp. backward) trees consisting of only one vertex. (See Figure 4.4(a–c).)

Note that we have stored $p(v)$ with each vertex v in a forward tree. Thus, we can sort the vertices in the whole forest of forward tress by increasing parent numbers $p_f(v)$, where

$$p_f(v) = \begin{cases} p(v) & \text{if } p(v) < v \text{ and } p(v) \neq 0 \\ -\infty & \text{otherwise} \end{cases}.$$

By the maximality of forward trees, different forward trees are independent of each other. Moreover, $p_f(v) < v$, for each vertex in a forward tree. Thus, the resulting forest of forward trees is topologically sorted.

Analogously, we can topologically sort the forest of backward trees by sorting its vertices by decreasing parent numbers $p_b(v)$, where

$$p_b(v) = \begin{cases} p(v) & \text{if } p(v) > v \\ +\infty & \text{otherwise} \end{cases}.$$

Then, we use time-forward processing to 2-color forward and backward trees from the roots to the leaves as follows:

We color forward trees using colors 1 and 2. If a vertex is the root of a forward tree, it is colored with color 1. Otherwise, it is colored with the color that its parent does not have. This gives a 2-coloring because the only adjacencies of vertices in the tree are between children and parents and we make sure that a child always has a color different from its parent's. Denote the color assigned to a vertex v in this step as v's *forward color* $c_f(v)$. (See Figure 4.4(b).)

Backward trees are colored using colors 3 and 4. We color the root of a backward tree with color 3 and proceed in the same way as for forward trees afterwards. Denote the color assigned to a vertex in this step as v's *backward color* $c_b(v)$. (See Figure 4.4(c).)

Each vertex is contained in a forward and a backward tree. Thus, it is assigned a pair of colors $(c_f, c_b) \in \{1, 2\} \times \{3, 4\}$. We call two such color pairs *adjacent*, if it is possible that T contains two adjacent vertices with these two color pairs assigned to them. We define an adjacency graph for color pairs. This graph contains a vertex for each color pair and an edge between adjacent color pairs. We will show that this adjacency graph is 3-colorable. We make the following observations:

1. The target vertex of a forward edge must be the root of a backward tree. Thus, it must have backward color 3.

2. Analogously, the target vertex of a backward edge must have forward color 1.

3. If the forward color of the source vertex of a forward edge is 1, the forward color of the target vertex of this edge must be 2.

4. If the backward color of the source vertex of a backward edge is 3, the backward color of the target vertex of this edge must be 4.

Figure 4.2 shows the possible color pairs assigned to vertices connected by forward or backward edges. This results in the adjacency graph shown in Figure 4.3. The labels beside the vertices represent a possible 3-coloring c of the graph.

This 3-coloring of the adjacency graph gives us a 3-coloring of the given tree T. Let (c_f, c_b) be the color pair assigned to a vertex v of T. Then we assign color $c((c_f, c_b))$

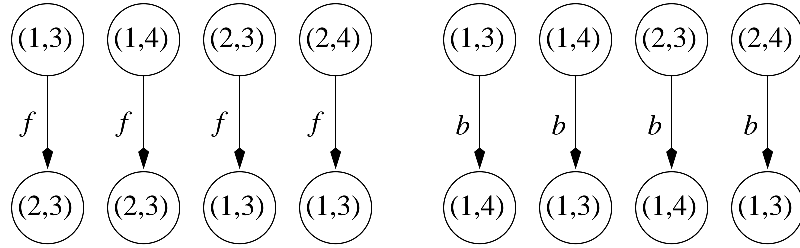

Figure 4.2: All possible color combinations assigned to vertices connected by forward (f) and backward (b) edges.

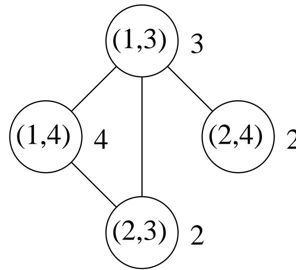

Figure 4.3: The adjacency graph derived from adjacencies in Figure 4.2. Each vertex is labeled inside with the color pair it represents and outside with its color in a possible 3-coloring.

to v. As adjacent color pairs have different colors in the 3-coloring of the adjacency graph, this is indeed a 3-coloring of T.

Looking a little closer at the 3-coloring in Figure 4.3, we can observe that each vertex with colors $(2, x)$ is finally colored 2. Colors $(1, x)$ map to x. Thus, when 2-coloring backward trees, we simply overwrite the colors of vertices v with $c_f(v) = 1$, i.e., $c(v) = c_b(v)$. For vertices v with $c_f(v) = 2$, we keep the color, i.e., $c(v) = c_f(v)$. (See Figures 4.4(b–d).)

Putting pieces together, we obtain Algorithm 4.1 for 3-coloring a rooted tree. We have shown the following lemma.

Lemma 4.1 *Given a rooted tree T of size N, we can compute a 3-coloring of T with $O(sort(N))$ I/Os.*

Proof. The correctness of Algorithm 4.1 follows from the above discussion.

As for the I/O-complexity, Steps 1 and 3 clearly take $O(sort(N))$ I/Os. Steps

3-COLOR(T, c):

Input: A rooted tree T.

Output: A 3-coloring c of T.

1: Sort the vertices of T by increasing parent numbers $p_f(v)$.
2: Use time-forward processing to 2-color forward trees with colors 1 and 2.
3: Sort the vertices of T by decreasing parent numbers $p_b(v)$.
4: Use time-forward processing to 2-color backward trees with colors 3 and 4, overwriting the colors of color-1 vertices and keeping the colors of color-2 vertices.

Algorithm 4.1: 3-coloring a rooted tree T.

2 and 4 each take $O(sort(N))$ I/Os by Corollary 4.1. Thus, Algorithm 4.1 takes $O(sort(N))$ I/Os. □

4.3 Computing Vertex-to-Root Distances

In this section, we show how to compute, for each vertex of T, its distance to the root r of T (see Figure 4.5(b)). Our algorithm is a generalization of the list ranking algorithm by Chiang *et al.* [12]. Algorithm 4.2 shows the pseudo-code. We prove the following lemma.

LEMMA 4.2 *Given a rooted tree T of size N, we can compute, for all vertices of T, their distances to the root of T with $O(sort(N))$ I/Os.*

PROOF. First, we have to provide some details of Steps 2, 6, and, 8 of procedure RECCOMPLEVELS.

Step 2 computes $l(v)$ for each vertex $v \in T$ as follows: Let $p'_v = \langle v = v_1, v_2, \ldots, v_k, v_{k+1} = r\rangle$ be the path from v to the root r of T, i.e., $v_{i+1} = p(v_i)$ for $1 \leq i \leq k$. Then we compute

$$l(v) = \sum_{i=1}^{k} w(v_i).$$

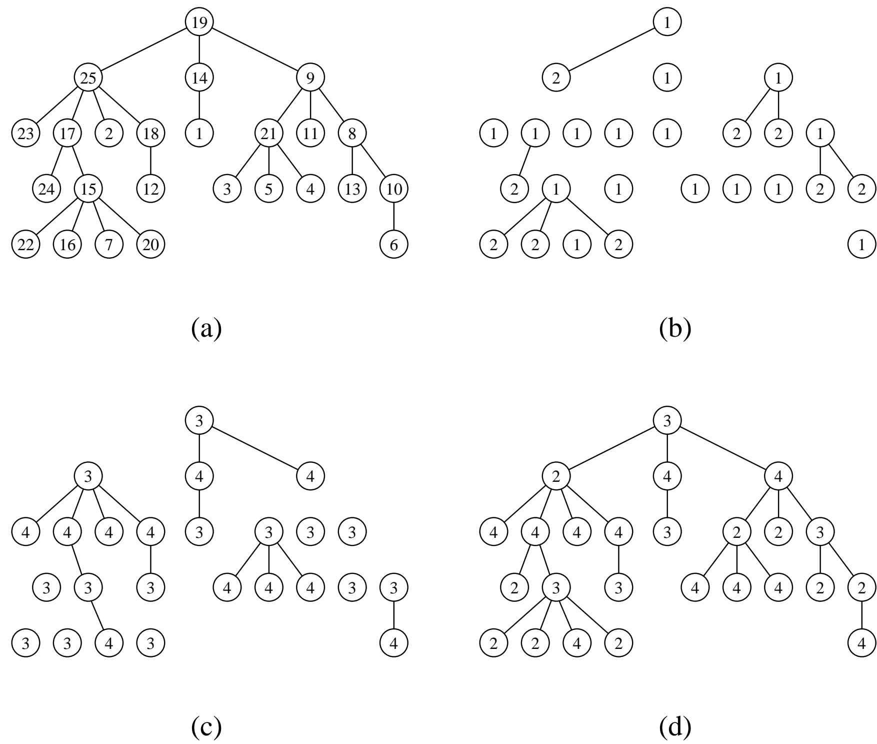

Figure 4.4: Figure (a) shows an example tree T labeled with vertex numbers. Figure (b) shows the corresponding forest of forward trees labeled with colors 1 and 2 according to the 2-coloring algorithm. Figure (c) shows the forest of backward trees labeled with colors 3 and 4 according to the 2-coloring algorithm. Figure (d) shows the original tree T labeled with colors $2, 3$, and 4. This coloring is derived from the two 2-colorings in figures (b) and (c).

COMPUTELEVELS(T, l):

Input: A rooted tree T with root r.

Output: For each vertex v of T its distance $l(v)$ to the root of T.

1: Assign weights $w(v) = 1$ to the vertices of T. For the root r, $w(r) = 0$.
{These weights denote the length of the path between v and $p(v)$ in T.}

2: $l \leftarrow$ RECCOMPLEVELS(T, w)

RECCOMPLEVELS(T, w):

1: **if** $|T| \leq M$ **then**

2: Load T into internal memory, compute the levels of its vertices in internal memory, and write l to disk afterwards.

3: **else**

4: 3-COLOR(T, c)

5: Choose $I = \{v \in T : c(v) = c'\}$ as an independent vertex set of T.
{Let c_1, c_2, c_3 be the three colors of the 3-coloring. $n(c_i)$ is the number of vertices with color c_i in T. Then we choose $c' \in \{c_1, c_2, c_3\}$ such that $n(c') = \max\{n(c_1), n(c_2), n(c_3)\}$.}

6: Bridge out the independent set. This results in a modified tree T' and modified weights w'.

7: RECCOMPLEVELS(T', w', l)

8: Reintegrate bridged-out elements and compute their levels.

9: **end if**

Algorithm 4.2: Computing for each vertex its distance to the root.

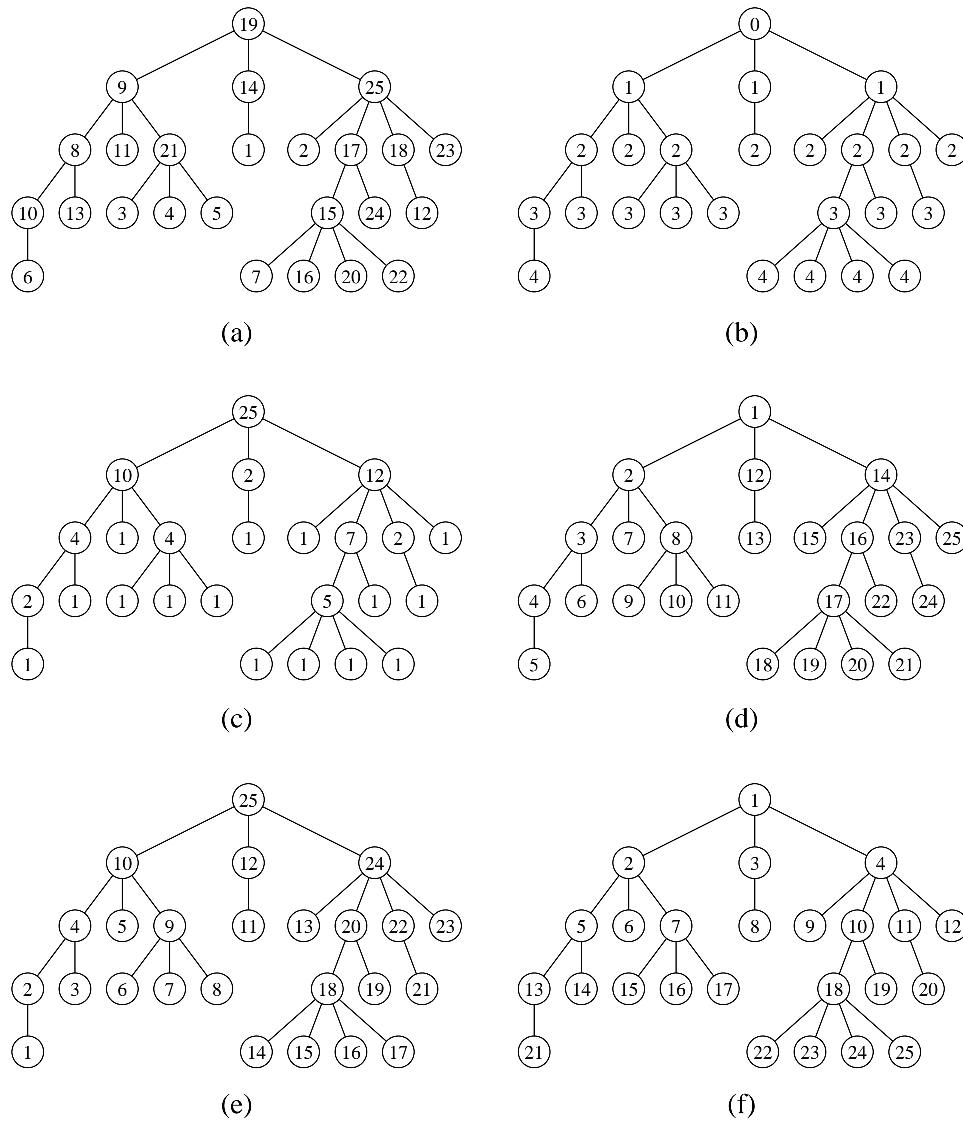

Figure 4.5: Figure (a) shows the example tree T of Figure 4.4(a) rearranged such that the children of each vertex are sorted by increasing vertex numbers from left to right because this is the order of the children used when computing preorder, postorder, and levelorder numbers. Figures (b) through (e) show the same tree with vertices labeled with their distances to the root of T, subtree sizes, preorder, postorder, and levelorder numbers, respectively.

If we are given the original tree T, then $w(v) = 1$ for each $v \in T$. Thus, $l(v)$ is indeed the distance from v to r (number of edges between v and r). So assume that we have done some recursive calls to RecCompLevels, and the given tree is not the original tree. In Step 6, we will make sure that p'_v is a subpath of the path p_v from v to r in the original tree. The weight of each vertex $v_i \in p'_v$ will be one more than the number of vertices in p_v between v_i and v_{i+1}. Thus, the sum of the weights on p'_v is the length of p_v in this case as well.

In Step 6, bridging out means that, for each vertex $v \in I$, we make the parent $p(v)$ of v the parent of each child w of v. In addition, we set $w'(w) = w(w) + w(v)$. Since I is an independent set, $p(v)$ and all children w of v cannot be in I. Thus, all children w of v are indeed in T' and $p(v)$ is their parent in T'.

Now, $w(v)$ was one more than the number of previously bridged-out vertices between v and $p(v)$. Analogously, $w(w)$ was one more than the number of previously bridged-out vertices between w and v. Now we bridge out one additional vertex between w and $p(v)$, namely v. Thus, $w'(w) = w(v) + w(w)$ is one more than the bridged-out elements between w and $p(v)$.

In Step 8, $p(v) \notin I$ for each $v \in I$. Thus, the recursive call to RecCompLevels returns the proper value $l(p(v))$. That is, we can compute $l(v) = l(p(v)) + w(v)$.

Now let us analyze the I/O-complexity of ComputeLevels. Step 1 of ComputeLevels requires a single scan over the vertices of T. Thus, ComputeLevels takes $O(scan(N))$ I/Os plus the I/Os performed by RecCompLevels.

In procedure RecCompLevels, we execute only Step 2, if $|T| \leq M$. This requires two scan operations: one to read T into internal memory and one to write the distances to external memory. Thus, Step 2 takes $O(scan(N))$ I/Os.

If $|T| > M$, we execute Steps 4 through 8. Step 4 takes $O(sort(N))$ I/Os by Lemma 4.1. Step 5 does not take any computation or I/Os at all because we can count the number of vertices colored with each color while executing Step 4. In Step 6 we decide whether $v \in I$ by looking at its color.

Step 6 can be implemented with $O(sort(N))$ I/Os as follows: Let c' be the color of the vertices in the independent set. Sort the vertices of T by the "$\sqsubset$" relation which is defined as $v \sqsubset w$ iff

1. $c(v) \neq c'$ and $c(w) = c'$,

2. $c(v) = c(w) \neq c'$ and $p(v) < p(w)$, or

3. $c(v) = c(w) = c'$ and $v < w$.

This creates two lists of vertices. The first one contains all vertices of T' sorted by their parents in T. The other one contains all vertices in I sorted by vertex numbers. We scan these two lists simultaneously in order to compute $w'(v) = w(v) + w(p(v))$ for each vertex v in T' with $p(v) \in I$. The details of these two scans are as follows:

We start both scans at the heads of the lists. Let v be the current vertex in $V(T')$ and w be the current vertex in I. There are three cases.

1. If $p(v) = w$, we set $w'(v) = w(v) + w(w)$ and $p'(v) = p(w)$ and advance to the next vertex in $V(T')$.

2. If $p(v) < w$, v's parent cannot be in I. If it were, we would have reached Case 1 and advanced to the next vertex in $V(T')$ before. Thus, we set $w'(v) = w(v)$ and $p'(v) = p(v)$ and advance to the next vertex in $V(T')$.

3. If $p(v) > w$, there is no vertex v' after v in $V(T')$ with $p(v') \leq w$ because $V(T')$ is sorted by increasing parent names. Therefore, we advance to the next vertex in I.

In Step 7, we apply RecCompLevels recursively to a tree T'. The size of T' is $|T'| \leq \frac{2}{3}|T|$ because $|I| \geq \frac{1}{3}|T|$ (we have chosen the most popular color c').

In Step 8, we apply the same technique as in Step 6 to compute $l(v) = w(v) + l(p(v))$, for each vertex v in I. This time, we sort the vertices of T' by increasing vertex numbers and the vertices of I by increasing parent numbers. The correctness of this technique and the $O(sort(N))$ I/O-bound follow analogously to those for Step 6.

We have shown that, except for the recursive call in Step 7, procedure RecCompLevels takes $O(sort(N))$ I/Os. We obtain the following recurrence for the

I/O-complexity $\mathcal{I}(N)$ of RecCompLevel:

$$\mathcal{I}(N) \leq \begin{cases} O(scan(N)) & \text{if } N \leq M \\ O(sort(N)) + \mathcal{I}(\frac{2}{3}N) & \text{if } N > M. \end{cases}$$

Assume that $\mathcal{I}(N') \leq c \cdot sort(N')$ for all $N' < N$. For $N' \leq M$ and an appropriately chosen constant c, this is true. Then,

$$\begin{aligned} \mathcal{I}(N) &\leq c' \cdot sort(N) + c \cdot sort\left(\frac{2}{3}N\right) \\ &\leq c' \cdot sort(N) + \frac{2}{3}c \cdot sort(N) \\ &\leq c \cdot sort(N) \qquad (\text{for } c \geq 3c'). \end{aligned}$$

That is, $\mathcal{I}(N) = O(sort(N))$.

This basically concludes the proof. However, there is a small technical detail that we have to deal with. Storing the vertices in T' and I consecutively as we do, we must have a way to recognize the last vertex of T'. This can only be done by storing the size of T' explicitly, for each recursive call. That is, we have to maintain all these tree sizes on a stack. There are $O(\log N)$ recursive calls and thus $O(\log N)$ subtree sizes to be maintained on this stack. Using an optimal external-memory stack, this takes $O(\frac{\log N}{DB}) = o(sort(N))$ I/Os in total. Thus, the claimed I/O bound still holds.

As ComputeLevels takes only $O(scan(N))$ I/Os in addition to those performed by RecCompLevels, the I/O-complexity of ComputeLevels is $O(sort(N))$. □

Corollary 4.2 *Given a forest F of rooted trees of total size N, we can compute, for each vertex v of F, its distance to the root of the tree in F containing v with $O(sort(N))$ I/Os.*

Proof. This follows immediately from Lemma 4.2 because none of the techniques relies on the fact that the tree is connected. The only important property is that each vertex has only one parent. This is true in a forest as well. □

ComputeSubtreeWeights(T, S):

Input: A rooted tree T.

Output: For each vertex $v \in T$, the weight $W(v)$ of $T(v)$.

1: ComputeLevels(T, l)

2: Use time-forward processing to compute, for each vertex v in T,

$$W(v) \leftarrow w(v) + \sum_{u \in C(v)} W(u),$$

where $C(v) = \{w \in T : p(w) = v\}$.

Algorithm 4.3: Computing subtree weights.

4.4 Computing Subtree Sizes or Weights

Given a rooted tree T, we define subtrees $T(v)$ for all vertices of v. $T(v)$ consists of all descendants of v, inclusive. We want to compute $|T(v)|$ for all vertices of T.

Let a weight function $w : T \to \mathbb{R}$ be given. We define subtree weights

$$W(v) = \sum_{u \in T(v)} w(u).$$

Computing $|T(v)|$ reduces to computing $W(v)$, where $w(v) = 1$ for all vertices v in T.

Lemma 4.3 *Given a rooted tree T of size N and a weight function $w : T \to \mathbb{R}^+$, we can compute, for all vertices v of T, the weights $W(v)$ of the subtrees $T(v)$ of T with $O(sort(N))$ I/Os.*

Corollary 4.3 *Given a rooted tree T of size N, we can compute, for all vertices v of T, the sizes of the subtrees $T(v)$ of T with $O(sort(N))$ I/Os.*

Proof. Let us analyze the number of I/Os performed by Algorithm 4.3. Step 1 takes $O(sort(N))$ I/Os by Lemma 4.2. Step 2 takes $O(sort(N))$ I/Os by Corollary 4.1. Thus, Algorithm 4.3 takes $O(sort(N))$ I/Os.

Now let us prove the correctness of Algorithm 4.3. $T(v)$ consists of v itself and the maximal subtrees rooted at the children of v. Therefore, $W(v)$ is the sum of the weights of the subtrees $T(w)$ rooted at v's children w plus $w(v)$. That is, our algorithm is correct. □

4.5 Computing Preorder, Postorder, and Levelorder Numbers

Preorder, postorder, and levelorder (or BFS) numbers are probably the most popular labels to define an order to visit the the vertices of a given rooted tree T. We will show how to compute these three particular numberings with $O(sort(N))$ I/Os.

4.5.1 Preorder and Postorder Numbers

Preorder (resp. postorder) numbers of the vertices of a given rooted tree T of size N can be defined as follows. (Usually, one uses an Euler tour to define these numbers; but our definition provides the idea how to compute these numbers for free):

Assign numbers 1 through N (sometimes also 0 through $N-1$) to the vertices of T such that

1. No two vertices have the same number,
2. The numbers in each maximal subtree $T(v)$ are contiguous, and
3. The root v of each maximal subtree $T(v)$ has the smallest (resp. greatest) number in $T(v)$.

See Figures 4.5(d) and 4.5(e) for an example. We denote the preorder (resp. postorder) number of v by $n_p(v)$ (resp. $n_P(v)$). Algorithms 4.4 and 4.5 compute these numbers for all vertices of T. They use labels $q(v)$ to define an order on the children of each vertex. By default, we choose $q(v) = v$. In Chapter 6, we will use these labels to ensure some special properties of the computed preorder numbers.

ComputePreorderNumbers(T):

Input: A rooted tree T.

Output: For each vertex v of T its preorder number $n_p(v)$.

1: $S \leftarrow$ ComputeSubtreeSizes(T)

2: Use time-forward processing from the root to the leaves to compute $n_p(v)$ for all v as follows:

```
if v = r then
  n_p(r) = 1
else if v is the leftmost child of p(v) then
  n_p(v) = n_p(p(v)) + 1
else
  w ← left sibling of v
  n_p(v) = n_p(w) + S(w)
end if
```

Algorithm 4.4: Computing preorder numbers.

ComputePostorderNumbers(T):

Input: A rooted tree T.

Output: For each vertex v of T its postorder number $n_P(v)$.

1: $S \leftarrow$ ComputeSubtreeSizes(T)

2: Use time-forward processing from the root to the leaves to compute $n_p(v)$ for all v as follows:

```
if v = r then
  n_P(r) = |T|
else if v is the rightmost child of p(v) then
  n_P(v) = n_P(p(v)) - 1
else
  w ← right sibling of v
  n_P(v) = n_P(w) - S(v)
end if
```

Algorithm 4.5: Computing postorder numbers.

LEMMA 4.4 *Given a rooted tree T of size N, we can compute, for all vertices v of T, their preorder (resp. postorder) numbers with $O(sort(N))$ I/Os.*

PROOF. Let us analyze Algorithm 4.4. The correctness and I/O-complexity of Algorithm 4.5 follows by symmetry.

First, we prove the correctness of Algorithm 4.4. We start by proving that each vertex v is the vertex with smallest number $n_p(v)$ in its subtree $T(v)$. For a leaf this is trivially true. Consider an internal vertex v with children $w_1, \ldots, w_k$ from left to right. By induction hypothesis, $w_1, \ldots, w_k$ have the smallest numbers in their respective subtrees $T(w_1), \ldots, T(w_k)$. Furthermore, $n_p(w_1) = n_p(v) + 1$, $n_p(w_i) = n_p(w_{i-1}) + S(w_{i-1})$, and $S(w_{i-1}) > 0$, for $1 < i \leq k$. Thus, $n_p(v) < n_p(w_i)$, for $1 \leq i \leq k$. As every vertex $u \neq v$ in $T(v)$ must be contained in one of the subtrees $T(w_i)$, we have $n_p(u) \geq n_p(w_i) > n_p(v)$. Hence, v has the smallest number in $T(v)$.

Next, we prove that the vertices in a subtree $T(v)$ have contiguous numbers and no number occurs more than once. Again, this claim is trivially true for the leaves of T. Consider an internal vertex v with children $w_1, \ldots, w_k$ again. By induction hypothesis, the nodes in $T(w_i)$ are numbered $n_p(w_i)$ through $n_p(w_i) + S(w_i) - 1$, for $1 \leq i \leq k$. For $1 \leq i < k$, $n_p(w_{i+1}) = n_p(w_i) + S(w_i)$. This implies that the vertices of $T(w_1), \ldots, T(w_k)$ have contiguous numbers and no number occurs more than once. The numbers that occur are $n_p(w_1)$ through $n_p(w_1) + \sum_{j=1}^{k} S(w_j) - 1$. v has been assigned number $n_p(v) = n_p(w_1) - 1$. Thus, all vertices of $T(v)$ are numbered contiguously and no number is used more than once. If we choose the root of T as the vertex v, this immediately implies that the numbers assigned to the vertices of T are 1 through N. The details for the time-forward processing in Step 2 are as follows:

First, we sort the vertices of T by the "$\lessdot$" relation defined as $v \lessdot w$ iff one of the following is true:

1. $l(v) < l(w)$,

2. $l(v) = l(w)$ and $p(v) < p(w)$, or

3. $l(v) = l(w)$, $p(v) = p(w)$, and $q(v) < q(w)$.

This ensures that the vertices are stored in top-down order. The children of any vertex are stored consecutively. The first vertex stored is the root r of T because it is the only vertex at the top level. We assign preorder number 1 to it and send this number to r's children. For the currently processed vertex, we store its parent, preorder number, and subtree size in internal memory. When processing the next vertex w, we test whether $p(v) = p(w)$. If so, we compute $n_p(w) = n_p(v) + S(v)$ because v is w's left sibling. Otherwise, w is the leftmost child of $p(w)$. Then we use $n_p(p(w))$ to compute $n_p(w) = n_p(p(w)) + 1$. $n_p(p(w))$ is retrieved from the priority queue.

Now let us analyze the I/O-complexity of this algorithm. By Corollary 4.3, Step 1 takes $O(\mathit{sort}(N))$ I/Os. Step 2 takes $O(\mathit{sort}(N))$ I/Os, by Corollary 4.1. □

The following observation follows immediately from the definition of the "$\lessdot$" relation.

Observation 4.1 *Let v and w be two children of a vertex u with $q(v) < q(w)$. Then v has a smaller preorder number than w.*

4.5.2 Levelorder Numbers

There are different definitions of levelorder numbers. The least strict definition just defines levelorder numbers as an assignment of number 1 through N to the vertices of T such that vertices at higher levels have smaller numbers than those on lower levels. This version can easily be computed by first computing the levels of all vertices of T, sorting the vertices by increasing levels, and numbering them in their order of appearance. This takes $O(\mathit{sort}(N))$ I/Os.

We are interested in computing a more restrictive version of levelorder numbers. Let v and w be two vertices with levelorder numbers $n_l(v) < n_l(w)$. Let v' be a child of v and w' be a child of w. Then we require that $n_l(v') < n_l(w')$. This basically means that we draw T in the plane and number the vertices level by level from left to right (see Figure 4.5(f)). This still leaves us some freedom about the embedding of the tree. However, once it is embedded, the numbers are well-defined. The next lemma states that we can compute levelorder numbers with $O(\mathit{sort}(N))$ I/Os.

ComputeLevelorderNumbers(T):

Input: A rooted tree T.

Output: For each vertex v of T, its levelorder number $n_l(v)$.

1: $n_p \leftarrow$ ComputePreorderNumbers(T)
{This computes labels $l(v)$ as part of the computation. Thus, we can use them in the next step.}
2: Sort the vertices of T by '$\prec$', where $v \prec w$, iff $l(v) < l(w)$ or $l(v) = l(w)$ and $n_p(v) < n_p(w)$.
3: Scan the vertex list of T and number the vertices of T in their order of appearance.

Algorithm 4.6: Computing levelorder numbers.

Lemma 4.5 *Given a rooted tree T of size N stored in $\lceil N/B \rceil$ blocks evenly distributed over D parallel disks, we can compute, for each vertex v of T, its levelorder number with $O(sort(N))$ I/Os.*

Proof. By Lemma 4.4, Step 1 takes $O(sort(N))$ I/Os. Step 2 takes $O(sort(N))$ I/Os as well. Step 3 just scans the vertex list of T and numbers the vertices. This takes $O(scan(N))$ I/Os. Thus, Algorithm 4.6 takes $O(sort(N))$ I/Os. Next, we prove the correctness of Algorithm 4.6.

In Step 2, we sort the vertices of T primarily by increasing distance to the root. Thus, all vertices at a level l in T appear before all vertices at another level $l' > l$. Thus, they will have smaller levelorder numbers than the vertices at level l'.

If v and w are vertices at the same level and $n_l(v) < n_l(w)$, then $v \prec w$. Thus, $n_p(v) < n_p(w)$ because $l(v) = l(w)$. All descendants v' of v have greater preorder number than v. The vertices in $T(v)$ are numbered consecutively. Thus, $n_p(v) < n_p(v') < n_p(w)$ for all vertices v' in $T(v)$. Analogously, $n_p(w) < n_p(w')$, for all descendants w' of w. Since $l(v) = l(w)$, $l(v') = l(w')$ for all children v' of v and w' of w. Moreover, $n_p(v') < n_p(w) < n_p(w')$. Thus, $v' \prec w'$, and therefore $n_l(v') < n_l(w')$. □

4.6 Lower Bounds

In this section, we show that, except for 3-coloring, all labels computed in this chapter require $\Omega(perm(N))$ I/Os to compute them.

We can consider a list as a special rooted tree. If we let the tail of the list be the root, then list ranking reduces to computing preorder numbers, levelorder numbers, or distances to the root without any additional computation or I/O. If we let the head of the list be the root, then list ranking reduces to computing postorder numbers or subtree sizes without any additional computation or I/O.

Chiang *et al.* [12] have shown a lower bound of $\Omega(perm(N))$ I/Os for list ranking. Thus, we can conclude the following lemma.

LEMMA 4.6 *Computing distances to the root, subtree sizes, preorder, postorder, and levelorder numbers takes* $\Omega(perm(N))$ *I/Os.*

Note that, for *all* practical cases, $sort(N) = perm(N)$. See [12] for an estimate of the minimum N such that $perm(N) < sort(N)$, for practical values of M and B. This calculation shows that, in practice, there can really be *no* problem instance with large enough N.

We summarize the results of this chapter in the following theorem.

THEOREM 4.2 *Given a rooted tree* T *of size* N*, it is possible to compute a 3-coloring of* T*, distances to the root, subtree sizes* $|T(v)|$*, preorder, postorder, and levelorder numbers for all vertices of* T *with* $O(sort(N))$ *I/Os. Except for 3-coloring, these labelings require* $\Omega(perm(N))$ *I/Os to compute them.*

Chapter 5

Blocking Rooted Trees: The Algorithm

In this chapter, we present an $O(sort(N))$ algorithm to block rooted trees in the way described in Chapter 3. In fact, it will not block the tree precisely as described in that chapter. It will use the same ideas, but will also be sensitive to the structure of the given tree. This way, we may achieve a better storage blow-up than predicted by the worst-case estimation of Theorem 3.1.

5.1 Framework of the Algorithm

Initially, we assume that the tree is given in the way as in our shortest path data structure. That is, we are given pairs $(v, p(v))$, where v is the name of a vertex in T and $p(v)$ is a pointer to the pair $(w, p(w))$ representing v's parent w. If v is the root of T, then $p(v) = \textbf{null}$. Since our algorithm keeps moving the vertices of T around, these pointers are a little inconvenient. Hence, the first step of our algorithm replaces the pointer $p(v)$ by the name of v's parent. For the root, we replace **null** by 0, assuming, w.l.o.g., that no vertex is named 0.

After arranging the vertices in the right way, however, we have to replace these vertex names by pointers again. Otherwise, we would not be able to traverse bottom-up paths quickly. We would have to search our data structure for a copy of the parent

TREEBLOCKING(T):

Input: A tree T represented as a list of pairs $(v, p(v))$ striped across D disks.
Output: The tree T blocked for fast path traversals.

1: MAKEEDGELIST(T)
{Replace parent pointers by parent names.}
2: $l \leftarrow$ COMPUTELEVELS(T)
{Compute the distance to the root, for each vertex of T.}
3: $n_p \leftarrow$ COMPUTEPREORDERNUMBERS(T, l)
4: $s \leftarrow$ COMPUTESUBTREELABELS(T, l, n_p)
{Here, subtrees are the trees that result from cutting T into layers. We assign unique labels to all subtrees.}
5: Sort vertices by increasing labels $s(v)$ and $n_p(v)$.
6: CONSTRUCTSUBTREES(T, l, n_p, s)
{Construct the subtrees $T_{i,j}$ of all subtrees T_i. This concludes the actual blocking algorithm.}
7: MAKEPARENTPOINTERS(T)
{Replace parent names by parent pointers again.}

Algorithm 5.1: The general framework of the blocking algorithm.

of the current vertex. This would require many I/Os if the parent does not happen to be in the same block. Even if the parent was in the same block, we would at least have to search the block. This would not require any additional I/Os, but would still require much unnecessary computation.

The replacement of parent pointers by vertex names basically leaves us with a set of edges directed from children to parents.

Algorithm 5.1 shows the pseudo-code of procedure TREEBLOCKING, which blocks the given tree. Observe that, instead of computing preorder numbers in each subtree separately, as we do in Section 3.4.2, we use preorder numbers computed in the whole tree T for the construction. The next observation states that this gives the same result.

OBSERVATION 5.1 *Let T_i be a subtree of T, and let the vertices of T_i be sorted by*

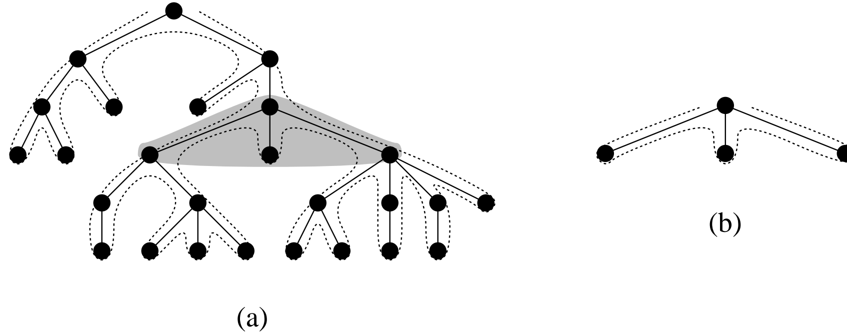

Figure 5.1: Figure (a) shows a rooted tree T with an Euler tour $\mathcal{E}$ (dotted curve). Figure (b) shows the shaded subtree T_i with an Euler tour $\mathcal{E}_i$. $\mathcal{E}_i$ is constructed from $\mathcal{E}$ as in the proof of Observation 5.1.

their preorder numbers $n_p(v)$ in T. Then there exists a preorder numbering $n'_p(v)$ of T_i such that, sorting the vertices of T_i by $n'_p(v)$, they are stored in the same order as when sorted by $n_p(v)$.

PROOF. Let r_i be the root of T_i. Consider the Euler tour $\mathcal{E}$ of T defining the numbers $n_p(v)$. We construct an Euler tour $\mathcal{E}_i$ of T_i from $\mathcal{E}$ as follows: Remove all vertices not in T_i from $\mathcal{E}$. Merge resulting adjacent copies of the same vertex into a single copy. Intuitively, this means that we cut $\mathcal{E}$ at the root r_i of T_i and at the leaves of T_i. At r_i we leave $\mathcal{E}_i$ open, so that r_i is the new start and endpoint of the tour, At each leaf l of T_i with parent $p(l)$, we make edge $(l, p(l))$ the successor of edge $(p(l), l)$ on the tour (see Figure 5.1). We use $\mathcal{E}_i$ to define the preorder numbers $n'_p(v)$.

It remains to show that, if $n_p(v) < n_p(w)$ for two vertices $v, w \in T_i$, then $n'_p(v) < n'_p(w)$. If $n_p(v) < n_p(w)$, v had its first appearance before w's first appearance on $\mathcal{E}$. Since v and w are both vertices of T_i, they are both in $\mathcal{E}_i$. v has its first appearance before w's first appearance in $\mathcal{E}_i$ as well. Hence, $n'_p(v) < n'_p(w)$. □

In Step 5 of Algorithm 5.1, we sort the vertices primarily by subtree labels and only secondarily by preorder numbers. Thus, a change of preorder numbers could only affect the order of the vertices in the same subtree. By Observation 5.1, the order of the vertices in each subtree remains unchanged.

The next theorem states the main result of this chapter.

THEOREM 5.1 *Given a rooted tree T, we can store T in at most $5D\left\lceil\frac{|T|}{DB}\right\rceil$ blocks evenly distributed over D parallel disks so that traversing any bottom-up path of length k in T takes at most $\lceil 3k/DB\rceil + 1$ I/Os. Blocking T this way takes $O(sort(N))$ I/Os.*

PROOF. The speed-up and storage blow-up follow from Theorem 3.1.

For the construction, we use Algorithm 5.1. The correctness of Algorithm 5.1 follows from Lemmas 4.2, 4.4, 5.1, 5.2, 5.3, and 5.4.

By the same lemmas, all steps, except Step 6, take $O(sort(N))$ I/Os. Step 6 takes $O(scan(N))$ I/Os. Thus, Algorithm 5.1 takes $O(sort(N))$ I/Os. □

The next sections provide the details of Steps 1, 4, 6, and 7. In all stated I/O-bounds stated, N denotes the number of vertices in T. This is important, as the blocked version of T may contain multiple copies of many vertices of T.

5.2 MAKEEDGELIST

Procedure MAKEEDGELIST replaces parent pointers in T by vertex numbers. As mentioned before, this basically transforms T into a list of edges directed from children to parents. The only exception is the entry $(r, 0)$ produced by this procedure, where r is the root of T. We will keep this entry nevertheless because it makes subsequent steps easier.

Algorithm 5.2 shows the code of procedure MAKEEDGELIST. The following lemma states its correctness and bounds the number of I/Os that it performs.

LEMMA 5.1 *Algorithm 5.2 produces an edge-list representation of T with $O(sort(N))$ I/Os.*

PROOF. Step 1 and 3 take $O(scan(N))$ I/Os. Step 2 takes $O(sort(N))$ I/Os. We assume that deallocating disk space is possible with a constant number of I/Os. Then Step 4 takes $O(1)$ I/Os. Thus, the I/O-complexity of Algorithm 5.2 is $O(sort(N))$.

Let us prove its correctness. If $adr(v) = p(v')$ in Step 3, we set $p(v') = v$ because $p(v')$ points to v's address. After that we are done with vertex v'. Thus, we can proceed to the next vertex in T'. If $adv(v) < p(v')$, we go to the next vertex v in T.

MAKEEDGELIST(T):

Input: A rooted tree T represented as a list of vertices with pointers to their parents.
Output: T, represented as a list of edges directed from children to parents.

1: Make a copy T' of T.
2: Sort the vertices of T' by increasing parent addresses.
3: Scan T and T' simultaneously to determine the parents of the vertices in T':
{v is the current vertex in T. v' is the current vertex in T'. $p(v')$ is a pointer to a vertex address. $adr(v)$ is v's address.}
  if $adr(v) < p(v')$ **then**
    Advance to the next vertex in T.
  else
    {If $adr(v) = p(v')$.}
    $p(v') \leftarrow v$. Advance to the next vertex in T'.
  end if
4: Discard T and return T' as the modified version of T.

Algorithm 5.2: Transforming parent pointers into parent numbers.

As $p(v')$ is a proper address in T, we will find it using our strategy. Then we will have the case $adr(v) = p(v')$ again. The case $adv(v) > p(v')$ cannot occur. Assume that we have this case. As $p(v')$ points to a proper address in T, there exists a vertex v'' that is stored at this address. This vertex must have been the current vertex at some point during the scan of T. Then we must have advanced from v'' to the next vertex in T because $adr(v'') < adr(v)$. This happens only if $adr(v'') < p(v^*)$, for some vertex v^* stored before v' in T'. That is, $p(v^*) > adr(v'') = p(v')$, in contradiction to the order by which T' is sorted. Thus, Algorithm 5.2 is correct. □

5.3 MAKEPARENTPOINTERS

Procedure MAKEPARENTPOINTERS replaces, for each copy of a vertex v of T, the name of the parent $w = p(v)$ of v by a pointer to a copy of w. We do this locally in each block. That is, as long as there is a copy of $p(v)$ in the same block as the currently considered copy of v, we set the pointer to this copy. Procedure CONSTRUCTSUBTREES ensures that each subtree $T_{i,j}$ of T is stored in a single block. Thus, by giving

preference to local copies of $p(v)$, we guarantee that we need only one I/O to traverse a bottom-up path in any $T_{i,j}$. Algorithm 5.3 shows the pseudo-code of procedure MAKEPARENTPOINTERS.

LEMMA 5.2 *Algorithm 5.3 replaces parent names by pointers to the parents with $O(sort(N))$ I/Os.*

PROOF. Let us first prove the correctness of Algorithm 5.3. The replacement of local parent in Step 1 can be done in internal memory. We simply assume that we use a correct internal-memory algorithm to do this. For all non-local parents, we create a list L of "queries". A query is a parent name $p(v)$ that we want to replace plus an address where we want to replace it. Step 5 answers these queries. Its correctness follows for the same reasons as the correctness of Step 2 in Algorithm 5.2. Step 7 writes the answers to the queries in L to the addresses where these queries came from. Again, the arguments for the correctness of Step 7 are similar to those for the correctness of Step 2 in Algorithm 5.2. Thus, Algorithm 5.3 is correct.

Let us analyze its I/O-complexity. $|L| \leq |T| = |T'|$. By Theorem 3.1, $|T| = O(N)$. Thus, Steps 1, 2, 5, and 7 take $O(scan(N))$ I/Os. Steps 3, 4, and 6 take $O(sort(N))$ I/Os. Step 8 takes $O(1)$ I/Os. Thus, Algorithm 5.3 takes $O(sort(N))$ I/Os. □

5.4 COMPUTESUBTREELABELS

Procedure COMPUTESUBTREELABELS labels each vertex v of T with a number that uniquely identifies the subtree T_i containing v. In particular, we label all vertices in T_i with the preorder number of the root of T_i. Since each preorder number appears exactly once in T, this label is indeed unique for T_i. Algorithm 5.4 shows the pseudo-code of procedure COMPUTESUBTREELABELS.

LEMMA 5.3 *Algorithm 5.4 labels each vertex v of T with the preorder number of the root of the subtree T_i of T containing v. This takes $O(sort(N))$ I/Os.*

PROOF. The only information that we need to determine the label $s(v)$ of a vertex is the label $s(p(v))$ and its own level number $l(v)$. A vertex at level $l(v) = k(DB/3)$ for

MAKEPARENTPOINTERS(T):

Input: A blocked rooted tree T containing possibly several copies of the same vertex.

Output: T with parent names replaced by pointers to the parents. These pointers are local to their blocks as far as possible.

1: Scan T and replace parent names by pointers to local copies of the parent:
{v is the current vertex.}
   if there is a copy of $p(v)$ in v's block **then**
      Make $p(v)$ point to this copy.
   else
      Write the pair $(adr(v), p(v))$ to a list L.
   end if

2: Scan T and make a copy T' of T including the address in T of each copied vertex.

3: Sort T' by vertex names.

4: Sort L by parent names $p(v)$.

5: Scan T' and L simultaneously to replace, in L, parent names $p(v)$ by the address of $p(v)$ in T:
{v is the current vertex in L. v' is the current vertex in T'.}
   if $v' < p(v)$ **then**
      Advance to the next vertex in T'.
   else
      {If $v' = p(v)$.}
      $p(v) \leftarrow adr(v')$ (address in T, as stored with v'!). Advance to the next vertex in L.
   end if

6: Sort L by addresses $adr(v)$.

7: Scan T and L simultaneously to write pointers $p(v)$ in L to addresses $adr(v)$ (as stored with $p(v)$ in L) in T:
{v is the current vertex in T. v' is the current vertex in L.}
   if $adr(v') < adr(v)$ **then**
      Advance to the next vertex in L.
   else
      {If $adr(v') = adr(v)$.}
      $p(v) \leftarrow p(v')$. Advance to the next vertex in T.
   end if

8: Discard T' and L.

Algorithm 5.3: Replacing parent names by pointers to the parents.

COMPUTESUBTREELABELS(T, l, n_p):

Input: A rooted tree T represented by edges directed from children to parent. The vertices of T are labeled with their preorder numbers $n_p(v)$ and their levels $l(v)$ in T.

Output: A list s containing the subtree labels for all vertices of T.

1: Sort the vertices of T by increasing labels $l(v)$ and increasing preorder numbers $n_p(v)$.

2: Compute labels $s(v)$ using time-forward processing from the root of T to the leaves:
{v is the current vertex.}
if $l(v) = \frac{kDB}{3}$ for some $k \in \mathbb{N}$ **then**
$s(v) \leftarrow n_p(v)$
else
$s(v) \leftarrow n_p(p(v))$
end if

Algorithm 5.4: Labeling all vertices of each subtree T_i of T with a number uniquely identifying T_i.

some $k \in \mathbb{N}$ is the root of a subtree T_i. Thus, we choose v's preorder number, $n_p(v)$, as its label $s(v)$. Every vertex of T at another level is a non-root vertex of a subtree T_i of T. Thus, it has the same label as its parent.

Step 1 takes $O(sort(N))$ I/Os. Step 2 takes $O(sort(N))$ I/Os, by Corollary 4.1. Thus, Algorithm 5.4 takes $O(sort(N))$ I/Os. □

5.5 CONSTRUCTSUBTREES

In this section, we show how to implement the most important part of procedure TREEBLOCKING: the construction of the subtrees $T_{i,j}$. We claim that we can do this with $O(scan(N))$ I/Os after performing Steps 1–5 of Algorithm 5.1. Algorithms 5.5, 5.6, and 5.7 show the code.

We have divided the task of blocking T into three major subtasks, each of which is solved by a separate procedure. CONSTRUCTSUBTREES constructs the subtrees $T_{i,j}$ as in Lemma 3.5, more or less. It collects the vertices of the current subtree in the

ConstructSubtrees(T, l, n_p, s):

Input: A rooted tree T, the levels l, preorder numbers n_p, and subtree labels s of all vertices of T.

Output: T, blocked for fast bottom-up traversals.

1: Create empty buffers $\mathcal{B}$ and $\mathcal{B}_s$ of size DB.
{$\mathcal{B}$ is used for the construction of subtrees $T_{i,j}$. In $\mathcal{B}_s$, we collect small subtrees $T_{i,j}$.}

2: Scan T to create subtrees $T_{i,j}$:
{v is the current vertex. w is the previous vertex.}

```
if v is the first vertex in T or s(v) ≠ s(w) then
  {v is the first vertex of a new subtree T_i.}
  if B is not empty then
    WriteSubtree(B, B_s)
  end if
  Empty B and add v to B.
else
  if B is full then
    Write B to disk.
    KeepAncestors(v, B)
  end if
  Add v to B.
end if
```

3: Flush $\mathcal{B}$:

```
if B is not empty then
  WriteSubtree(B, B_s)
end if
```

4: Flush $\mathcal{B}_s$:

```
if B_s is not empty then
  Write B_s to disk.
end if
```

Algorithm 5.5: Constructing the subtrees of T.

KeepAncestors($v, \mathcal{B}$):

1: **for** all levels l in T_i with $l < l(v)$ **do**
2: Among all vertices w with $l(w) = l$, keep the one with maximal preorder number $n_p(w)$ such that $n_p(w) < n_p(v)$.
3: **end for**
4: {All vertices that we do not explicitly keep in $\mathcal{B}$ are discarded.}

Algorithm 5.6: Code of the KeepAncestors procedure.

```
WriteSubtree(B, B_s):
1: if |B| > DB/2 then
2:    Write B to disk
3: else
4:    Add the content of B to B_s
5:    if |B_s| > DB/2 then
6:       Write B_s to disk
7:       Empty B_s
8:    end if
9: end if
```

Algorithm 5.7: Code of the WriteSubtree procedure.

buffer $\mathcal{B}$ and relies on procedure KeepAncestors. The task of this procedure is to ensure that, each time we put the first vertex v of a new subtree $T_{i,j}$ into the buffer $\mathcal{B}$, we also put the ancestors of v in T_i into $\mathcal{B}$. In fact, these ancestors are already in the buffer and we just keep them there. The task of collecting the subtrees in blocks as described in Lemma 3.4 is carried out by WriteSubtree. Let us consider this procedure first.

The block-merging process in Lemma 3.4 consists of two parts. Initially, we assume that we put each subtree $T_{i,j}$ into its own (logical) block. The first part of the block-merging merges pairs of small logical blocks (of size at most $DB/2$) into one until there is at most one small logical block left. The second part merges the remaining small logical block with another large block, if this is possible. The second part reduces the size of the blocked tree T by at most one block. This is not worth the effort. Thus, we ignore the second part of the block-merging process.

The logical blocks containing the subtrees $T_{i,j}$ are given to WriteSubtree as the input buffers $\mathcal{B}$, one per call to WriteSubtree. Large logical blocks (of size greater than $DB/2$) are written to disk immediately. Small logical blocks are collected in the buffer $\mathcal{B}_s$ until the size of $\mathcal{B}_s$ exceeds $DB/2$. Then we write $\mathcal{B}_s$ to disk and empty it. This way, we ensure that, except for the very last block written in Step 4 of ConstructSubtrees, every block that we write to disk has size at least $DB/2$.

Assume for now that KeepAncestors correctly puts the ancestors of vertex v into the buffer $\mathcal{B}$. We have to show that ConstructSubtrees decomposes each

subtree T_i of T into subtrees as in Lemma 3.5 or better. Step 2 of Algorithm 5.5 solves this task.

If v starts a new subtree T_i, we flush $\mathcal{B}$ to disk and start constructing a new subtree $T_{i,0}$ with an empty buffer. Thus, we have room for DB vertices in $\mathcal{B}$. That is, if $|T_i| \leq DB$, we construct only a single subtree, as in the proof of Lemma 3.5.

If v does not start a new subtree T_i, we have to distinguish two cases. If $\mathcal{B}$ is not full, we add v to $\mathcal{B}$. If $\mathcal{B}$ is full, we have to flush $\mathcal{B}$ and start constructing a new subtree $T_{i,j}$. We start this new subtree with v and all its ancestors in T_i. Procedure KeepAncestors ensures this. As v can have at most $\frac{DB}{3}$ ancestors, inclusive, there is room for at least $\frac{2DB}{3}$ more vertices in $\mathcal{B}$. That is, we achieve a storage blow-up that is at least as good as claimed in Lemma 3.5.

It remains to show that KeepAncestors puts the ancestors of v in T_i into $\mathcal{B}$. Let us consider the subtree T' of T_i consisting of the previous subtree $T_{i,j-1}$ plus vertex v and all its ancestors. The last vertex w of $T_{i,j-1}$ and v have consecutive preorder numbers in T_i. Thus, by Observation 3.2, all proper ancestors of v are ancestors of w. That is, they are in $T_{i,j-1}$. Hence, we can construct the list of ancestors of v by keeping the right vertices of $T_{i,j-1}$ in $\mathcal{B}$. This is what KeepAncestors does. For each level above v, it chooses the vertex with greatest preorder number. To prove that this really chooses the ancestors of v, we make the following observations:

(1) All ancestors of v in T_i have to be in T_i and at levels above v, one per level.

(2) Assume that we have chosen a vertex p at a particular level and that another vertex q is v's ancestor at this level. p and q have smaller preorder numbers than v because they are in $T_{i,j-1}$ and v is not. $n_p(p) > n_p(q)$ because we have chosen p instead of q. That is, $n_p(q) < n_p(p) < n_p(v)$. Then q must be an ancestor of p by Observation 3.2. This is impossible because p and q are at the same level. That is, v's ancestors all have to be the vertices with greates preorder numbers in their levels. Thus, our selection criterion is correct, and Algorithm 5.5 is as well.

Let us analyze the I/O-complexity of ConstructSubtrees. We divide the analysis into input and output operations. The only input operations are performed when scanning T in Step 2 of Algorithm 5.5. Thus, we perform $O(scan(N))$ input operations. The only output operations are performed in procedure WriteSubtree

and in Step 4 of Algorithm 5.5. Step 4 takes exactly one I/O. WriteSubtree writes a block to disk, only if it is at least half full. By Theorem 3.1, the storage blow-up is at most 5. Thus, WriteSubtree performs $O(scan(N))$ I/Os, as well. Hence, Algorithm 5.5 takes $O(scan(N))$ I/Os.

Summarizing, we have shown the following lemma.

Lemma 5.4 *Procedure* ConstructSubtrees *blocks the given tree with* $O(scan(N))$ *I/Os.*

This concludes the proof of Theorem 5.1.

Chapter 6

Separating Embedded Planar Graphs in External Memory

In this chapter, we present an external-memory algorithm for separating embedded planar graphs. It is based on a planar separator algorithm due to Lipton and Tarjan [26]. We show that our algorithm takes $O(sort(N))$ I/Os, if we are given a BFS-tree of the given graph. If no BFS-tree is given, the best known algorithms to compute such a tree [25, 24] take $\Omega(N)$ I/Os in the worst case, where N is the number of vertices in the given graph.

It is very interesting that the existence of an I/O-efficient algorithm for breadth-first search is still open, while there exists a trivial optimal sequential algorithm that solves this problem in internal memory. A fair amount of research on breadth-first search, depth-first search, and related graph traversal strategies has been done for the PRAM model [31, 21, 1, 11, 23]; but all these algorithms do considerably more than linear work. This seems to point out that BFS is indeed hard in models other than a sequential random access machine.

As BFS is the bottle-neck of our algorithm, we took a look at other separator algorithms [22, 17, 3, 27, 20], hoping to find one that does not compute a BFS-tree as part of the computation. All of them do, and they rely on special properties of BFS-trees. We were not able to make any of these algorithms work with an arbitrary spanning tree. In the last chapter, we point out how we hope to avoid computing a

BFS-tree or to use additional information that we are given in certain applications to facilitate the computation of a BFS-tree. So far, however, we made only little progress in either of these two directions.

6.1 Data Representation

Before discussing our algorithm, we have to explain how the graph and its embedding is represented. As for the graph itself, we represent its vertex and edge sets as two lists $V(G)$ and $E(G)$. Every vertex in $V(G)$ is represented by a unique name. Every edge $\{v, w\} \in E(G)$ is represented as a pair (v, w).

The embedding is represented by the order of the edges around each vertex. That is, we augment each edge $e = (v, w)$ to a 4-tuple (v, w, n_v, n_w), where n_v (resp. n_w) is the position of e in the order of edges counterclockwise around v (resp. w). This order is circular; but we can easily make it linear by choosing an arbitrary edge to be the first one and then counting counterclockwise around the vertex. Call this first edge the *reference edge* of v.

6.2 Framework of the Algorithm

In this section, we present the framework of our separator algorithm. Assume that we are given an embedded planar graph with weights assigned to its vertices. The sum of these weights is at most 1. The basic idea is as follows:

Let $G_1, \dots, G_k$ be the connected components of G. At most one of them can have weight greater than $\frac{2}{3}$. Call this component G', if it exists. We compute a separator S of G' that divides G' into connected components $G'_1, \dots, G'_{k'}$, each of weight at most $\frac{2}{3}$. If no component G' with $w(G') \geq \frac{2}{3}$ exists, we set $S = \emptyset$. Either way, every connected component of $G - S$ has weight at most $\frac{2}{3}$. Now we can construct A and B by grouping the right connected components of $G - S$ together.

Algorithm 6.1 shows the pseudo-code of our separator algorithm. The following theorem states the main result of this chapter.

SEPARATE(G):

Input: A planar graph G as a vertex list $V(G)$ and an edge list $E(G)$.

Output: $V(G)$ and $E(G)$ such that

- all elements of the separator $S \subseteq V(G)$ are labeled with 0,
- all edges in $E(G) \setminus E(G - S)$ are labeled with ∞,
- all other vertices and edges are labeled with 1, if they are in A, and with 2, if they are in B, and
- $V(G)$ and $E(G)$ are sorted by increasing labels.

1: COMPUTECONNECTEDCOMPONENTS(G)
   {Label all vertices and edges of G with positive labels $C(x)$ identifying the connected components that they are part of.}
2: FINDHEAVIESTCOMPONENT(G, G')
   {Compute the weights of all connected components of G and determine the component G' with greatest weight.}
3: **if** G' has weight greater than $\frac{2}{3}$ **then**
4: SEPARATECONNECTED(G')
   {Compute a $\sqrt{N}$-separator S of G'. That is, label all vertices in S with 0 and all edges in $E(G') \setminus E(G' - S)$ with ∞.}
5: COMPUTECONNECTEDCOMPONENTS($G' - S$)
   {Label all vertices and edges of $G' - S$ with positive labels identifying the connected components that they are part of. These labels have to be different from those assigned in Step 1.}
6: $G' \leftarrow$ FINDHEAVIESTCOMPONENT(G)
   {Compute component weights again and determine the new heaviest component G' of $G - S$.}
7: **end if**
8: COLLECTCOMPONENTS(G, G')
   {Construct components A and B of weight at most $\frac{2}{3}$ each. Label all vertices and edges in A with 1. The vertices and edges in B are labeled with 2. Sort $V(G)$ and $E(G)$ by increasing labels.}

Algorithm 6.1: The separator algorithm.

THEOREM 6.1 *Given an embedded planar graph G with N vertices, we can compute a $\sqrt{N}$-separator of G with $O(sort(N) + bfs(N))$ I/Os, where $bfs(N)$ is the number of I/Os required to do breadth-first search in a planar graph with N vertices.*

PROOF. Lemmas 6.1, 6.3, 6.5, and 6.4 state that procedures COMPUTECONNECTEDCOMPONENTS, FINDHEAVIESTCOMPONENT, SEPARATECONNECTED, and COLLECTCOMPONENTS do what we claim in the comments of each step in Algorithm 6.1.

Step 4 of Algorithm 6.1 labels all separator vertices with $C(v) = 0$. The edges in $E(G') \backslash E(G' - S) = E(G) \backslash E(G - S)$ are labeled with $C(e) = \infty$. Step 8 labels the vertices and edges in A with $C(x) = 1$. Vertices and edges in B are labeled with $C(x) = 2$. Finally, we sort vertices and edges by increasing labels. This produces the desired output. As for the size of the separator, note that procedure SEPARATECONNECTED computes a $\sqrt{N}$-separator S of G'. That is, $|S| = O\left(\sqrt{|G'|}\right) = O\left(\sqrt{|G|}\right) = O(\sqrt{N})$ because $G' \subseteq G$. Thus, Algorithm 6.1 is correct.

By Lemmas 6.1, 6.3, and 6.4, the I/O-complexity of each of the procedures COMPUTECONNECTEDCOMPONENTS, FINDHEAVIESTCOMPONENT, and COLLECTCOMPONENTS is $O(sort(N))$. By Lemma 6.5, the I/O-complexity of procedure SEPARATECONNECTED is $O(bfs(N) + sort(N))$.

There is a small detail that we have to deal with. How do we pass subgraphs of G to procedures SEPARATECONNECTED and COMPUTECONNECTEDCOMPONENTS? The input of these algorithms are given as disk addresses and sizes of $V(G)$ and $E(G)$, where G is the input of the algorithm.

Procedure FINDHEAVIESTCOMPONENT returns $V(G)$ and $E(G)$ sorted such that the vertices in $V(G')$ and the edges in $E(G')$ will be stored consecutively. Then we can just pass the addresses and sizes of $V(G')$ and $E(G')$ to SEPARATECONNECTED. Before calling COMPUTECONNECTEDCOMPONENTS in line 5 of Algorithm 6.1, we sort $V(G')$ and $E(G')$ by increasing component labels. Then we pass the address of the first vertex v in $V(G')$ with $C(v) > 0$ and $|V(G') \setminus C|$ to COMPUTECONNECTEDCOMPONENTS. We pass the address of $E(G')$ to COMPUTECONNECTEDCOMPONENTS and exclude all edges with label $C(e) = \infty$ from the size of $E(G')$. These edges are stored at the end of $E(G)$. This preparation of the input for procedure

ComputeConnectedComponent takes $O(sort(N))$ I/Os. Thus, Algorithm 6.1 takes $O(bfs(N) + sort(N))$ I/Os. □

Corollary 6.1 *Given an embedded planar graph G with N vertices, we can compute a $\sqrt{N}$-separator of G with $O(bfs(N))$ I/Os, where $bfs(N)$ is the number of I/Os required to do breadth-first search in a planar graph with N vertices.*

Proof. The previous theorem states that Algorithm 6.1 computes the desired separator with $O(bfs(N) + sort(N))$ I/Os. We can run Algorithm 6.1 and Lipton and Tarjan's internal-memory algorithm simultaneously. Each algorithm has its own copy of the graph. Whenever the currently running algorithm is about to perform an I/O-operation, we stop it and start the other algorithm. This algorithm now actually performs the I/O-operation that it wanted to perform when we stopped it the last time. We continue its execution until it is about to perform the next I/O. Then we switch back to the algorithm that we just stopped. This guarantees that, at a given time, the numbers of I/Os that both algorithms have performed differ by at most one. We use the output of the algorithm that produces its output first.

The internal-memory algorithm takes $O(N)$ I/Os in the worst case. Thus, our simulation of both algorithms takes $O(\min\{bfs(N) + sort(N), N\})$ I/Os.

As already mentioned, the I/O-complexity of list ranking is $\Omega(perm(N))$. Considering a list as a planar graph, list ranking reduces to breadth-first search in the list. Thus, breadth-first search has a lower bound of $\Omega(perm(N)) = \Omega(\min\{N, sort(N)\})$ I/Os.

If $perm(N) = sort(N)$, i.e., $N \geq sort(N)$, $bfs(N) = \Omega(sort(N))$. Thus, the I/O-complexity of our simulation algorithm becomes $O(\min\{bfs(N), N\}) = O(bfs(N))$.

If $perm(N) < sort(N)$, i.e., $N < sort(N)$, the I/O-complexity of our simulation becomes $O(N) = O(perm(N)) = O(bfs(N))$. Thus, in either case, we can compute a planar separator with $O(bfs(N))$ I/Os. □

6.2.1 Computing Connected Components

The next lemma states that procedure ComputeConnectedComponents takes $O(sort(N))$ I/Os.

LEMMA 6.1 *Given a planar graph G with N vertices, we can compute the connected components of G with $O(sort(N))$ I/Os.*

PROOF. We can use an algorithm by Chiang *et al.* [12] that computes the connected components of G. This algorithm takes $O(sort(N))$ I/Os for sparse graphs. Here, sparsity means that $|E(G)| = O(|V(G)|)$. This is true for planar graphs.

However, the authors of [12] did not specify the output representation of their algorithm. We assume that it labels the vertices of G so that two vertices belong to the same connected component of G iff they have the same label.

In order to ensure that Step 5 of Algorithm 6.1 does not produce the same labels as Step 1, we first relabel each component with the name of a vertex in it. This guarantees that each component labeled in Steps 1 or 5 has a different label.

To do this, we sort the vertex list $V(G)$ of G by increasing component labels. Then we scan $V(G)$. If the current vertex v belongs to the same component as its predecessor w, we assign a new component label $C(v) = C(w)$ to v. Otherwise, we define $C(v) = v$. As this involves sorting $V(G)$ once and scanning it once, this takes $O(sort(N))$ I/Os.

To label the edges of G, we copy the label from the first endpoint of each edge to the edge itself. It suffices to consider only the first endpoint, as both endpoints are in the same connected component. This copying is done by procedure COPYLABELS. By the next lemma, procedure COPYLABELS takes $O(sort(N))$ I/Os. Hence, labeling all vertices and edges of G with appropriate component labels takes $O(sort(N))$ I/Os. □

We will use procedure COPYLABELS again in this and the next chapter. We will show that it works correctly and takes $O(sort(N))$ I/Os, no matter whether we copy the information from the first or the second endpoint of an edge. However, the endpoint that provides the information to be copied must be known in advance for each edge.

LEMMA 6.2 *Given a graph G with N vertices. For each edge $e = \{v, w\}$ of G, let $v_s(e) \in \{v, w\}$. Then copying a constant amount of information from $v_s(e)$ to e, for all edges e of G, takes $O(sort(N))$ I/Os.*

COPYLABELS(G):

Input:

- A planar graph G as a vertex list $V(G)$ and an edge list $E(G)$,
- For each edge $e = \{v, w\}$, a label $v_s(e) \in \{v, w\}$ specifying the endpoint from which to copy the information, and
- Names of $O(1)$ labels that have to be copied.

Output: $V(G)$ and $E(G)$ with the labels copied to the edges in $E(G)$.

1: Sort $V(G)$ by increasing vertex names and $E(G)$ by increasing labels $v_s(e)$.
2: Scan $V(G)$ and $E(G)$ simultaneously as follows:
{v is the currently processed vertex. e is the currently processed edge.}
if $v < v_s(e)$ **then**
Advance to the next vertex in $V(G)$.
else
{If $v = v_s(e)$.}
Copy the specified information from v to e. Advance to the next edge in $E(G)$.
end if

Algorithm 6.2: Copying information from vertices to edges.

The choice of $v_s(e)$ for each edge e depends on the context where we apply the procedure. In the proof of the previous lemma, $v_s(e)$ is the first endpoint of e.

PROOF. Algorithm 6.2 shows the pseudo-code of COPYLABELS. Let us first prove its correctness. If $v = v_s(e)$ in Step 2, we copy the information from v to e as desired. After that we are done with edge e. Thus, we can proceed to the next edge in $E(G)$. If $v < v_s(e)$, $v_s(e)$ must be stored after v in $V(G)$ because $V(G)$ is sorted by increasing vertex names. As $v_s(e)$ is in $V(G)$, we will find it by searching forward in $V(G)$. Then we will have the case $v = v_s(e)$ again. The case $v > v_s(e)$ cannot occur. Assume that we have this case. As $v' = v_s(e)$ is in $V(G)$, v' must have been the current vertex at some point during the scan of $V(G)$. Then we must have advanced from v' to the next vertex in $V(G)$ because $v' < v$ and thus v is stored after v' in $V(G)$. This happens only if $v' < v_s(e')$, for some edge e' stored before e in $V(G)$. That is, $v_s(e') > v' = v_s(e)$, in contradiction to the order by which $E(G)$ is sorted. Thus,

Algorithm 6.2 is correct.

Let us analyze its I/O-complexity. As $|V(G)| = N$ and $|E(G)| = O(N)$, Step 1 takes $O(sort(N))$ I/Os. The second step consists of two simultaneous scans. Keeping a buffer of size DB in internal memory for each scan, these two scans take $O(scan(N))$ I/Os. Thus, Algorithm 6.2 takes $O(sort(N))$ I/Os. □

Obviously, we can use Algorithm 6.2 to copy information from edges e to their vertices $v_s(e)$ as well.

6.2.2 Finding the Heaviest Component

In this section, we show how to find the heaviest connected component in a graph. We can use the following lemma in both Steps 2 and 6 of Algorithm 6.1 because, in Step 2, $S = \emptyset$. That is, in Step 2, the connected components of $G - S$ are just the connected components of G.

LEMMA 6.3 *Procedure* FINDHEAVIESTCOMPONENT *computes the heaviest component of $G - S$ with $O(sort(N))$ I/Os.*

PROOF. Algorithm 6.3 shows the pseudo-code of procedure FINDHEAVIESTCOMPONENT.

Once the vertex set $V(G)$ is sorted by increasing component labels, the vertices of each component are stored consecutively. The vertices in the separator C are stored at the beginning of $V(G)$. Thus, we can indeed scan $V(G)$ and sum the weights of all consecutive vertices with equal labels, in order to compute the component weights. For the same reason, our method for computing the addresses and sizes of $V(G')$ and $E(G')$ is correct.

Step 1 takes $O(sort(N))$ I/Os, Steps 2 and 6 together scan $V(G)$ once. Steps 3–5 do not require any I/Os. Step 7 scans $V(G)$ once and $E(G)$ once. Thus, this algorithm takes $O(sort(N))$ I/Os. □

FINDHEAVIESTCOMPONENT(G):

Input: The vertex set $V(G)$ and the edge set $E(G)$ of the graph G. Vertices and edges are labeled with component labels $C(x)$. Two vertices or edges are in the same component iff they have the same label. If $C(v) = 0$, v is a separator vertex. If $C(e) = \infty$, e is not in $G - S$.

Output: The label and weight of the heaviest connected component G' in G. The sets $V(G)$ and $E(G)$ sorted such that the vertices (resp. edges) of G' are stored consecutively. The addresses and sizes of $V(G')$ and $E(G')$.

1: Sort $V(G)$ and $E(G)$ by increasing component labels $C(x)$.

2: Scan $V(G)$ until we find the first vertex v with $C(v) > 0$.
{Skip separator vertices.}

3: $w_{\max} \leftarrow 0$ {The weight of the heaviest component.}

4: $C_{\max} \leftarrow 0$ {The label of the heaviest component.}

5: $w \leftarrow 0$ {The weight of the current component.}

6: Continue scanning $V(G)$:
{v is the currently processed vertex.}

    $w \leftarrow w + w(v)$
    $u \leftarrow$ next vertex after v in $V(G)$
    if $C(u) \neq C(v)$ or v is the last vertex in $V(G)$ **then**
        if $w > w_{\max}$ **then**
            $w_{\max} \leftarrow w$
            $C_{\max} \leftarrow C(v)$
        end if
        $w \leftarrow 0$
    end if

7: Scan $V(G)$ and $E(G)$ to compute:

- The addresses of the first vertex v with $C(v) = C_{\max}$,
- The number of vertices v with $C(v) = C_{\max}$,
- The addresses of the first edge e with $C(e) = C_{\max}$, and
- The number of edges e with $C(e) = C_{\max}$.

Algorithm 6.3: Finding the heaviest component.

6.2.3 Constructing A and B

Algorithm 6.4 shows the pseudo-code of procedure CollectComponents. We prove the following lemma.

Lemma 6.4 *Procedure* CollectComponents *constructs sets A and B of weight at most $\frac{2}{3}$ each. This takes $O(sort(N))$ I/Os.*

Proof. Let us first prove the correctness of Algorithm 6.4. Before we call procedure CollectComponents in Algorithm 6.1, we ensure that no connect component of $G - S$ has weight greater than $\frac{2}{3}$. That is, if the heaviest component G' of $G - S$ has weight $w(G') \geq \frac{1}{3}$, then $\frac{1}{3} \leq w(G') \leq \frac{2}{3}$. We choose $A = G'$ and $B = G - S - G'$. Then $w(A) = w(G') \leq \frac{2}{3}$ and $w(B) \leq w(G) - w(A) \leq 1 - \frac{1}{3} = \frac{2}{3}$. Thus, our strategy is correct in this case.

If the heaviest component G' has weight less than $\frac{1}{3}$, we add the components of $G - S$ to A, one by one, until $w(A) \geq \frac{1}{3}$. Before we added the last component to A, the weight of A was less than $\frac{1}{3}$. The last component has weight less than $\frac{1}{3}$. Thus, the weight of A is less than $\frac{2}{3}$. As the weight of A is at least $\frac{1}{3}$, the weight of B is at most $\frac{2}{3}$. Thus, our strategy works in this case as well.

To see that the scan in Step 5 works, note that $V(G)$ and $E(G)$ are sorted by increasing component labels. The scans in lines 2 and 4 use only local information. Thus, Algorithm 6.4 is correct.

In either of the two cases, we scan $V(G)$ and $E(G)$ once. Afterwards, we sort $V(G)$ and $E(G)$ once. Hence, Algorithm 6.4 takes $O(sort(N))$ I/Os. □

We have provided the framework of our algorithm. It remains to show how to find a small separator of a connected graph. The next section is dedicated to this problem.

6.3 Separating Connected Planar Graphs

Procedure SeparateConnected computes a $\sqrt{N}$-separator of a given connected planar graph. It consists of two major steps. In the first step, we triangulate the

COLLECTCOMPONENTS(G, G'):

Input: The vertex set $V(G)$ and the edge set $E(G)$ of the graph G. Vertices and edges are labeled with component labels $C(x)$. Two vertices or edges are in the same component iff they have the same label. If $C(v) = 0$, v is a separator vertex. If $C(e) = \infty$, e is not in $G - S$. $V(G)$ and $E(G)$ are sorted by increasing component labels. The label $C(G')$ such that all vertices and edges in the connected component G' have label $C(x) = C(G')$. The weight $w(G')$ of G'.

Output: $V(G)$ and $E(G)$ such that

- all elements of the separator $S \subseteq V(G)$ are labeled with 0,
- all edges in $E(G) \setminus E(G - S)$ are labeled with ∞,
- all other vertices and edges are labeled with 1, if they are in A, and with 2, if they are in B, and
- $V(G)$ and $E(G)$ are sorted by increasing labels.

1: **if** $w(G') \geq \frac{1}{3}$ **then**
2: Scan $V(G)$ and $E(G)$ to relabel vertices and edges. Relabel a vertex v with 1, if $C(v) = C(G')$, with 2, if $0 < C(v) \neq C(G')$, and with 0, if $C(v) = 0$. Relabel an edge e with 1, if $C(e) = C(G')$, with 2, if $C(G') \neq C(e) < \infty$, and with ∞, if $C(e) = \infty$.
3: **else**
4: Scan $V(G)$ until we find the first vertex v with $C(v) > 0$.
5: Continue scanning $V(G)$ and $E(G)$ to relabel vertices and edges: $\{v$ is the current vertex in $V(G)$. e is the current edge in $E(G)$.$\}$
  $w \leftarrow 0$
  while $w < \frac{1}{3}$ **do**
    Relabel all vertices and edges with the same label as v with 1. Add the weights of the vertices to w.
  end while
  Relabel the remaining vertices in $V(G)$ and edges in $E(G)$ with 2. Stop relabeling edges as soon has we find the first edge e with $C(e) = \infty$.
6: **end if**
7: Sort $V(G)$ and $E(G)$ by increasing vertex and edge labels.

Algorithm 6.4: Constructing components A and B.

SEPARATECONNECTED(G):

Input: The vertex set $V(G)$ and the edge set $E(G)$ of the graph G. Vertices and edges are labeled with the same positive component label.

Output: $V(G)$ and $E(G)$ such that all vertices in the separator S of G have component label $C(v) = 0$. All edges in $E(G) \setminus E(G - S)$ have component label $C(e) = \infty$.

1: Scan $V(G)$ to check whether there is a vertex v with $w(v) \geq \frac{1}{3}$. Label the first such vertex with $C(v) = 0$, if it exists.
2: **if** there is no vertex v with $w(v) \geq \frac{1}{3}$ **then**
3: $G' \leftarrow$ TRIANGULATE(G)
{Compute a triangulation G' of G. All edges in $E(G') \setminus E(G)$ are marked.}
4: SEPARATETRIANGULATION(G')
{Compute a $\sqrt{N}$-separator S of G'. Label all vertices in S with component label 0.}
5: Scan $E(G')$ to remove all marked edges.
{This produces G again.}
6: **end if**
7: Use procedure COPYLABELS to label all edges that have at least one endpoint in C with $C(e) = \infty$.

Algorithm 6.5: Separating an embedded connected planar graph.

given graph G. In the second step, we compute a $\sqrt{N}$-separator of this triangulation, which is also a separator of G.

The pseudo-code of procedure SEPARATECONNECTED is shown in Algorithm 6.5.

LEMMA 6.5 *Given an embedded planar graph G with N vertices, procedure* SEPARATECONNECTED *computes a separator S of G with $O(bfs(N) + sort(N))$ I/Os. The size of S is at most $2\sqrt{2}\sqrt{N}$.*

PROOF. If there is a vertex v with $w(v) \geq \frac{1}{3}$, $w(G - v) \leq \frac{2}{3}$. Thus, if we set $S = \{v\}$, all components of $G - S$ have weight at most $\frac{2}{3}$. Otherwise, by Theorem 7.1, procedure TRIANGULATE computes a triangulation G' of G. By Lemma 6.6, procedure SEPARATETRIANGULATION computes a separator S of G' with $|S| \leq 2\sqrt{2}\sqrt{N}$. As G is a subgraph of G', S is a separator of G as well.

The edges in $E(G) \setminus E(G - S)$ are exactly the edges with at least one vertex in C. Thus, we label the right edges in Step 7.

Step 1 takes $O(scan(N))$ I/Os, as we scan $V(G)$ once. By Theorem 7.1, procedure TRIANGULATE takes $O(sort(N))$ I/Os. The details of this procedure are presented in Chapter 7. Procedure SEPARATETRIANGULATION takes $O(bfs(N)+sort(N))$ I/Os, by Lemma 6.6. Removing marked edges takes a single scan over $E(G')$. $|E(G'| \leq 3N-6$. Thus, this scan takes $O(scan(N))$ I/Os. Step 7 takes $O(sort(N))$ I/Os, by Lemma 6.2. Hence, the I/O-complexity of Algorithm 6.5 is $O(sort(N) + bfs(N))$. □

It remains to provide the details of procedure SEPARATETRIANGULATION. It follows the same lines as the algorithm by Lipton and Tarjan [26]. Thus, the correctness of our algorithm follows from that of Lipton and Tarjan's algorithm, if we can prove that the used external-memory techniques work. The basic idea used by Lipton and Tarjan is as follows:

Compute a BFS-tree T of the given graph G. As T is a BFS-tree, the two endpoints of a non-tree edge in G are at most one level apart. Therefore, the removal of a complete level from T separates the upper part from the lower part.

We compute the level l_1 closest to the root r of T such that the total weight of the vertices on levels 0 through l_1 exceeds $\frac{1}{2}$. Let K be the number of vertices on levels 0 through l_1. Then we compute levels $l_0 \leq l_1$ and $l_2 > l_1$ such that $L(l_0) + 2(l_1 - l_0) \leq 2\sqrt{K}$ and $L(l_2) + 2(l_2 - l_1 - 1) \leq 2\sqrt{N-K}$, where $L(l)$ is the number of vertices on level l. Lipton and Tarjan proved that two such levels l_0 and l_2 exist.

Now, l_0 and l_2 cut T (and G) into three parts: levels 0 through $l_0 - 1$, levels $l_0 + 1$ through $l_2 - 1$, and levels $l_2 + 1$ and greater. Call these parts G_1, G_2, and G_3. By the construction of l_0 and l_2, $w(G_1) \leq \frac{1}{2}$ and $w(G_3) \leq \frac{1}{2}$. We cannot give an upper bound on the weight of G_2. However, we will construct a separator of G_2 of size at most $2(l_2 - l_0 - 1)$. This separator together with the vertices on levels l_0 and l_2 gives a separator of G. Its size is bounded by

$$L(l_0) + L(l_2) + 2(l_2 - l_1 - 1) \leq 2\sqrt{K} + 2\sqrt{N-K} \leq 2\sqrt{2}\sqrt{N}.$$

The pseudo-code of our algorithm is shown in Algorithm 6.6.

SEPARATETRIANGULATION(G):

Input: The vertex set $V(G)$ and the edge set $E(G)$ of the triangulation G. Each vertex v in $V(G)$ is labeled with a component label $C(v) > 0$.

Output: The vertex set $V(G)$ and the edge set $E(G)$. Vertices v in the separator S of $V(G)$ are labeled with $C(v) = 0$. The component labels of all other vertices remain unchanged.

1: Compute a BFS-tree T of G with root r.
{We assume that this algorithm labels each vertex v of G with labels $d(r, v)$ (distance to the root) and $p(v)$ (parent in T). All edges of G are labeled as tree edge or non-tree edge.}

2: Compute level sizes $L(l)$ and level weights $W(l)$:
Sort $V(G)$ by increasing labels $d(r, v)$.
Scan $V(G)$ to count vertices and sum vertex weights on each level. Write these level sizes and weights to lists L and W.

3: Compute l_0 and l_2:
Scan W and sum level weights until the total weight exceeds $\frac{1}{2}$. This gives us level l_1.
Scan L to compute the prefix sum $K = L(0) + L(1) + \cdots + L(l_1)$.
Scan L backward from l_1 to find l_0. Scan L forward from l_1 to find l_2.

4: Scan $V(G)$ and label all vertices v with $d(r, v) \in \{l_0, l_2\}$ with $C(v) = 0$.

5: $q \leftarrow$ ORDERCHILDREN(G, T):
{This computes labels q such that, for two children w_1 and w_2 of a vertex v, $q(w_1) < q(w_2)$, if the edge $\{v, w_1\}$ appears between the edges $\{v, p(v)\}$ and $\{v, w_2\}$ counterclockwise around v.}

6: $n_p \leftarrow$ COMPUTEPREORDERNUMBERS($V(G), q$):
{This uses the labels in q to ensure that the Euler tour defining the preorder numbers always chooses the leftmost non-traversed child of the current vertex.}

7: SEPARATEMIDDLEPART(G, l_0, l_2, n_p)
{Compute a separator of G_2.}

Algorithm 6.6: Separating a triangulation.

LEMMA 6.6 *Algorithm 6.6 constructs a $\sqrt{N}$-separator of a given triangulation G with $O(bfs(N) + sort(N))$ I/Os.*

PROOF. Let us first prove the correctness of Algorithm 6.6. For Step 1, we use the best known algorithm for computing a BFS-tree in external memory (e.g., [24]). We may assume that it is correct. Step 2 first ensures that the vertices are sorted by increasing distance to the root. That is, all vertices on the same level in T are stored consecutively. Thus, counting these vertices and computing the weight of each level can be done with a single scan of $V(G)$. Note that the lists L and W are sorted by increasing level numbers because $V(G)$ is. Therfore, in Step 3, we can scan W and sum the weights in the traversed sublist of W to compute l_1. For the same reason, the scan of L gives us K, and it takes two additional scans to compute l_0 and l_2. As each vertex is labeled with its distance to the root, Step 4 takes another scan.

By Lemma 6.9, procedure SEPARATEMIDDLEPART computes a separator of size at most $2(l_2 - l_0 - 1)$. Thus, our algorithm realizes Lipton and Tarjan's approach and is correct.

The output produced by procedure ORDERCHILDREN is used by procedure COMPUTEPREORDERNUMBERS to ensure that the Euler tour defining these preorder numbers always chooses the leftmost non-traversed child of a vertex. Lemma 6.7 states that procedure ORDERCHILDREN indeed produces labels $q(v)$ as in the comment of Step 5. Observation 4.1 implies that these labels $q(v)$ are sufficient to ensure that the defining Euler tour of the preorder numbers $n_p(v)$ always chooses the leftmost non-traversed child of a vertex. These preorder numbers are used by procedure SEPARATEMIDDLEPART. To compute preorder numbers, we use Algorithm 4.4. We can provide the right input for this algorithm because every vertex in $V(G)$ has a pointer to its parent $p(v)$ in T.

Step 1 takes $O(bfs(N))$ I/Os. Step 2 takes $O(sort(N))$ I/Os. Steps 3 and 4 take $O(scan(N))$ I/Os. By Lemmas 6.7, 4.4, and 6.9, procedures ORDERCHILDREN, COMPUTEPREORDERNUMBERS, and SEPARATEMIDDLEPART take $O(sort(N))$ I/Os. Thus, Algorithm 6.6 takes $O(bfs(N) + sort(N))$ I/Os. □

Next, we present procedure ORDERCHILDREN.

ORDERCHILDREN(G, T):

Input: The vertex set $V(G)$ and the edge set $E(G)$ of the triangulation G.

Output: Labels $q(v)$ for all vertices of G such that, for two children w_1 and w_2 of a vertex v, $q(w_1) < q(w_2)$ iff w_1 is to the left of w_2 with respect to v.

1: CHANGEREFEDGE(G, T)
{Renumber the edges around each vertex v so that $\{v, p(v)\}$ is the reference edge for vertex v.}

2: Use procedure COPYLABELS to assign $q(v) \leftarrow n_{p(v)}(\{v, p(v)\})$.

Algorithm 6.7: Computing Labels q for the vertices of G.

LEMMA 6.7 *Algorithm 6.7 produces the desired labels $q(v)$ with $O(sort(N))$ I/Os.*

PROOF. Let w_1 and w_2 be two children of v. w_1 appears to the left of w_2. Then, after the execution of procedure CHANGEREFEDGE, $n_v(\{v, w_1\}) < n_v(\{v, w_2\})$. In Step 2, we assign $q(w_1) = n_v(\{v, w_1\})$ and $q(w_2) = n_v(\{v, w_2\})$. Thus, $q(w_1) < q(w_2)$ as desired.

By Lemmas 6.8 and 6.2, procedures CHANGEREFEDGE and COPYLABELS take $O(sort(N))$ I/Os. Thus, Algorithm 6.7 takes $O(sort(N))$ I/Os. □

LEMMA 6.8 *Algorithm 6.8 changes the reference edge of each vertex v to $\{v, p(v)\}$. This takes $O(sort(N))$ I/Os.*

PROOF. In Steps 1 and 2, we create a list E'. This list is the concatenation of edge lists $E'(v)$, where $E'(v)$ is the list of all edges incident on vertex v. Each list $E'(v)$ is sorted counterclockwise around v, starting at v's reference edge. Let $e_0, \ldots, e_k$ be the list $E'(v)$; e_i be the edge $\{v, p(v)\}$. Then, in Step 3, we compute i and k and store these values with e_0. In Step 4, we assign new labels n_v to the edges $e_0, \ldots, e_k$: For $j < i$, $n_v(i) = j + k - i + 1$. For $j \geq i$, $n_v(i) = j - i$. This makes e_i the new reference edge and does not change the circular order of the edges around v.

E' contains two copies of an edge $e = \{v, w\} \in E(G)$. One has an updated value of $n_v(e)$. For the other copy, we updated $n_w(e)$. Step 5 ensures that these two copies

ChangeRefEdge(G, T):

Input: The vertex set $V(G)$ and the edge set $E(G)$ of the triangulation G.

Output: The edges in $E(G)$ relabeled such that $n_v(e_0) < n_v(e_1) < \cdots < n_v(e_k)$, where $e_0, \ldots, n_k$ are the edges incident on a vertex v, counterclockwise around v, and $e_0 = \{v, p(v)\}$.

1: Scan $E(G)$ to create an edge list E':
$\{e = \{v, w\}$ is the current edge.$\}$
   Make two copies e' and e'' of e in E'. Let $v(e') = v$, $n_v(e') = n_v(e)$, $v(e'') = w$, and $n_v(e'') = n_w(e)$.

2: Sort E' by increasing labels $v(e)$ and $n_v(e)$.

3: Scan E' backward:
$\{e$ is the current edge. e' is the previously visited edge.$\}$

```
if e is the first visited edge or v(e) ≠ v(e') then
  k ← 0                          {The number of edges incident on v(e).}
  i ← 0                          {The index of the edge {v(e), p(v(e))}.}
end if
if e is a tree edge and d(r, v(e)) > d(r, w) then
  i ← 0                          {e = {v(e), w}
end if
if e is the last edge with label v(e) during our traversal then
  k(e) ← k, i(e) ← i
end if
k ← k + 1, i ← i + 1
```

4: Scan E' forward:
$\{e$ is the current edge. e' is the previous edge.$\}$

```
if e is the first edge or v(e) ≠ v(e') then
  i ← k(e) − i(e) + 1
end if
if e is a tree edge and d(r, v(e)) > d(r, w) then
  i ← 0                          {e = {v(e), w}
end if
n_v(e) ← i, i ← i + 1
```

5: Sort E' by increasing endpoints.
$\{$For each edge $\{v, w\}$, $v < w$. Then $\{v, w\} < \{v', w'\}$ iff $v < v'$ or $v = v'$ and $w < w'$.$\}$

6: Scan E' and merge the two copies of each edge $e \in E(G)$:
   Let e' and e'' be the two copies. They are stored consecutively.
   Let $v(e') = v$ and $v(e'') = w$. Assign $n_v(e) = n_v(e')$ and $n_w(e) = n_v(e'')$.

Algorithm 6.8: Changing the reference edges for all vertices of G.

SEPARATEMIDDLEPART(G, l_0, l_2, n_p):

Input: The vertex set $V(G)$ and the edge set $E(G)$ of the triangulation G. Two levels l_0 and l_2 that bound the middle part G_2 of G. Preorder numbers $n_p(v)$ of the vertices of T (i.e., G).

Output: The vertices v in $V(G)$ that form a separator of G_2 labeled with $C(v) = 0$.

1: Construct G_2:
   Scan $V(G)$ and mark all vertices v with $d(r, v) \leq l_0$ or $d(r, v) \geq l_2$.
   Use COPYLABELS to copy the labels $d(r, v)$, $d(r, w)$, $n_p(v)$, and $n_p(w)$ to every edge $\{v, w\}$.
   Scan $E(G)$ and mark all edges with at least one marked endpoint. Do not mark tree edges $\{v, w\}$ with $d(r, v) = l_0$ and $d(r, w) = l_0 + 1$.
   Scan $V(G)$ and $E(G)$ and move all unmarked edges to $V(G_2)$ and $E(G_2)$.

2: $\hat{G} \leftarrow$ MAKETRIANGULATION(G_2, l_0, n_p)
   {This also constructs a BFS-tree $\hat{T}$ of $\hat{G}$. This procedure adds a vertex $\hat{r}$ with $w(\hat{r}) = 0$ and new edges to G_2. We mark this vertex and these edges, in order to remove them later.}

3: FINDTREECYCLE($\hat{G}, \hat{T}$)
   {This labels the vertices v on the tree cycle with $C(v) = 0$.}

4: Scan $V(\hat{G})$ and $E(\hat{G})$ to copy unmarked vertices and edges back to $V(G)$ and $E(G)$.
   {This removes all marked vertices and edges from $\hat{G}$ and reconstructs G.}

Algorithm 6.9: Separating G_2.

are stored consecutively. Step 6 "merges" them. That is, we create a single copy of e with both $n_v(e)$ and $n_w(e)$ updated. Thus, Algorithm 6.8 is correct.

Steps 2 and 5 take $O(sort(N))$ I/Os because $|E'| = O(N)$. Steps 1, 3, 4, and 6 take $O(scan(N))$ I/Os. Thus, Algorithm 6.8 takes $O(sort(N))$ I/Os. □

It remains to provide the details of procedure SEPARATEMIDDLEPART. The idea is to triangulate G_2 and to construct a BFS-tree of the resulting triangulation. Then, we compute a small simple cycle separator of this triangulation. Of course, this is also a separator of G_2.

LEMMA 6.9 *Algorithm 6.9 computes a separator of G_2 with $O(sort(N))$ I/Os. The size of the separator is at most $2(l_2 - l_0 - 1)$.*

PROOF. G_2 is the subgraph of G induced by all vertices v with $l_0 < d(r,v) < l_2$. Thus, if procedure COPYLABELS is correct, Step 1 is correct. Lemma 6.2 states the correctness of procedure COPYLABELS. What we construct is not exactly the graph G_2. The constructed edge set contains $E(G_2)$ plus all tree edges between levels l_0 and $l_0 + 1$. We will need these edges in procedure MAKETRIANGULATION.

By Lemma 6.10, procedure MAKETRIANGULATION constructs a triangulation $\hat{G}$ and a spanning tree $\hat{T}$ of $\hat{G}$ of height $l_2 - l_0$. The root of this spanning tree is $\hat{r}$. By Lemma 6.11, the separator S constructed by procedure FINDTREECYCLE is a path in $\hat{T}$ of size at most $2(l_2 - l_0) - 1$. If its length is $2(l_2 - l_0) - 1$, it must contain the root $\hat{r}$ of $\hat{T}$. However, $\hat{r}$ is no vertex of G_2. Thus, $|S \cap V(G_2)| \leq 2(l_2 - l_0 - 1)$. As G_2 is a subgraph of $\hat{G}$, S is a separator of G_2.

Step 4 copies all vertices and edges of G_2 back into $V(G)$ and $E(G)$. This automatically removes $\hat{r}$ from the separator.

Steps 1 and 4 scan lists of size $O(N)$. This takes $O(scan(N))$ I/Os. By Lemmas 6.10 and 6.11, procedures MAKETRIANGULATION and FINDTREECYCLE take $O(sort(N))$ I/Os. Thus, Algorithm 6.9 takes $O(sort(N))$ I/Os. □

Procedure MAKETRIANGULATION uses G_2 and the subtrees of T in G_2 to construct a triangulation $\hat{G}$ and a BFS-tree of $\hat{G}$ efficiently. More precisely, we add a new vertex $\hat{r}$ to T and make all vertices on level $l_0 + 1$ children of $\hat{r}$. This creates the tree $\hat{T}$. We add the corresponding edges to G and triangulate the resulting graph. The result is $\hat{G}$. $\hat{T}$ is a BFS-tree of $\hat{G}$ [26].

We also have to embed the new edges of $\hat{G}$ properly. To triangulate the graph, we use procedure TRIANGULATE, as presented in Chapter 7. This procedure embeds new edges properly. Thus, we only have to deal with the tree edges between $\hat{r}$ and the vertices on level $l_0 + 1$. This is what we use the additional edges in G_2 for.

For each vertex v in G_2 with $d(r,v) = l_0 + 1$, there is an edge $\{v, p(v)\}$ in $E(G_2)$. For each such edge, we replace $p(v)$ by $\hat{r}$ (the new parent of v). Basically, this adds the desired tree edges $\{v, \hat{r}\}$ to $\hat{G}$. We sort these edges by increasing preorder numbers $n_p(v)$. Let $e_1, \ldots, e_k$ be the resulting list of edges. Then we set $n_{\hat{r}}(e_i) = i$. We claim that this embeds these edges properly.

Consider an edge $\{v, \hat{r}\}$. With respect to v, this edge is already embedded properly

MAKETRIANGULATION(G_2, l_0, n_p):

Input: The vertex set $V(G_2)$ of the graph G_2 and an edge set E. E contains $E(G_2)$ and all tree edges in G between vertices on levels l_0 and $l_0 + 1$. The level l_0. Preorder numbers (in T) of the vertices of G_2.

Output: The triangulation $\hat{G}$.

1: Add $\hat{r}$ to $V(G_2)$. $d(r, v) \leftarrow l_0$.

2: Add edges $\{v, \hat{r}\}$ for all vertices v with $d(r, v) = l_0 + 1$ to $E(G_2)$:

Scan E and move all edges $\{v, w\}$ with $d(r, w) = l_0$ and $d(r, v) = l_0 + 1$ to a set E'.
Sort E' by increasing preorder numbers $n_p(v)$, where $d(r, v) = l_0 + 1$.
Scan $E' = (e_1, \ldots, e_k)$, replace $e_i = \{v_i, w_i\}$ by $e'_i = \{v_i, \hat{r}\}$, where $d(r, v_i) = l_0 + 1$. Set $n_{\hat{r}}(e'_i) = i$ and $n_{v_i}(e'_i) = n_{v_i}(e_i)$. Mark e'_i as a tree edge.
Scan E' to append it to $E = E(G_2)$.

3: Scan $V(G_2)$: For each vertex, $d(r, v) \leftarrow d(r, v) - l_0$. For each vertex with $d(r, v) = 1$, $p(v) \leftarrow \hat{r}$.

4: $\hat{G} \leftarrow$ TRIANGULATE(G_2)

Algorithm 6.10: Augmenting G_2 to a triangulation $\hat{G}$.

because $\{v, p(v)\}$ was. Let $v_1, \ldots, v_k$ be the vertices on level $l_0 + 1$ sorted from left to right in T. Then $n_p(v_i) < n_p(v_j)$, for $i < j$, because the preorder numbers respect the embedding. Thus, $e_i = \{v_i, \hat{r}\}$, and $e_1, \ldots, e_k$ is the order of these edges counterclockwise around $\hat{r}$.

Algorithm 6.10 provides the pseudo-code of procedure MAKETRIANGULATION.

LEMMA 6.10 *Algorithm 6.10 constructs the triangulation $\hat{G}$ and the spanning tree $\hat{T}$ of $\hat{G}$ with $O(sort(N))$ I/Os.*

PROOF. It follows from the above discussion that Algorithm 6.10 constructs and embeds $\hat{G}$ properly. In Step 3, we update the distances to the root of all vertices and the parents of all vertices that are now children of $\hat{r}$. We mark all edges between $\hat{r}$ and its children as tree edges. All other edges are not marked, i.e., are non-tree edges. That is, we construct $\hat{T}$ properly.

Step 1 takes $O(1)$ I/Os. Step 2 involves three scans and a single sort. Thus, it takes $O(sort(N))$ I/Os. Step 3 takes $O(scan(N))$ I/Os. By, Theorem 7.1, Step 4 takes $O(sort(N))$ I/Os. Thus, Algorithm 6.10 takes $O(sort(N))$ I/Os. □

6.4 Finding a Simple Cycle Separator in an Embedded Triangulation

It remains to provide the details of procedure FINDTREECYCLE. The input is a triangulation G and a BFS-tree T of G. The idea is as follows:

Given a non-tree edge $e = \{v, w\}$ of G, this edge defines a cycle $c(e)$ in G. This cycle consists of e itself and the two tree paths from v to u and from w to u, where u is the lowest common ancestor of v and w in T. Call this cycle e's *tree cycle*. It separates the vertices inside $c(e)$ from those outside $c(e)$. Lipton and Tarjan proved that, in a triangulation G with $w(v) \leq \frac{1}{3}$, for all its vertices, there is a non-tree edge e such that $c(e)$ is a separator of G [26].

The main problem is to find the tree edge e. Lipton and Tarjan's solution to this problem does not easily translate into an efficient external-memory algorithm. That is why we take another approach. The basic idea is to compute all sorts of labels for all edges of G. Then, we scan the list of non-tree edges and use these labels to determine, for each non-tree edge, the weights of the two regions inside and outside the corresponding tree cycle. Once, we have found the non-tree edge e, we label the vertices on $c(e)$ with $C(v) = 0$. Algorithm 6.11 shows the details.

LEMMA 6.11 *Algorithm 6.11 computes a simple cycle separator of the triangulation G. The size of the separator is at most $2h(T) - 1$, where $h(T)$ is the height of the BFS-tree T of G.*

PROOF. The correctness of Algorithm 6.11 follows from that of procedures COMPUTELCA, FINDGOODEDGE, and LABELTREECYCLE. Lemmas 6.14 and 6.15 state the correctness of procedures FINDGOODEDGE and LABELTREECYCLE.

For procedure COMPUTELCA, we use an algorithm by Chiang *et al.* [12]. This algorithm answers K LCA-queries on a tree T of size N with $O\left(\left(1 + \frac{K}{N}\right) sort(N)\right)$

FINDTREECYCLE(G, T):

Input: The vertex set $V(G)$ and the edge set $E(G)$ of the triangulation G. A BFS-tree T of G.

Output: A simple cycle separator S of G. That is, all vertices in S are labeled with $C(v) = 0$ in $V(G)$.

1: COMPUTEVERTEXLABELS(G, T)
{Vertex labels needed by procedure FINDGOODEDGE in Step 5.}

2: COMPUTELCA(G, T)
{Compute the lowest common ancestor u of v and w, for each edge $\{v, w\} \in E(G)$.}

3: Use COPYLABELS to copy all labels computed for u, v, w to edge $\{v, w\}$, for all edges in $E(G)$. $\{u = LCA(v, w)\}$

4: COMPUTEEDGELABELS(G, T)
{Compute additional edge labels.}

5: $e \leftarrow$ FINDGOODEDGE(G, T)
{Compute, for each non-tree edge e in $E(G)$, the weight of the region $R_1(e)$ inside $c(e)$ and the weight of the region $R_2(e)$ outside $c(e)$. Return an edge e with $w(R_1(e)) \leq \frac{2}{3}$ and $w(R_2(e)) \leq \frac{2}{3}$.}

6: LABELTREECYCLE(G, T, e)
{Label all vertices on $c(e)$ with $C(v) = 0$.}

Algorithm 6.11: Finding a simple cycle separator in a triangulation G.

I/Os. As there are $O(N)$ non-tree edges in G, $K = O(N)$. Thus, ComputeLCA takes $O(sort(N))$ I/Os.

By Lemmas 6.12, 6.2, 6.13, and 6.15, Steps 1, 3, 4, and 6 each take $O(sort(N))$ I/Os. Step 5 takes $O(scan(N))$ I/Os, by Lemma 6.14. Thus, Algorithm 6.11 takes $O(sort(N))$ I/Os.

The size of the separator follows from the definition of a tree cycle. Such a tree cycle consists of two paths, each of length at most h. These two paths share a vertex. Thus, this vertex is counted twice. That is, the length of the tree cycle is at most $2h - 1$. □

The vertex labels computed by procedure ComputeVertexLabels are: preorder numbers $n_p(v)$, subtree weights $W(v)$, as defined in Section 4.4, "weighted preorder numbers"

$$\nu_p(v) = \sum_{n_p(w) \leq n_p(v)} w(w), \text{ and}$$

"weighted distances to the root"

$$\delta(r, v) = \sum_{w \in A(v)} w(w),$$

where $A(v)$ is the set of v's ancestors, inclusive. Preorder numbers are defined to respect the embedding.

Algorithm 6.12 shows the pseudo-code that computes these labels.

Lemma 6.12 *Algorithm 6.12 takes $O(sort(N))$ I/Os.*

Proof. This follows immediately from Lemmas 6.7, 4.4, and 4.3 and Corollary 4.1. □

For the edges, we will need subtree weights $t(e)$ and certain prefix sums $t_p^v(e)$ of these weights. For a non-tree edge, we define $t(e) = 0$. For a tree edge $\{v, p(v)\}$, we define $t(e) = W(v)$.

Now, let v be a vertex of G. Let $e_0, \ldots, e_k$ be the edges incident on v, counterclockwise around v, starting at the reference edge e_0. Then, we define $t_p^v(e_0) = 0$. For

COMPUTEVERTEXLABELS(G, T):

Input: The vertex set $V(G)$ and the edge set $E(G)$ of the triangulation G. A BFS-tree T of G.

Output: Vertex labels $n_p(v)$, $\nu_p(v)$, $W(v)$, and $\delta(r, v)$.

1: $q \leftarrow$ ORDERCHILDREN(G, T)
2: $n_p \leftarrow$ COMPUTEPREORDERNUMBERS(G, T, q)
3: $W \leftarrow$ COMPUTESUBTREEWEIGHTS(G, T)
4: Sort $V(G)$ by increasing preorder numbers.
5: Scan $V(G)$ and compute the prefix sums $\nu_p(v) = \sum_{n_p(w) \leq n_p(v)} w(w)$.
6: Sort $V(G)$ by increasing distance to the root.
7: Use time-forward processing to compute labels $\delta(r, v)$:
$\delta(r, r) \leftarrow w(r)$, $\delta(r, v) \leftarrow \delta(r, p(v)) + w(v)$

Algorithm 6.12: Computing vertex labels.

$i > 0$, we define

$$t_p^v(e_i) = \sum_{j=1}^{i} t(e_j).$$

Intuitively, $t_p^v(e_i)$ is the weight of the subtrees of $T(v)$ that are to the left of e_i. ($T(v)$ is is defined as in Section 4.4.)

Algorithm 6.13 shows how to compute these labels.

LEMMA 6.13 *Algorithm 6.13 computes labels $t(e)$ and $t_p^v(e)$ with $O(sort(N))$ I/Os.*

PROOF. The labels $t(e)$ computed in Step 1 match the definition of these labels. In Steps 2 and 3, we create a collection E' of edge lists $E'(v)$, where $E'(v)$ contains all edges incident on v, sorted counterclockwise around v. Then the prefix sums computed in Step 4 match the definition of the labels $t_p^v(e)$. In Step 4, we have computed $t_p^v(e)$, for one copy of edge $e = \{v, w\}$ and $t_p^w(e)$, for another copy of e. In Step 5, we ensure that both copies are stored consecutively. Step 6 merges these copies into one. Thus, we have labels $t_p^v(e)$ and $t_p^w(e)$ stored with this single copy of edge e.

COMPUTEEDGELABELS(G, T):

Input: The vertex set $V(G)$ and the edge set $E(G)$ of the triangulation G. A BFS-tree T of G.

Output: Edge labels $t(e)$ and $t_p(e)$ of the edges in e in $E(G)$.

1: Scan $E(G)$ to compute $t(e)$:
  if e is a non-tree edge **then**
    $t(e) \leftarrow 0$
  else
    $\{e = \{v, w\}$ and $d(r, v) < d(r, w)\}$
    $t(e) \leftarrow W(w)$ $\{W(w)$ is already stored locally with e.$\}$
  end if

2: Scan $E(G)$ to create an edge list E':
  $\{e = \{v, w\}$ is the current edge.$\}$
  Make two copies e' and e'' of e in E'. Let $v(e') = v$, $n_v(e') = n_v(e)$, $v(e'') = w$, and $n_v(e'') = n_w(e)$.

3: Sort E' by increasing labels $v(e)$ and $n_v(e)$.

4: Scan E' to compute prefix sums $t_p^v(e)$:
  $\{e = \{v, w\}$ is the current edge. e' is the previous edge.$\}$
  if e is the first edge or $v(e) \neq v(e')$ **then**
    $t_p \leftarrow 0$
    $t_p(e) \leftarrow 0$
  else
    $t_p \leftarrow t_p + t(e)$
    $t_p(e) \leftarrow t_p$
  end if

5: Sort E' by increasing endpoints.
  $\{$For each edge $\{v, w\}$, $v < w$. Then $\{v, w\} < \{v', w'\}$ iff $v < v'$ or $v = v'$ and $w < w'$.$\}$

6: Scan E' and merge the two copies of each edge $e \in E(G)$:
  Let e' and e'' be the two copies. They are stored consecutively.
  Let $v(e') = v$ and $v(e'') = w$. Assign $t_p^v(e) = t_p(e')$ and $t_p^w(e) = t_p(e'')$.

Algorithm 6.13: Computing edge labels.

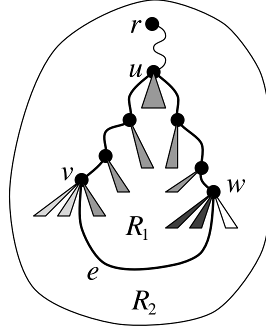

Figure 6.1: A non-tree edge e. The fat edges represent e's tree cycle. R_1 is the region inside the cycle. R_2 is the region outside the cycle.

Steps 3 and 5 take $O(sort(N))$ I/Os. Steps 1, 2, 4, and 6 take $O(scan(N))$ I/Os. Thus, Algorithm 6.13 takes $O(sort(N))$ I/Os. □

Next, we show how one can use the labels computed by procedures ComputeVertexLabels and ComputeEdgeLabels to find a non-tree edge e such that $c(e)$ is a separator of G.

Let $e = \{v, w\}$ be a non-tree edge. We define $R_1(e)$ to be the region inside e's tree cycle. $R_2(e)$ be the region outside e's tree cycle. We claim that, using the vertex and edge labels that we just computed, we can compute $w(R_1(e))$ and $w(R_2(e))$.

Let u be the lowest common ancestor of v and w. As T is a BFS-tree and e is non-tree edge, $v \neq u \neq w$: If, w.l.o.g., $u = v$, then v would have to be w's parent because v and w are only one level apart. Then, however, there would be a non-tree edge $\{v, w\}$ and a tree edge $\{v, w\}$. That is, G would not be a simple graph. Let v be to the left of w, i.e., $n_p(v) < n_p(w)$. This situation is shown in Figure 6.1.

As the preorder numbers $n_p(v)$ respect the embedding, the vertices x with preorder numbers $n_p(v) < n_p(x) \leq n_p(w)$ are exactly the vertices in the light and medium grey subtrees and on the tree path from u to w, excluding u. The weight of these vertices is $\nu_p(w) - \nu_p(v)$.

The weight of the vertices on the tree path from u to w, excluding u, is $\delta(r, w) - \delta(r, u)$. Thus, $\nu_p(w) - \nu_p(v) + \delta(r, u) - \delta(r, w)$ is the weight of the light and medium grey subtrees. To compute $w(R_1(e))$, we have to add the weight of the dark grey subtrees and subtract the weight of the light grey subtrees.

FINDGOODEDGE(G, T):

Input: The vertex set $V(G)$ and the edge set $E(G)$ of the triangulation G. A BFS-tree T of G.

Output: An edge e such that the the tree cycle $c(e)$ divides G into two regions of weight at most $\frac{2}{3}$ each.

1: Scan $E(G)$ to find edge e:
{e' is the current edge.}
  if e' is a non-tree edge **then**
    Compute $w(R_1(e))$ according to equation 6.1.
    Compute $w(R_2(e))$ according to equation 6.2.
    if $w(R_1(e)) \leq \frac{2}{3}$ and $w(R_2(e)) \leq \frac{2}{3}$ **then**
      $e \leftarrow e'$
      Stop scanning.
    end if
  end if

Algorithm 6.14: Finding an edge with a tree cycle that is a separator.

Now, the weight of the light grey subtrees is exactly $t_p^v(e)$ because $t(e) = 0$. Analogously, the weight of the dark grey subtrees is $t_p^w(e)$. Thus, the weight of $R_1(e)$ is

$$w(R_1(e)) = \nu_p(w) - \nu_p(v) + \delta(r, u) - \delta(r, w) + t_p^w(e) - t_p^v(e). \tag{6.1}$$

The weight of e's tree cycle is $\delta(r, v) + \delta(r, w) - 2\delta(r, u) + w(u)$. Thus, the weight of $R_2(e)$ is

$$w(R_2(e)) = w(G) - w(R_1(e)) - \delta(r, v) - \delta(r, w) + 2\delta(r, u) - w(u). \tag{6.2}$$

All these labels are stored locally with e. Thus, we can scan $E(G)$ and look for the first tree edge e with $w(R_1(e)) \leq \frac{2}{3}$ and $w(R_2(e)) \leq \frac{2}{3}$. We have shown the following lemma.

LEMMA 6.14 *Algorithm 6.14 computes a non-tree edge e with $w(R_1(e)) \leq \frac{2}{3}$ and $w(R_2(e)) \leq \frac{2}{3}$. This takes $O(scan(N))$ I/Os.*

Finally, we have to label the tree cycle $c(e)$, where e is the edge returned by procedure FINDGOODEDGE. Algorithm 6.15 shows the pseudo-code of procedure LABELTREECYCLE.

LABELTREECYCLE(G, T, e):

Input: The vertex set $V(G)$ and the edge set $E(G)$ of the triangulation G. A BFS-tree T of G. A non-tree edge $e = \{v, w\}$.

Output: The vertices on the tree cycle $c(e)$ labeled with $C(e) = 0$.

1: Sort $V(G)$ by increasing labels $d(r, v)$ and $n_p(v)$.

2: $d \leftarrow d(r, v)$

3: Scan $V(G)$ backward to label the vertices on the path from v to u:
{x is the current vertex. x' is the last vertex.}

```
    if d(r,x) = d and n_p(x) ≤ n_p(v) then
      C(x) ← 0
      d ← d − 1
    end if
    if x = u then
      Stop scanning.                                   {u = LCA(v,w)}
    end if
```

4: $d \leftarrow d(r, w)$

5: Scan $V(G)$ backward to label the vertices on the path from w to u:
{x is the current vertex.}

```
    if d(r,x) = d and n_p(x) ≤ n_p(w) then
      C(x) ← 0
      d ← d − 1
    end if
    if x = u then
      Stop scanning.                                   {u = LCA(v,w)}
    end if
```

Algorithm 6.15: Computing the tree cycle for a given edge e.

LEMMA 6.15 *Algorithm 6.15 labels the vertices on the tree cycle $c(e)$, for a given edge e with $C(v) = 0$. This takes $O(sort(N))$ I/Os.*

PROOF. The correctness of Algorithm 6.15 follows immediately, if we can prove that the ancestor of a vertex v at level d is the vertex x with $d(r, x) = d$ and $n_p(x) = \max\{n_p(y) : d(r, y) = d \wedge n_p(y) \leq n_p(v)\}$.

As x is an ancestor of v, $n_p(x) \leq n_p(v)$, by Observation 3.1. Now assume that there is a vertex y with $d(r, y) = d(r, x)$ and $n_p(x) < n_p(y) \leq n_p(v)$. As $n_p(x) < n_p(v)$, $d(r, x) < d(r, v)$. Thus, as $d(r, y) = d(r, x)$, $y \neq v$ and $n_p(y) < n_p(v)$. By Observation 3.2, x must be an ancestor of y. This is impossible because $d(r, y) = d(r, x)$. Thus, such a vertex y cannot exist. This proves the correctness of Algorithm 6.15.

Step 1 takes $O(sort(N))$ I/Os. Steps 3 and 5 take $O(scan(N))$ I/Os. Thus, the I/O-complexity of Algorithm 6.15 is $O(sort(N))$. □

6.5 Graph Separators and Shortest Paths

In this chapter, we presented a graph separator algorithm that followed the concepts of Lipton and Tarjan's approach. We followed their concepts even in that we wanted to compute *two* components A and B, each of total weight at most $\frac{2}{3}$. These two components are not necessarily connected.

When constructing the hierarchical decomposition described in Section 2.1, this is very inconvenient because not every vertex of a component has a path to every separator vertex on the boundary of this component. Hence, we want the components to be connected.

Looking at the code of Algorithm 6.1, we realize that, before constructing components A and B in Step 8, our algorithm did already identify all connected components of $G - C$. That is, in this step, we throw most of the information away. Sorting $V(G)$ and $E(G)$ in Step 8 instead of calling procedure COLLECTCOMPONENTS, we retain this information and produce the output needed for Algorithm 2.3. This change does not affect the running time of Algorithm 6.1.

Chapter 7

Triangulating Embedded Planar Graphs in External Memory

In this chapter, we present an external-memory algorithm for triangulating embedded planar graphs. This is the missing part of the separator algorithm in Chapter 6. Beside that, such an algorithm might be of independent interest.

We assume the same data representation as in Chapter 6. See section 6.1 for details.

7.1 Framework of the Algorithm

In this section, we present the framework of our triangulation algorithm. It divides the problem into two parts. In the first part, we identify the faces of the given graph G. We represent each face as a list of the vertices on its boundary, sorted clockwise around the face. In the second part, we use this information to triangulate the faces of G.

THEOREM 7.1 *Given an embedded connected planar graph G with N vertices, Algorithm 7.1 computes an embedded triangulation G' of G. This takes $O(sort(N))$ I/Os.*

TRIANGULATE(G):

Input: The vertex set $V(G)$ and the edge set $E(G)$ of the graph G.

Output: A triangulation G' represented as vertex set $V(G')$ and edge set $E(G')$. Edges in $E(G') \setminus E(G)$ are marked.

1: $F \leftarrow$ IDENTIFYFACES(G)
{F contains the vertex lists representing the faces of G. To distinguish different lists, vertices in the same list have the same label $f(v)$, while vertices in different lists have different labels. Each vertex in F is augmented with additional information. This information is used by TRIANGULATEFACES to embed the new edges properly.}

2: $G' \leftarrow$ TRIANGULATEFACES(G, F)
{Construct G'.}

Algorithm 7.1: Triangulating a embedded planar graph.

PROOF. The correctness of Algorithm 7.1 follows immediately from that of procedures IDENTIFYFACES and TRIANGULATEFACES.

By Lemmas 7.1 and 7.8, procedures IDENTIFYFACES and TRIANGULATEFACES take $O(sort(N))$ I/Os. Thus, procedure TRIANGULATE takes $O(sort(N))$ I/Os. □

The next two sections provide the details of procedures IDENTIFYFACES and TRIANGULATEFACES.

7.2 Identifying Faces

Procedure IDENTIFYFACES computes a list F that is the concatenation of several vertex lists, one for each face. These vertex lists will be used by TRIANGULATEFACES to compute the triangulation G'.

First, we define a graph $\hat{G} = (V(\hat{G}), E(\hat{G}))$. For each edge $\{v, w\} \in E(G)$, we add two vertices $v_{(v,w)}$ and $v_{(w,v)}$ to $V(\hat{G})$. Intuitively, $v_{(v,w)}$ represents the right side of $\{v, w\}$ when walking along $\{v, w\}$ from v to w; $v_{(w,v)}$ represents the other side of $\{v, w\}$. For each vertex $v \in V(G)$, let $\{v, w_0\}, \ldots, \{v, w_{k-1}\}$ be the list of edges incident on it, counterclockwise around v. Then we add directed edges

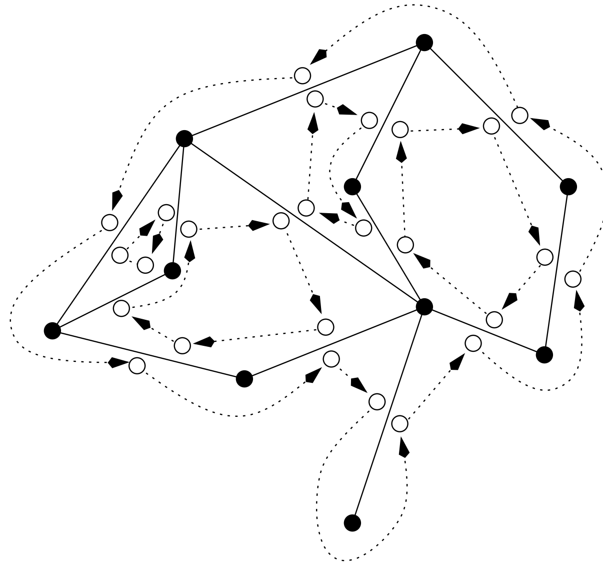

Figure 7.1: A given graph G (black vertices and solid lines). White vertices and dotted arrows represent the graph $\hat{G}$ for G.

$(v_{(w_i,v)}, v_{(v,w_{(i+1) \bmod k})}), 0 \leq i < k$ to $E(\hat{G})$. The resulting graph is shown in Figure 7.1. Now each face of G is represented by a cycle in $\hat{G}$. Our goal is to construct a vertex list, for each face of G, that represents its boundary. Algorithm 7.2 shows how.

LEMMA 7.1 *Algorithm 7.2 takes $O(sort(N))$ I/Os.*

PROOF. This follows immediately from Lemmas 7.2 and 7.3. □

LEMMA 7.2 *Algorithm 7.3 constructs $\hat{G}$ with $O(sort(N))$ I/Os.*

PROOF. Let us first prove the correctness of Algorithm 7.3. The creation of the vertex set $V(\hat{G})$ in Step 1 matches the definition of $\hat{G}$. Consider the construction of $E(\hat{G})$.

Steps 2 and 3 create an edge list $E'(v)$, for each vertex $v \in V$. The edges in such a list are sorted counterclockwise around v. E' is the concatenation of these edge lists.

In Step 4, we store the first edge of the current edge list in e''. Then we add edges between the vertices in $V(\hat{G})$ corresponding to consecutive edges in E'. When we reach the last edge e in the current list, e'' is the first edge in this list. We add an

IDENTIFYFACES(G):

Input: The vertex set $V(G)$ and the edge set $E(G)$ of the graph G.

Output: A vertex list F such that:

- Each face of G is represented as a sublist F_f of F. Each such sublist is contiguous. Its vertices have the same unique label $f(v)$ to distinguish them from the vertices of the other sublists.
- Let $F_f = v_0, \dots, v_k$ be a sublist representing a face f. Then $v_0, \dots, v_k$ are the vertices on the boundary of f. They appear in this order clockwise around f. v_i stores labels $n_1(v_i) = n_{v_i}(\{v_{(i-1) \bmod k}, v_i\})$ and $n_2(v_i) = n_{v_i}(\{v_i, v_{(i+1) \bmod k}\})$.

1: $\hat{G} \leftarrow$ CONSTRUCTFACEGRAPH(G)
2: $F \leftarrow$ BUILDF($\hat{G}$)

Algorithm 7.2: Identifying the faces of a given graph.

edge between the vertices in $V(\hat{G})$ corresponding to e'' and e. Recall that there are two vertices in $V(\hat{G})$ corresponding to each edge. We always add edges between the right vertices.

Steps 1, 2, and 4 scan lists of length $O(N)$. Thus, these steps take $O(\mathit{sort}(N))$ I/Os. In Step 3, we sort a list of length $O(N)$. Thus, Algorithm 7.3 takes $O(\mathit{sort}(N))$ I/Os. □

LEMMA 7.3 *Algorithm 7.4 builds the list F with $O(\mathit{sort}(N))$ I/Os.*

PROOF. First, we show the correctness of Algorithm 7.4. In Step 1, we use the algorithm by Chiang *et al.* [12]. (In this step, we consider the edges of $\hat{G}$ as undirected.) Then we copy, for each edge, all information stored with its source vertex to the edge itself. To this end, we use procedure COPYLABELS. Lemma 6.2 states the correctness of this procedure. Sorting the edges by increasing face labels sorts all edges in $E(\hat{G})$ by the faces of G containing them. At the end of Step 2, we basically remove exactly one edge per face from $E(\hat{G})$. Now, each edge represents a vertex with a pointer to its parent. The root points to **null**. Thus, we can use Corollary 4.2 to rank all lists

CONSTRUCTFACEGRAPH(G):

Input: The vertex set $V(G)$ and the edge set $E(G)$ of the graph G.

Output: A directed graph $\hat{G}$ with

- two vertices $v_{(v,w)}$ and $v_{(w,v)}$ for each edge $\{v, w\}$ in $E(G)$ and
- an edge set $E(v)$ in $E(\hat{G})$, for each vertex $v \in V(G)$. For a particular vertex v, let $\{v, w_0\}, \ldots, \{v, w_{k-1}\}$ be the edges in G incident on v. $\{v, w_{(i+1) \bmod k}\}$ is the next edge counterclockwise around v after $\{v, w_i\}$. Then $E(v)$ contains edges $(v_{(w_i,v)}, v_{(v,w_{(i+1) \bmod k})})$, for $0 \leq i < k$.

1: Scan $E(G)$ and add two vertices $v_{(v,w)}$ and $v_{(w,v)}$ to $V(\hat{G})$, for each edge $\{v, w\}$ in $E(G)$. Store all information about $\{v, w\}$ with these vertices.

2: Scan $E(G)$ and create two copies e' and e'' of each edge $e = \{v, w\}$. Label e' with $v(e') = v$ and $n_v(e') = n_v(e)$ and e'' with $v(e'') = w$ and $n_v(e'') = n_w(e)$. Call the resulting edge set E'.

3: Sort E' by increasing labels v and n_v.

4: Scan E' to construct the edge set of $\hat{G}$:
$\{e$ is the current edge. e' is the previous edge.$\}$

```
if e is the first edge then
  e'' ← e
else if v(e') ≠ v(e) then
  {e'' = {v, w''}, e' = {v, w'}, and v(e') = v(e'') = v.}
  Add an edge (v_(w',v), v_(v,w'')) to E(Ĝ).
  e'' ← e
else
  {e = {v, w}, e' = {v, w'} and v(e) = v(e') = v.}
  Add an edge (v_(w',v), v_(v,w)) to E(Ĝ).
end if
if e is the last edge then
  Let e'' = {v, w''} and v(e'') = v. Then add an edge (v_(w,v), v_(v,w'')) to E(Ĝ).
end if
```

Algorithm 7.3: Constructing $\hat{G}$.

BUILDF($\hat{G}$):

Input: The vertex set $V(\hat{G})$ and the edge set $E(\hat{G})$ of the graph $\hat{G}$.

Output: A vertex list F such that:

- Each face f of G is represented as a sublist F_f of F. Each such sublist is contiguous. Its vertices have the same unique label $f(v)$ to distinguish them from the vertices in the other sublists.
- Let $F_f = v_0, \ldots, v_k$ be a sublist representing a face f. Then v_i stores labels $n_1(v_i) = n_{v_i}(\{v_{(i-1) \bmod k}, v_i\})$ and $n_2(v_i) = n_{v_i}(\{v_i, v_{(i+1) \bmod k}\})$.

1: Compute the connected components of $\hat{G}$.
{This labels the vertices v of $\hat{G}$ with labels $f(v)$ that identify the connected components in which they are contained.}

2: Make each circular component of $\hat{G}$ a list by removing one edge:
For each edge $e = (v, w)$ in $E(\hat{G})$, copy all information from v to e, using procedure COPY-LABELS.
Sort the edge set $E(\hat{G})$ by increasing labels $f(e)$.
Scan $E(\hat{G})$. If e is the first edge or the predecessor e' of the current edge e has component label $f(e') \neq f(e)$, change the target vertex of e to **null**.
{Now we can consider $E(\hat{G})$ as a forest of rooted trees. Every edge has been transformed into a vertex with pointer to its parent. The root points to **null**.}

3: Apply Corollary 4.2 to rank all lists in $E(\hat{G})$. Call the rank of e in its list $r(e)$.

4: Sort $E(\hat{G})$ by increasing labels $f(e)$ and decreasing ranks $r(e)$.

5: Scan $E(\hat{G})$ and add, for each edge $(v_{(v,w)}, x)$, the two vertices v and w to a list F'. Label v with $n_2(v) = n_v(\{v, w\})$ and w with $n_1(w) = n_w(\{v, w\})$. Let $f(v) = f(w) = f((v_{(v,w)}, x))$.

6: Scan F' to build F:
{v is the current vertex. w is the previous vertex.}
if v is the first vertex **then**
  $v' \leftarrow v$
else if $f(v) \neq f(w)$ **then**
  Let $n_2(w) = n_2(v')$. Add w to F.
  $v' \leftarrow v$
else if v is the last vertex **then**
  Let $n_2(v) = n_2(v')$. Add v to F.
else if $v = w$ **then**
  Let $n_2(w) = n_2(v)$. Add w to F.
end if

Algorithm 7.4: Constructing the vertex list F.

in $E(\hat{G})$. We sort $E(\hat{G})$ by increasing labels $f(e)$ and decreasing ranks $r(e)$. For the face in Figure 7.2(a), assume that we have removed the edge between $(v_{(2,1),(1,2)})$. The resulting sublist in $E(\hat{G})$ after Step 4 is

$$v_{(1,2)}, v_{(2,3)}, v_{(3,4)}, v_{(4,5)}, v_{(5,6)}, v_{(6,7)}, v_{(7,8)}, v_{(8,10)}, v_{(10,11)}, v_{(11,12)}, v_{(12,13)}, v_{(13,14)}, \\ v_{(14,15)}, v_{(15,14)}, v_{(14,16)}, v_{(16,14)}, v_{(14,13)}, v_{(13,17)}, v_{(17,10)}, v_{(10,8)}, v_{(8,9)}, v_{(9,2)}, v_{(2,1)}.$$

The list F' is a concatenation of vertex lists F'_f, one per face f of G. Let f be a face with boundary $v_0, \dots, v_{k-1}$. Then, F'_f is

$$(v_0, n_2(v_0)), (v_1, n_1(v_1)), (v_1, n_2(v_1)), (v_2, n_1(v_2)), \dots, \\ (v_{k-1}, n_1(v_{k-1})), (v_{k-1}, n_2(v_{k-1})), (v_0, n_1(v_0)).$$

Step 6 now creates a list

$$(v_1, n_1(v_1), n_2(v_1)), (v_2, n_1(v_2), n_2(v_2)), \dots, \\ (v_{k-1}, n_1(v_{k-1}), n_2(v_{k-1})), (v_0, n_1(v_0), n_2(v_0))$$

from F'_f. This is the desired list F_f. As we do this for each face, Algorithm 7.4 produces F.

Step 1 take $O(sort(N))$ I/Os as proved in [12]. Step 2 takes $O(sort(N))$ I/Os by Lemma 6.2. Step 3 takes $O(sort(N))$ I/Os by Corollary 4.2. Step 4 takes $O(sort(N))$ I/Os. Steps 5 and 6 scan two lists of length $O(N)$. Thus, they take $O(scan(N))$ I/Os. Hence, the I/O-complexity of Algorithm 7.4 is $O(sort(N))$. □

7.3 Triangulating Faces

In this section, we show how to triangulate the faces of G using the information stored in F. First, we describe our method and show that we do not add multiple edges. Then, we deal with the problem of embedding the new edges properly. Eventually, we show how these methods can be implemented I/O-efficiently.

Consider a face f of G and its sublist F_f in G. We triangulate f in three steps. In the first step, we reduce f to a simple face $\tilde{f}$. We call a face *simple*, if each vertex

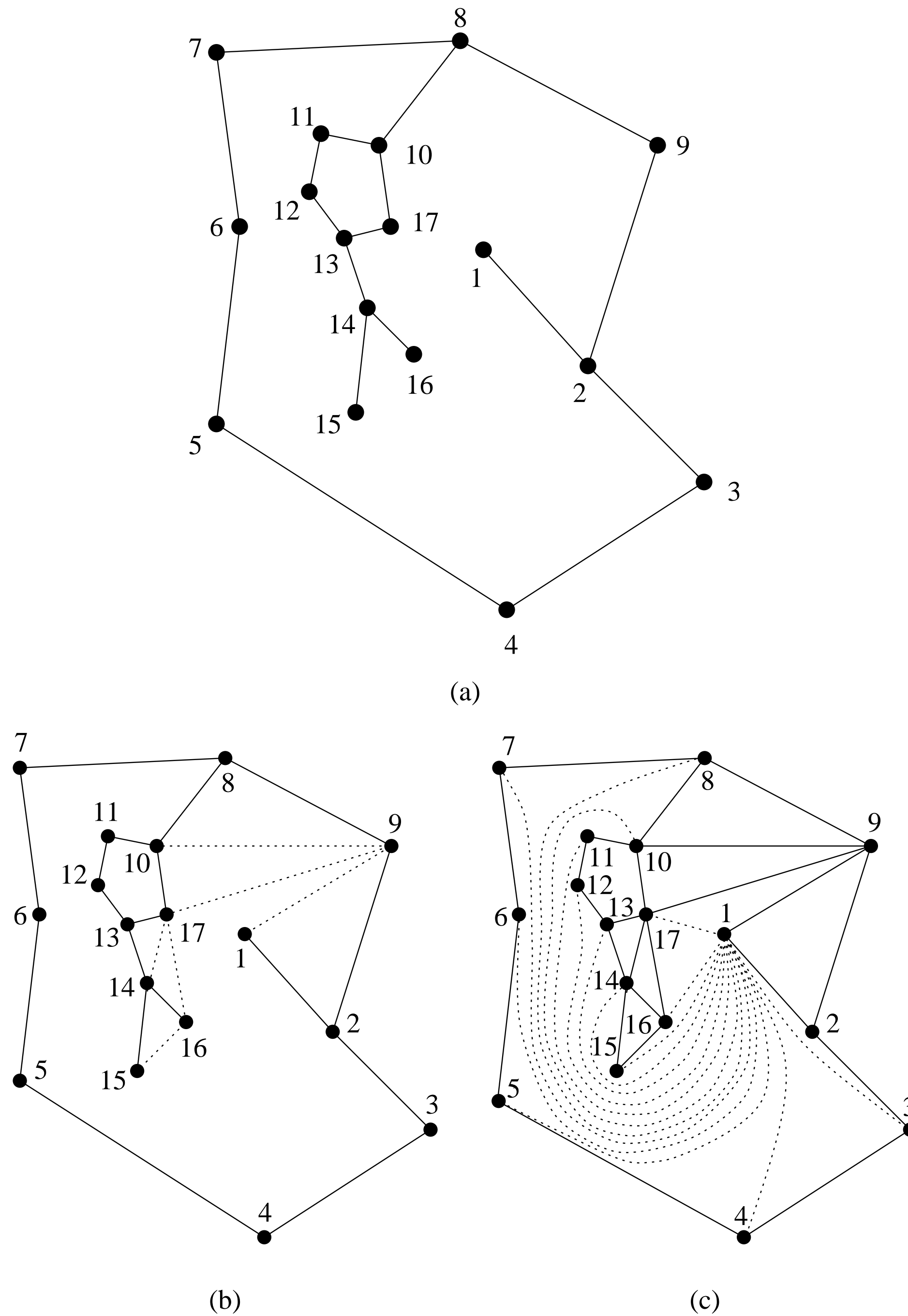

Figure 7.2: Figure (a) shows a face of the given graph. The dotted edge in figure (b) are added in the first pass of the algorithm. The second pass adds the dotted edges in figure (c). This triangulates the face.

on its boundary is visited only once in a clockwise traversal of the boundary. F_f will be reduced to $F_{\tilde{f}}$ accordingly. In the second step, we triangulate $\tilde{f}$. This will not add multiple edges in $\tilde{f}$. However, in adjacent faces $\tilde{f}$ and $\tilde{f}'$, we may have added diagonals with the same endpoints. Then we retriangulate $\tilde{f}'$ in a way that removes multiple edges and does not introduce new multiple edges.

The first step marks the first appearance of each vertex v in F_f, while it leaves all later appearances of v unmarked. Moreover, it appends a marked copy of the first vertex in F_f to the end of F_f. For the face in Figure 7.2(a), this results in the following list. Marked elements are printed in bold.

$$\mathbf{1}, \mathbf{2}, \mathbf{3}, \mathbf{4}, \mathbf{5}, \mathbf{6}, \mathbf{7}, \mathbf{8}, \mathbf{10}, \mathbf{11}, \mathbf{12}, \mathbf{13}, \mathbf{14}, \mathbf{15}, 14, \mathbf{16}, 14, 13, \mathbf{17}, 10, 8, \mathbf{9}, 2, \mathbf{1}.$$

We scan this list backward and remove each unmarked vertex by adding an edge between its predecessor and its successor in the current list. Call this procedure MakeSimpleFace. For the face in Figure 7.2(a), the result is shown in Figure 7.2(b). We prove the following lemma.

Lemma 7.4 *Given the list F_f of vertices visited when traversing the boundary of a face f once in clockwise order. Procedure* MakeSimpleFace *creates a simple face $\tilde{f}$ by cutting triangles off f.*

Proof. The way we mark the vertices in F_f, each vertex on f's boundary has exactly one marked copy in F_f. The first vertex has two marked copies in F_f; but this is only to close the cycle. For each unmarked copy of a vertex v, we add an edge between its predecessor and its successor in F_f. This removes this copy of v from F_f and the resulting face f'. We remove all unmarked copies from F_f this way. Thus, the final list $F_{\tilde{f}}$ contains each vertex on the boundary of f only once, except for the first vertex in $F_{\tilde{f}}$, which appears a second time at the end of $F_{\tilde{f}}$. That is, the final face $\tilde{f}$ is simple.

Each time we cut a vertex v off, we add an edge between its predecessor and its successor in the current list. Thus, the cut-off region is a triangle. □

Lemma 7.5 *Applying procedure* MakeSimpleFace *to all faces of G does not add multiple edges to G.*

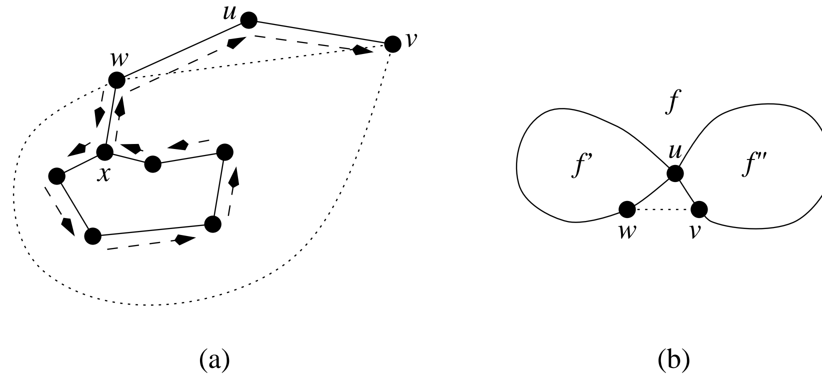

Figure 7.3: Illustrating why MAKESIMPLEFACE does not add multiple edges.

PROOF. First assume that MAKESIMPLEFACE adds multiple edges in a single face f. Let v and w be the endpoints of two such edges (see Figure 7.3(a)). The first edge removes u from the boundary of f. The second edge removes x from the boundary of f. However, these two edges form a closed curve that separates x from the rest of the face. Thus, the copy of x removed by adding the second edge must be x's first occurrence on the boundary. That is, it is marked and would not be removed. This contradiction proves that MAKESIMPLEFACE does not add multiple edges in a single face.

It remains to prove that MAKESIMPLEFACE does not add multiple edges in different faces. Consider Figure 7.3(b). There we remove an unmarked copy of the vertex u from the boundary of f by adding an edge $\{v, w\}$. As this copy of u is unmarked, there must be at least one more copy of u: the marked copy. The part of the boundary of f between the removed copy and the marked copy of u is a closed curve. Call the enclosed region f'. Analogously, the part of the boundary of f between the marked and the removed copy of u is a closed curve. Let it enclose region f''. Any face other than f that has w on its boundary must be contained in f'. Any face other than f that has v on its boundary must be contained in f''. However, there is no face that is in f' *and* in f''. Thus, there can be no face other than f that has v *and* w on its boundary. That is, we cannot add an edge $\{v, w\}$ in any other face than f. □

After the execution of procedure MAKESIMPLEFACE, we are left with a simple face $\tilde{f}$. $F_{\tilde{f}}$ is the list of boundary vertices of $\tilde{f}$. It contains a copy of the first vertex

v at the end. We remove the first two vertices and the last two vertices from $F_{\tilde{f}}$. Now we add an edge between v and each of the remaining vertices in $F_{\tilde{f}}$. Call this procedure TRIANGULATESIMPLEFACE. For our example face in Figure 7.2(b), $F_{\tilde{f}}$ is

$$1, 2, 3, 4, 5, 6, 7, 8, 10, 11, 12, 13, 14, 15, 16, 17, 9, 1.$$

We remove the first two vertices and the last two vertices:

$$3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17.$$

Then we add edges between vertex 1 and all vertices in $F_{\tilde{f}}$. The result is shown in Figure 7.2(c).

LEMMA 7.6 *Given a simple face $\tilde{f}$, procedure* TRIANGULATESIMPLEFACE *triangulates $\tilde{f}$.*

As we deal with all faces of G independently, procedure TRIANGULATESIMPLE-FACE may add the same diagonal $\{v, w\}$ to two different faces $\tilde{f}$ and $\tilde{f}'$ that have v and w on their boundaries. It can also happen that we add a diagonal $\{v, w\}$ to a face, and $\{v, w\}$ exists already in G. More generally, let $\tilde{f}_1, \ldots, \tilde{f}_k$ be the faces that contain a diagonal $\{v, w\}$. We have to remove $\{v, w\}$ from $k - 1$ of these faces, if $\{v, w\} \notin E(G)$, or from all of them, if $\{v, w\} \in E(G)$. In the first case, we mark $\{v, w\}$ as a *conflicting diagonal* in $\tilde{f}_2, \ldots, \tilde{f}_k$. In the second case, we mark it in all faces $\tilde{f}_i$. We call a face *conflicting*, if it has a conflicting diagonal in it. Otherwise, we call it *conflict-free*.

Every diagonal d in a face $\tilde{f}$ cuts $\tilde{f}$ into two parts. Let $v_1, \ldots, v_k$ be the vertices on the boundary of $\tilde{f}$. Let $d = \{v_i, v_j\}$. Then any other diagonal in $\tilde{f}$ has both its endpoints in one of the sets $V_1(d) = \{v_i, \ldots, v_j\}$ or $V_2(d) = \{v_j, \ldots, v_k, v_1, \ldots, v_i\}$. Let $D_1(d)$ be the set of diagonals with endpoints in $V_1(d)$; $D_2(d)$ contains the diagonals with endpoints in $V_2(d)$. We call a conflicting diagonal *maximal*, if $D_1(d) = \emptyset$ or $D_2(d) = \emptyset$. It is easy to see that each conflicting face contains a maximal conflicting diagonal.

Let $\tilde{f}$ be a conflicting face. We show how to make it conflict-free. Choose a maximal conflicting diagonal d in $\tilde{f}$. Let d' be the edge with which d conflicts. d cuts

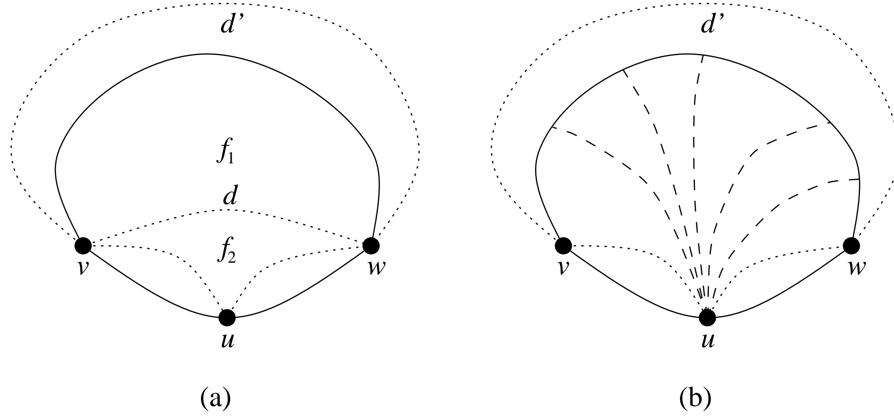

Figure 7.4: (a) A simple face $\tilde{f}$ with a conflicting diagonal d. d conflicts with d' and divides f into two parts f_1 and f_2. One of them, f_2, is conflict-free. u is the third vertex of the triangle in f_2 with d on its boundary. (b) The triangulation of $\tilde{f}$ produced by MakeConflictFree.

$\tilde{f}$ into two parts f_1 and f_2; f_2 be the part that does not contain conflicting diagonals. Let u be the third vertex of the triangle in f_2 that has $\{v, w\}$ on its boundary. Remove d and all diagonals in f_1 from $\tilde{f}$. Retriangulate the resulting subface of $\tilde{f}$ by adding diagonals between u and the boundary vertices of f_1, excluding v and w. See Figure 7.4. Call this procedure MakeConflictFree.

Lemma 7.7 *Procedure* MakeConflictFree *makes $\tilde{f}$ conflict-free.*

Proof. d and d' form a closed curve C. All boundary vertices of f_1, excluding v and w, are inside C. All boundary vertices of f_2, excluding v and w, are outside C. In particular, u is outside C. Thus, before retriangulating $\tilde{f}$, there can be no edge $\{u, z\}$ in the triangulation, where z is a vertex on the boundary of f_1. That is, all new diagonals are non-conflicting. We have only retained diagonals in f_2. By the choice of d and f_2, all diagonals in f_2 are non-conflicting. Thus, none of the diagonals in $\tilde{f}$ conflicts with any other diagonal in the triangulation. □

Lemma 7.7 implies that, after applying procedure MakeConflictFree to all conflicting faces of G, we obtain a proper triangulation of G.

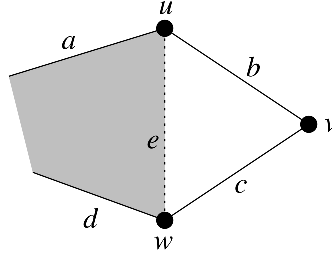

Figure 7.5: Adding a diagonal $e = \{u, w\}$ to a face.

What we have ignored so far is the problem of embedding diagonals properly. Assume that we want to add a diagonal $e = \{u, w\}$ as in Figure 7.5. Here, $n_1(u) = n_u(a)$, $n_2(u) = n_u(b)$, $n_1(w) = n_w(c)$, and $n_2(w) = n_w(d)$. To embed e between a and b, we have to ensure that $n_u(a) < n_u(e) < n_u(b)$, if $n_u(a) < n_u(b)$, or $n_u(a) < n_u(e)$, otherwise. We ensure this by choosing $n_u(e) = \frac{n_2(u)+n_1(u)}{2}$, if $n_1(u) < n_2(u)$, or $n_u(e) = n_2(u) + 1$, otherwise. For w, the procedure is analogous. Then we have to update $n_2(u) = n_u(e)$ and $n_1(w) = n_w(e)$. This ensures that subsequent edges added to the grey face are embedded properly.

Now we are ready to give the pseudo-code of procedure TriangulateFaces. It is shown in Algorithm 7.5.

Lemma 7.8 *Algorithm 7.5 constructs a triangulation G' of G with $O(sort(N))$ I/Os.*

Proof. The correctness of Algorithm 7.5 follows from the discussion at the beginning of this section and from the correctness of procedures MakeSimpleFaces, TriangulateSimpleFaces, FindConflictingDiagonals, and RetriangulateFaces. Lemmas 7.9, 7.10, 7.11, and 7.12 state the correctness of these procedures.

By the same lemmas, all procedures take $O(sort(N))$ or $O(scan(N))$ I/Os. Thus, Algorithm 7.5 takes $O(sort(N))$ I/Os. □

Lemma 7.9 *Algorithm 7.6 makes all faces simple and takes $O(sort(N))$ I/Os.*

Proof. The correctness of Algorithm 7.6 follows from Lemmas 7.4 and 7.5.

Step 1 involves two sorts and one scan of the list F. Steps 2 and 3 just scan F. Thus, the I/O-complexity of Algorithm 7.6 is $O(sort(N))$. □

TRIANGULATEFACES(G, F):

Input: The vertex set $V(G)$ and the edge set $E(G)$ of the graph G. The vertex list F as produced by IDENTIFYFACES.

Output: The vertex and edge sets of a triangulation G' of G.

1: $G'' \leftarrow$ MAKESIMPLEFACES(G, F)
{Make each face of G simple.}

2: $D \leftarrow$ TRIANGULATESIMPLEFACES(G'', F)
{D is a list of diagonals that would be added to G'' to triangulate the faces of G''. Each diagonal d in D is labeled with a label $f(d)$, where $f(d)$ is the face of G'' that contains d.}

3: FINDCONFLICTINGDIAGONALS($D, E(G'')$)
{This finds a maximal conflicting diagonal for each conflicting face. These diagonals are marked in D.}

4: $G' \leftarrow$ RETRIANGULATEFACES(G'', F, D)
{Construct the final triangulation G' of G.}

Algorithm 7.5: Triangulating faces.

LEMMA 7.10 *Algorithm 7.7 triangulates the faces of G''. This takes $O(scan(N))$ I/Os.*

PROOF. The correctness of Algorithm 7.7 follows from Lemma 7.6. Algorithm 7.7 scans the list F only once to triangulate the faces of G''. Thus, its I/O-complexity is $O(scan(N))$. □

LEMMA 7.11 *Algorithm 7.8 computes the maximal conflicting diagonals of all faces in G'' with $O(sort(N))$ I/Os.*

PROOF. Let us first prove the correctness of Algorithm 7.8. Step 1 merges D and $E(G'')$. That is, D is the edge set of the current triangulation of G'' now. Step 2 sorts D such that edges with the same endpoints are stored consecutively. Let $d_1, \ldots, d_k$ be a list of such edges with endpoints v and w. Step 3 marks $d_2, \ldots, d_k$, but not d_1. If $\{v, w\} \notin E(G'')$, $d_1, \ldots, d_k$ are all diagonals. That is, Step 3 marks all diagonals, except for one. If $\{v, w\} \in E(G'')$, then d_1 is this edge in $E(G'')$ by the order of D.

MAKESIMPLEFACES(G, F):

Input: The vertex set $V(G)$ and the edge set $E(G)$ of the graph G. The vertex list F as produced by IDENTIFYFACES.

Output: A new graph G'' that has only simple faces. A modified list F representing all simple faces that still have to be triangulated.

1: Mark all first occurrences of vertices in the sublists F_f of F.
   Sort the vertices in F by increasing face labels, increasing vertex number, and increasing original position in F. With each vertex in F, store its position before the sort.
   {The tertiary key ensures that the sort is stable.}
   Scan F. Let v be the current vertex and w be the previous vertex. Mark v, if $f(v) \neq f(w)$ or $v \neq w$.
   Sort the vertices in F by their stored position in F before the last sort.
   {This moves every vertex back to its original position in F.}

2: Scan F and append, for each sublist F_f, a copy of the first vertex in F_f to the end of F_f.

3: Scan F backward to add diagonals:
   {v is the current vertex. u appears before v in F. w appears after v in F.}
   if v is not marked **then**
     Add an edge $e = \{u, w\}$ to G.
     Compute $n_u(e)$, $n_w(e)$, $n_2(u)$, and $n_1(w)$ as described on page 117.
     Remove v from F.
   end if

Algorithm 7.6: Making all faces of G simple.

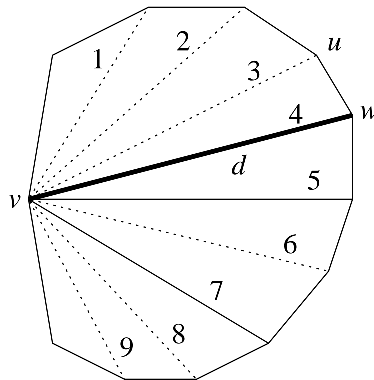

Figure 7.6: Solid edges represent the face boundary and conflicting diagonals. Dotted edges are non-conflicting. The fat edge is the first conflicting edge when traversing the list of diagonals bottom-up. Therfore, it is maximal.

TRIANGULATESIMPLEFACES(G'', F):

Input: The vertex set $V(G'')$ and the edge set $E(G'')$ of the graph G. The vertex list F as produced by IDENTIFYFACES.

Output: A list of diagonals D that would be added to G'' to triangulate it. Each diagonal d in D has a label $f(d)$ that identifies the face of G'' containing d. D is sorted by face labels. The sublist for a face f contains the diagonals added to f in their insertion order.

1: Scan F to triangulate the faces of G'':
{v is the current vertex. w is the previous vertex.}
  if v is the first vertex or $f(v) \neq f(w)$ **then**
    $x \leftarrow v$
    Skip the next vertex in F.
  else if v is the second last vertex in F **then**
    Skip the next vertex in F.
  else
    Add a diagonal $d = \{x, v\}$ with label $f(d) = f(v)$ at the end of D.
  end if

Algorithm 7.7: Triangulating simple faces.

$d_2, \ldots, d_k$ are diagonals. That is, Step 3 marks all diagonals, but not $d_1 \in E(G'')$. Step 4 removes the edges in $E(G'')$ from D. Obviously, Step 6 marks exactly one conflicting diagonal per conflicting face. Why is this diagonal maximal?

Consider a face f. The diagonals added by MAKESIMPLEFACES all have one endpoint in common. They are stored in f in the order shown in Figure 7.6. Step 6 traverses these diagonals in their order of appearance and marks the first conflicting diagonal d as maximal (fat). As this is the first conflicting diagonal, the part of f above d does not contain any conflicting diagonals. Thus, d is maximal.

Steps 2 and 5 take $O(sort(N))$ I/Os. Steps 1, 3, 4, and 6 each scan D once. This takes $O(scan(N))$ I/Os. Thus, Algorithm 7.8 takes $O(sort(N))$ I/Os. □

LEMMA 7.12 *Algorithm 7.9 triangulates G'' with $O(scan(N))$ I/Os. The resulting triangulation G' does not contain multiple edges.*

PROOF. Let us first prove that Algorithm 7.9 triangulates each face of G''. Assume that the current vertex v in F is the first vertex of a sublist F_f for a face f of G''; d is

FINDCONFLICTINGDIAGONALS($D, E(G'')$):

Input: A list of diagonals D that would be added to G'' to triangulate it. Each diagonal d in D has a label $f(d)$ that identifies the face of G'' containing d. D is sorted by face labels. The sublist for a face f contains the diagonals added to f in their insertion order. The edge set $E(G'')$ of the graph G''.

Output: Maximal conflicting diagonals in D marked as such.

1: Scan D and $E(G'')$ to append $E(G'')$ to D. Mark all edges in D as diagonals. With each diagonal, store its original position in D.

2: Sort D by increasing endpoints and such that edges $\{v, w\}$ in $E(G'')$ are stored before diagonals $\{v, w\}$.
$\{$For each edge $d = \{v, w\}$, $v < w$. Then, $d = \{v, w\} < d' = \{v', w'\}$ iff $v < v'$ or $v = v'$ and $w < w'$ or $v = v'$, $w = w'$, d is an edge in $E(G'')$, and d' is a diagonal.$\}$

3: Scan D to mark conflicting diagonals:
$\{d$ is the current edge. d' is the previous edge.$\}$
if $d = \{v, w\}$, $d' = \{v', w'\}$, $v = v'$ and $w = w'$ **then**
  Mark d as conflicting.
end if

4: Scan D to remove all edges in $E(G'')$ from D.

5: Restore D's original order.
$\{$This can be done by sorting D by the positions saved in Step 1.$\}$

6: Scan D to mark maximal conflicting diagonals:
$\{d$ is the current diagonal. d' is the previous diagonal.$\}$
if d is the first diagonal in D or $f(d') \neq f(d)$. **then**
  found_maximal ← **false**
end if
if d is conflicting and found_maximal = **false then**
  Mark d as maximal.
  found_maximal ← **true**
end if

Algorithm 7.8: Computing maximal conflicting diagonals.

RETRIANGULATEFACES(G'', F, D):

Input: The embedded graph G'' represented as its vertex and edge set. The vertex list F representing the boundaries of the faces of F. The list of diagonals added to G'' by procedure TRIANGULATESIMPLEFACES. Maximal conflicting diagonals in D are marked as such in D.

Output: An embedded triangulation G' of G''.

1: Scan F and D simultaneously to triangulate G'':
{v is the current vertex in F. v' is the previous vertex. d is the current diagonal in D. d' is the previous diagonal. The scan is driven by D.}

```
if d is the first diagonal or f(d) ≠ f(d') then
  x ← v
  conflicting ← false          {We assume that the current face is conflict-free.}
  Advance by two positions in F.
end if
if d is a maximal conflicting diagonal then
  x ← v'
  conflicting ← true           {The current face is conflicting.}
  Advance by one position in F.
end if
Add edge {x, v} to G''.
Compute n_u(e), n_w(e), n_2(u), and n_1(w) as described on page 117.
if d is the last diagonal in D with label f(d) then
  if conflicting then
    Advance by one position in F.
  else
    Advance by two positions in F.
  end if
end if
```

2: Return G''.

Algorithm 7.9: Computing the final triangulation.

the first diagonal in D that belongs to the triangulation of f. Let $v = v_0, v_1, v_2, \ldots, v_k$ be the vertices in F_f. $d_0, \ldots, d_{k-4}$ are the diagonals in f.

First, we set $x \leftarrow v_0$ and advance to v_2. If f does not contain a conflicting diagonal, we add diagonals $d_0 = \{v_0, v_2\}, d_1 = \{v_0, v_3\}, \ldots, d_{k-4} = \{v_0, v_{k-2}\}$ to G''. After adding d_{k-4}, which is the last diagonal of this face, we advance by two positions in F. That is, we skip v_{k-1} and v_k. Now v points to the first vertex of the next face, and d points to the first diagonal of that face.

If f contains a conflicting diagonal, let d_q be the marked maximal conflicting diagonal. We add diagonals $d_0 = \{v_0, v_2\}, d_1 = \{v_0, v_3\}, \ldots, d_{q-1} = \{v_0, v_{q+1}\}$ to G''. When we find d_q, the current vertex is v_{q+2}. We advance by one position in F and add diagonals $d'_q = \{v_{q+1}, v_{q+3}\}, d'_{q+1} = \{v_{q+1}, v_{q+3}\}, \ldots, d'_{k-4} = \{v_{q+1}, v_{k-1}\}$ (one new diagonal per old diagonal). Then, we advance by one position in F, i.e., skip v_k. Again, v points to the first vertex of the next face, and d points to the first diagonal of the next face.

It is easy to see that the added diagonals triangulate f. It remains to prove that the resulting triangulation G' does not contain multiple edges. If a face is conflict-free, our algorithm adds the same diagonals as procedure TRIANGULATESIMPLEFACES. Thus, this face remains conflict-free. If the face is conflicting, we argue as follows: In Figure 7.6, $v = v_0$, $w = v_{q+2}$, and $u = v_{q+1}$. On the other hand, u is the third vertex of the triangle uvw in the conflict-free part of f. That is, by making v_{q+1} the common endpoint of all diagonals d'_i, we implement exactly the method of procedure MAKECONFLICTFREE. By Lemma 7.7, this makes face f conflict-free in G'.

Algorithm 7.9 scans D and F once. As the size of these lists is $O(N)$, this takes $O(scan(N))$. □

Chapter 8

Conclusions

In this work, we presented an external-memory shortest path data structure and an I/O-efficient algorithm to construct it.

In order to allow for I/O-efficient queries of this data structure, we developed a method to store rooted trees in external memory such that bottom-up paths can be traversed I/O-efficiently.

To make the preprocessing algorithm efficient, we needed three major ingredients: (1) a separator algorithm, (2) a single source shortest path algorithm, and (3) an algorithm to block the shortest path trees in our data structure for fast path traversals.

For point (2), we used the currently best known single source shortest path algorithm due to Crauser, Mehlhorn, and Meyer [13]. We presented an I/O-efficient algorithm for point (3). We showed that graph separation is not harder than the more fundamental problem of constructing a BFS-tree of a given planar graph. As part of this proof, we developed an external-memory algorithm for triangulating embedded planar graphs.

Unfortunately, BFS turns out to be a really hard problem in external-memory. It is not clear whether it can be done I/O-efficiently at all. Our future work will therefore concentrate on the following three observations:

1. Given an embedded graph G, we can construct a triangulation G' of G. Given a separator of the dual of G', we can add the three boundary vertices of each face in this separator to a set S. S is a separator of G. The dual of G' is a

3-regular graph, i.e., extremely sparse and regular. Hopefully, BFS is easier in such graphs.

2. In application such as geographic information systems, the graphs that we are dealing with are in fact triangulated point sets. It might help to use the geometric information provided this way to compute a BFS-tree of the given graph. For instance, one can compute a monotone spanning tree of the graph with $O(scan(N))$ I/Os using the technique in [32, 33]. Then one could try to patch this tree into a BFS tree.

3. Lipton and Tarjan's separator algorithm requires a BFS-tree to ensure that removing a complete level from this tree indeed separates the upper part of the tree from the lower part. We can define a "relaxed BFS-tree" of a graph G as a spanning tree such that each level of k vertices is spanned by at most $\max\{k, \sqrt{2}\sqrt{N}\}$ non-tree edges. When removing this level from the tree, we also remove one vertex per edge spanning this level from the graph. This guarantees that removing these vertices from the tree separates the upper part from the lower part. The size of the separator is at most $4\sqrt{2}\sqrt{N} = O(\sqrt{N})$. That is, in the worst case this approach computes a separator of twice the size of the worst-case separator computed by Lipton and Tarjan's algorithm. We hope that computing a relaxed BFS-tree is easier than computing a BFS-tree, possibly using randomization.

For the shortest path preprocessing algorithm, even the internal-memory separator algorithm run in external memory is good enough; but an I/O-efficient separator algorithm is of interest in its own right.

Finally, we want to mention some open questions that immediately come to mind when reading all the previous chapters thoroughly: Can the tree-blocking approach be generalized to hierarchical directed acyclic graphs (HDAGs)? Does it help, if we restrict ourselves to planar HDAGs? As we have seen, for offline traversals of trees, it does not matter very much whether these traversals are top-down or bottom-up; but it makes a difference whether or not they modify the stored data. If they do, all copies of the modified tree vertex have to be updated. Is it possible to store trees

in a way such that we can achieve a speed-up better than $O(\log_d DB)$ for modifying offline traversals?

In the motivation for developing I/O-efficient algorithms, the increasing popularity of parallel computation is quoted. Therefore, parallel solutions for the problems considered in this work would be of interest. Having efficient BSP-, BSP*-, or CGM-algorithms (see [34] for the definition of the BSP-model, [8, 9, 10] for the BSP*-model, and [14] for the CGM-model), we could use the simulation of such algorithms on EM-BSP, EM-BSP*, or EM-CGM machines as proposed by Dehne, Dittrich, and Hutchinson [15]. Although it is not included in this thesis, we have shown already that our optimal blocking scheme for offline bottom-up traversals in rooted trees can easily be modified to allow for efficient (with regard to I/O and communcation) fully parallelized path traversals on an EM-BSP machine.

Appendix A

Deutsche Zusammenfassung

Die vorliegende Arbeit beschäftigt sich mit dem Problem, Graphen mit $\sqrt{N}$-Separatoren für die schnelle Berechnung kürzester Wege zwischen Knoten in diesen Graphen vorzuverarbeiten. Wir stellen eine External-Memory-Version der Kürzeste-Wege-Datenstruktur von Djidjev [16] vor.

Diese Datenstruktur kann zum Beispiel in geographischen Informationssystemen Anwendung finden, da in diesem Anwendungsgebiet solch riesige Datenmengen verarbeitet werden, daß oft nur ein kleiner Bruchteil der Daten in den Hauptspeicher des Computers paßt. Der Rest ist auf Festplatte gespeichert und muß in den Hauptspeicher geladen werden, wenn er benötigt wird. In solchen Szenarien ist der Flaschenhals der Berechnung nicht die eigentliche Berechnung, sondern der Datentransfer zwischen Hauptspeicher und Externspeicher. Deswegen ist es sinnvoll, Algorithmen zu entwikkeln, die die Anzahl der I/O-Operationen minimieren, möglicherweise sogar auf Kosten etwas höheren Rechenaufwandes oder Speicherbedarfs.

In Abschnitt 1.2 stellen wir die Spielregeln für den Entwurf I/O-effizienter Algorithmen vor. Diese werden uns durch die I/O-Modelle von Aggarwal und Vitter [2] und Vitter und Shriver [35] gegeben. In diesen Modellen nehmen wir an, daß Berechnungen im Hauptspeicher kostenlos sind, und wir versuchen die Anzahl der I/O-Operationen, die unser Algorithmus ausführt, zu minimieren. Der wichtigste Parameter dieser Modelle ist die Blockgröße B des externen Speichermediums. Die Übertragung eines Blockes benötigt eine I/O-Operation.

Das angestrebte Ideal ist nun, wenn man einen RAM-Algorithmus $\mathcal{A}$ mit Laufzeit $T_{\mathcal{A}}(N)$ hat, einen External-Memory-Algorithmus $\mathcal{A}'$ mit I/O-Komplexität $\mathcal{I}_{\mathcal{A}'}(N) = O(T_{\mathcal{A}}(N/B))$ oder besser zu finden. In Kapitel 2 stellen wir eine nach diesem Maßstab optimale External-Memory-Version der Kürzeste-Wege-Datenstruktur in [16] vor. Allerdings ist der vorgestellte Konstruktionsalgorithmus nicht optimal.

Die Datenstruktur und der Konstruktionsalgorithmus benutzen effiziente Lösungen für bestimmte Teilprobleme. Für die Datenstruktur ist dies eine Methode, Kürzeste-Wege-Bäume so im Externspeicher zu speichern, daß die Traversierung eines Pfades von einem beliebigen Knoten in einem solchen Baum zur Wurzel des Baumes wenige I/O-Operationen erfordert. Wir entwickeln eine solche optimale Methode in Kapitel 3. “Optimal” heißt hier, daß der auf diese Weise gespeicherte Baum T $O(|T|)$ Speicherplatz benötigt und die Traversierung eines Pfades der Länge K in T $O(K/B)$ I/O-Operation erfordert.

Für den Konstruktionsalgorithmus benötigen wir die folgenden Zutaten:

1. einen Single-Source-Shortest-Paths-Algorithmus,
2. einen Separator-Algorithmus für eingebettete planare Graphen und
3. einen Algorithmus, der die Kürzeste-Wege-Bäume in der Datenstruktur entsprechend der Methode aus Kapitel 3 speichert.

Als Single-Source-Shortest-Path-Algorithmus benutzen wir eine External-Memory-Version von Dijkstras Algorithmus, die von Crauser, Mehlhorn und Meyer [13] entwickelt wurde. Dieser Algorithmus ist nicht optimal und der Grund, warum der Konstruktionsalgorithmus in Kapitel 2 nicht I/O-optimal ist. Allerdings sind keine besseren Lösungen bekannt. Die Berechnung kürzester Wege scheint ein extrem hartes Problem in Bezug auf I/O-Komplexität zu sein. Selbst für Breitensuche, die als Spezialfall der Berechnung kürzester Wege aufgefaßt werden kann, existieren keine wirklich guten Lösungen.

Breitensuche ist auch der Flaschenhals in unserem Separator-Algorithmus, den wir in Kapitel 6 vorstellen. Genauer zeigen wir, daß man mit $O(sort(N))$ I/O-Operationen einen $\sqrt{N}$-separator in einem eingebetteten planaren Graphen berechnen

kann, wenn man einen BFS-Baum gegeben hat. Anderenfalls braucht die Berechnung $O(bfs(N))$ I/O-Operationen, wobei $bfs(N)$ die zur Berechnung eines BFS-Baums eines planaren Graphen benötigte Anzahl von I/O-Operationen ist. Unter Verwendung der besten bekannten Algorithmen für Breitensuche [25, 24] ist $bfs(N) = \Omega(N)$ im schlechteste Fall.

Ein Teilschritt unseres Separatoralgorithmus ist die Triangulierung des gegebenen Graphen. In Kapitel 7 stellen wir einen Algorithmus für dieses Problem vor.

In Kapitel 5 stellen wir einen I/O-effizienten Algorithmus vor, der die Kürzeste-Wege-Bäume unserer Kürzeste-Wege-Datenstruktur entsprechend der Methode aus Kapitel 3 speichert.

Die Algorithmen in den Kapiteln 5, 6 und 7 verwenden verschiedene gängige Markierungen wie Preorder-Nummern u.ä. für die Knoten in Wurzelbäumen. In Kapitel 4 entwickeln wir praktisch optimale Algorithmen, die diese Markierungen berechnen.

Kapitel 8 gibt einen Ausblick auf interessante offene Fragen für zukünftige Forschung, die im Zusammenhang mit den Ergebnissen dieser Arbeit stehen. Das Hauptaugenmerk liegt hier auf der Suche nach einer Möglichkeit, die Berechnung eines BFS-Baums in unserem Separator-Algorithmus zu umgehen.

Bibliography

[1] A. Aggarwal and R.J. Anderson. A random NC algorithm for depth first search. *Combinatorica*, 8(1):1–12, 1988.

[2] Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, pages 1116–1127, September 1988.

[3] Ljudmil Aleksandrov and Hristo Djidjev. Linear algorithms for partitioning embedded graphs of bounded genus. *SIAM Journal of Discrete Mathematics*, 9:129–150, 1996.

[4] B. Alpern, L. Carter, E. Feig, and Selker. The uniform memory hierarchy model of computation. *Algorithmica*, 12:72–109, 1994.

[5] Lars Arge. The buffer tree: A new technique for optimal I/O-algorithms. In *Proceedings of the Workshop on Algorithms and Data Structures*, volume 955 of *Lecture Notes in Computer Science*, pages 334–345, 1995.

[6] Lars Arge. *Efficient External-Memory Data Structures and Applications*. PhD thesis, Department of Computer Science, University of Aarhus, Aarhus/ Denmark, February 1996.

[7] Lars Arge, Mikael Knudsen, and Kirsten Larsen. A general lower bound on the I/O-complexity of comparison-based algorithms. In *Proceedings of the Workshop on Algorithms and Data Structures*, volume 709 of *Lecture Notes in Computer Science*, pages 83–94, 1993.

[8] A. Bäumker and W. Dittrich. Fully dynamic search for an extension of the BSP model. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 233–242, 1996.

[9] A. Bäumker, W. Dittrich, and F. Meyer auf der Heide. Efficient parallel algorithms: c-optimal multisearch for an extension of the BSP model. In *Proceedings of the Annual European Symposium on Algorithms*, pages 17–30, 1995.

[10] A. Bäumker, W. Dittrich, and A. Pietracaprina. The deterministic complexity of parallel multisearch. In *Proceedings of the Scandinavian Workshop on Algorithms Theory*, pages 404–415, 1996.

[11] Joseph Cheriyan, Ming-Yang Kao, and Ramakrishna Thurimella. Scan-first search and sparse certificates: An improved parallel algorithm for k-vertex connectivity. *SIAM Journal on Computing*, 22(1):157–174, February 1993.

[12] Yi-Jen Chiang, Michael T. Goodrich, Edward F. Grove, Roberto Tamassia, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory graph algorithms. In *Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms*, January 1995.

[13] Andreas Crauser, Kurt Mehlhorn, and Ulrich Meyer. Kürzeste-Wege-Berechnung bei sehr großen Datenmengen. In Otto Spaniol, editor, *Promotion tut not: Innovationsmotor "Graduiertenkolleg"*, volume 21 of *Aachener Beiträge zur Informatik*. Verlag der Augustinus Buchhandlung, 1997.

[14] Frank Dehne, Andreas Fabri, and Andrew Rau-Chaplin. Scalable parallel computational geometry for coarse grained multicomputers. *International Journal on Computational Geometry*, 6(3):379–400, 1996.

[15] Franke Dehne, Wolfgang Dittrich, and David Hutchinson. Efficient external memory algorithms by simulating coarse-grained parallel algorithms. In *Proceedings of the 9th ACM Symposium on Parallel Algorithms and Architectures*, pages 106–115, 1997.

[16] Hristo N. Djidjev. Efficient algorithms for shortest path queries in planar digraphs. In *Proceedings of the 22nd Workshop on Graph-Theoretic Concepts in Computer Science*, Lecture Notes in Computer Science, pages 151–165. Springer Verlag, 1996.

[17] Hristo N. Djidjev. Partitioning graphs with costs and weights on vertices: Algorithms and applications. volume 1284 of *Lecture Notes in Computer Science*, pages 130–143. Springer Verlag, 1997.

[18] Hristo N. Djidjev, Grammati E. Pantziou, and Christos D. Zariolagis. On-line and improved algorithms for dynamic shortest paths. In *Proceedings of the 12th Symposium on Theoretical Aspects of Computer Science (STACS '95)*, volume 900 of *Lecture Notes in Computer Science*, pages 193–204. Springer-Verlag, 1995.

[19] Esteban Feuerstein and Alberto Marchetti-Spaccamela. Dynamic algorithms for shortest paths in planar graphs. *Theoretical Computer Science*, 116:359–371, 1993.

[20] Hillel Gazit and Gary L. Miller. A parallel algorithm for finding a separator in planar graphs. In *Proceedings of the 28th Symposium on Foundations of Computer Science*, pages 238–248, 1987.

[21] Hillel Gazit and Gary L. Miller. An improved parallel algorithm that computes the BFS numbering of a directed graph. *Information Processing Letters*, 28:61–65, 1988.

[22] Michael T. Goodrich. Planar separators and parallel polygon triangulation. In *Proceedings of the 24th ACM Symposium on Theory of Computing*, pages 507–516, 1992.

[23] Torben Hagerup. Planar depth-first search in $O(\log n)$ parallel time. *SIAM Journal on Computing*, 19(4):678–704, August 1990.

[24] M.V. Kameshwar and Abhiram Ranade. I/O-complexity of graph algorithms. In *10th Annual ACM-SIAM Symposium on Discrete Algorithms*, 1999.

[25] Vijay Kumar and Eric J. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proceedings of the 8th IEEE Sumposium on Parallel and Distributed Computing*, October 1996.

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

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

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

[29] M.H. Nodine and J.S. Vitter. Paradigms for optimal sorting with multiple disks. In *Proceedings of the 26th Hawaii International Conference on Systems Sciences*, January 1993.

[30] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–28, 1994.

[31] Justin R. Smith. Parallel algorithms for depth-first searches I. planar graphs. *SIAM Journal on Computing*, 15(3):814–830, August 1986.

[32] Roberto Tamassia and Franco P. Preparata. Dynamic maintenance of planar digraphs, with applications. *Algorithmica*, 5:509–527, 1990.

[33] Roberto Tamassia and Jeffrey S. Vitter. Parallel transitive closure and point location in planar structures. *SIAM Journal on Computing*, 20(4):708–725, August 1991.

[34] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103, August 1990.

[35] J.S. Vitter and E.A.M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2–3):110–147, 1994.

[36] J.S. Vitter and E.A.M. Shriver. Algorithms for parallel memory II: Hierarchical multilevel memories. *Algorithmica*, 12(2–3):148–169, 1994.